Science of Computer Programming 18 (1992) 181-204
Elsevier

Balanced trees with removals: an exercise in rewriting and proof

C.M.P. Reade

Department of Computer Science, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK; and FMG, SED, Informatics Department, Rutherford Appleton Laboratory, Chilton, Didcot, OX11 0QX, UK

Communicated by R.S. Bird
Received May 1991
Revised November 1991

Abstract

Reade, C.M.P., Balanced trees with removals: an exercise in rewriting and proof, Science of Computer Programming 18 (1992) 181-204.

An equational algorithm to remove values from 2-3-trees is given along with a novel proof technique for use in showing correctness. The removal algorithm involves rewrite rules which ensure balance is restored in trees and uses similar methods to those used by Hoffman and O'Donnell for an insertion algorithm. However, the combination of equational rewrite steps is more subtle for removals and the algorithm is less obviously correct. Diagrams are used to show informally how the rewriting steps preserve order and balance and a method for formalising a correctness proof is also shown. This formalisation involves proofs with "subtypes" in the sense of sets of values of the same type which aid the derivation of properties of auxiliary functions.

1. Introduction

An elegant equational program for inserting values into 2-3-trees is provided by Hoffman and O'Donnell [6]. Their algorithm involves equational rewrite rules which ensure that the balance and order of a tree is restored when a new value is inserted. A solution to the problem of removing values from 2-3-trees is presented here which requires a more complex combination of equational rewrite rules. The subtlety of the algorithm forces a more careful consideration of correctness than seems necessary for the simpler insertion algorithm and a method of proof (involving typed sets of values) is introduced for showing the formal correctness of both algorithms.

Correspondence to: C.M.P. Reade, Department of Computer Science, Brunel University, Uxbridge, Middlesex, UB8 3PH, UK. Email: Chris.Reade@brunel.ac.uk

2. Two-three-trees, equational programs and insertions

2.1. *Two-three-trees*

The algorithms concern 2-3-trees (see Fig. 1) where each tree is either empty (E) or a 2-node of the form Tr2(t1,a,t2) or a 3-node of the form Tr3(t1,a,t2,b,t3) and trees are *ordered* and *balanced*:

Definition. Assuming node items are drawn from a totally ordered set S, a tree t (constructed from Tr2, Tr3 and E) is *ordered* if $\exists i,j \in S$ $(i<j)$ such that t has ordered node items (strictly) between i and j, where: E *has ordered node items between* i *and* j if $i<j$; Tr2(t_1,a,t_2) *has ordered node items between* i *and* j if $i<a<j$ and t_1 has ordered node items between i and a and t_2 has ordered node items between a and j; Tr3(t_1,a,t_2,b,t_3) *has ordered node items between* i *and* j if $i<a<b<j$ and t_1 has ordered node items between i and a and t_2 has ordered node items between a and b and t_3 has ordered node items between b and j.

Definition. A tree t (constructed from Tr2, Tr3 and E) is *balanced* if $\exists$ integer $k \geq 0$ such that t is balanced with depth k, where E *is balanced with depth* 0; Tr2(t_1,a,t_2) *is balanced with depth* $k+1$ if t_1 and t_2 are balanced with depth k; Tr3(t_1,a,t_2,b,t_3) *is balanced with depth* $k+1$ if t_1, t_2 and t_3 are balanced with depth k.

In the sequel we will assume that the node items are integers for simplicity, and write tree23 for the type of all integer trees formed from the constructors (not necessarily ordered or balanced).

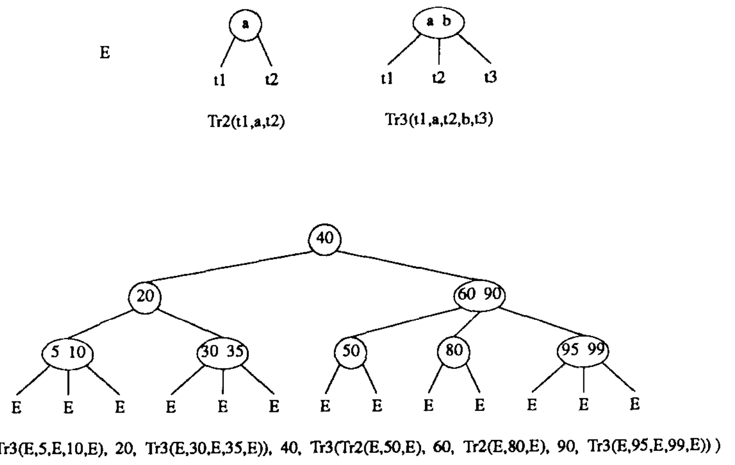
E
a
t1 t2
Tr2(t1,a,t2)
a b
t1 t2 t3
Tr3(t1,a,t2,b,t3)
40
20
60 90
5 10
30 35
50
80
95 99
E E E E E E E E E E E E E
Tr2(Tr2(Tr3(E,5,E,10,E), 20, Tr3(E,30,E,35,E)), 40, Tr3(Tr2(E,50,E), 60, Tr2(E,80,E), 90, Tr3(E,95,E,99,E)))

Fig. 1. Diagrams for 2-3-trees.

2.2. Insertions

Appendix A contains an equational program for the insertion of a value into a 2-3-tree based on the one described by Hoffman and O'Donnell [6]. An extra constructor

$$\mathsf{Put}:\mathsf{tree23}\times\mathsf{int}\times\mathsf{tree23}\rightarrow\mathsf{tree23}$$

is used in the algorithm and may be created during an insertion, but subsequently removed during rebalancing. Put nodes (of the form Put(t1,a,t2)) are inserted at a leaf by a put function (distinct from the Put constructor) which also replaces Tr2 and Tr3 constructors by functions tr2 and tr3 as it descends the tree. The functions tr2 and tr3 restore balance by (respectively) absorbing or bubbling up the Put node and creating normal Tr2 and Tr3 nodes. (A sentinal function checktop absorbs Put nodes reaching the top of the tree, increasing tree depth by 1.)

Figure 2 gives a graphical description of the rewrite rules for put, tr2 and tr3 where we have represented applications of the latter two functions as boxes. (Error cases are omitted.) An expression of the form tr2(t1,a,t2) can be thought of as an active 2-node that may rewrite to something else, where Tr2(t1,a,t2) is a passive 2-node data object. Similarly tr3 can be thought of as an active version of Tr3.

2.3. Equational and functional programs

The insertion algorithm is (almost) presented as an equational program of a particular form which has come to be called a *constructor system of equations* (Thatte [11]). For such systems, the equations adhere to a convention whereby *active* (defined) functions are distinguished from *passive* (data-constructor) functions. On the left-hand side of an equation the expression always has the form $f(E_1, \ldots, E_n)$ where f is active and the E_i only involve variables and passive functions or constants (i.e. the E_i are patterns). Furthermore variables are not repeated on the left-hand side of an equation and variables on the right of an equation are always introduced on the left. The distinction between constructors and defined functions is advocated for equational definitions for technical reasons even though it is not a necessary restriction on the more general form of equational programs (see [8, Section 12.1]).

One difference in conventions between equational and functional forms is that in the former the equations form an unordered set (so that left-hand sides should cover distinct cases) whereas in the latter overlaps are resolved by taking the (textually) first case to apply. In practice it is more convenient to write definitions with the latter convention which we have adopted for the equations given here. We can convert from the overlapping form to provide a set of unordered equations (see for example [12] for details of such an algorithm). In all the examples given here, the conversion amounts to simply splitting cases which overlap into a collection of subcases (obtained by enumerating constructions for the variables which overlap) and deleting overridden cases.

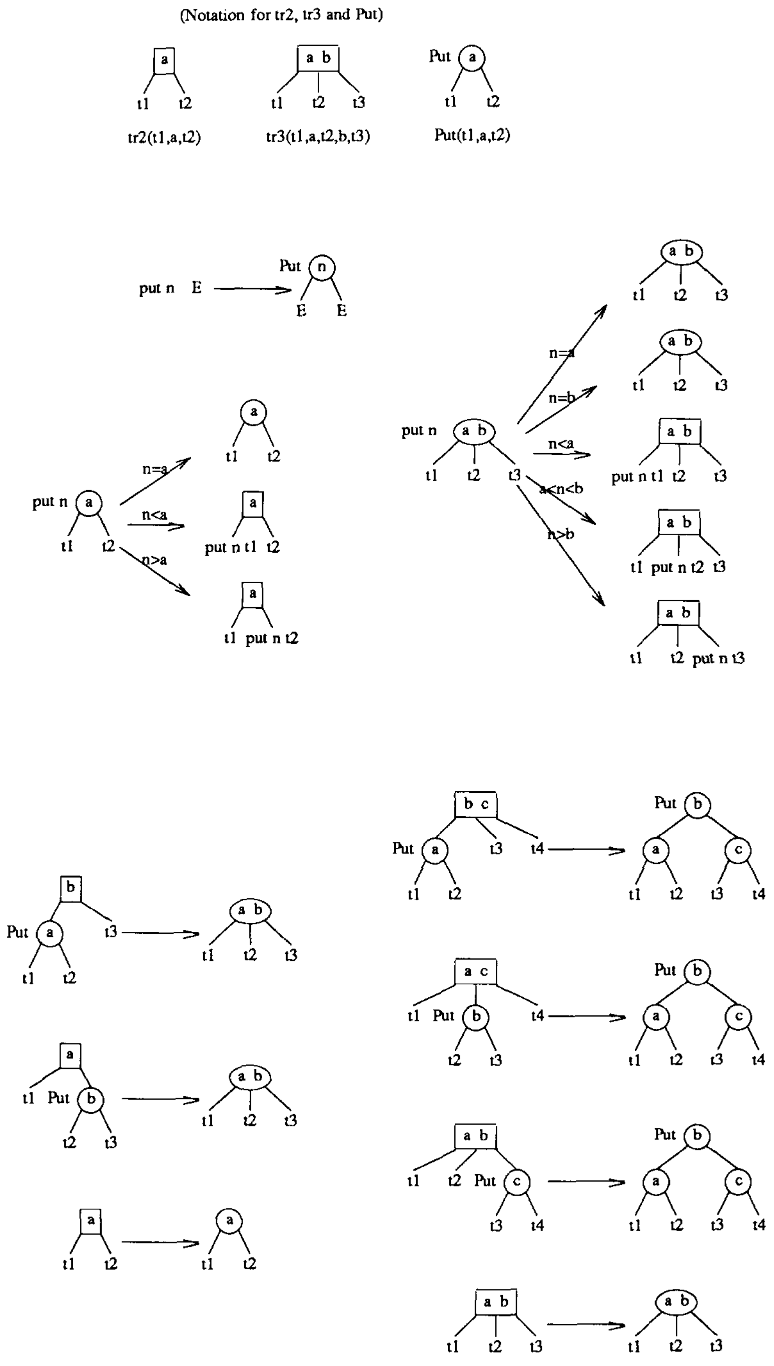

Fig. 2. Rewrite rules for put, tr2 and tr3.

3. A remove operation

When an item is removed from a tree, subtrees have to be combined while order and balance are preserved. We seek an algorithm which avoids extra scanning of subtrees (e.g. to calculate depth) and which avoids nesting auxiliary constructors (such as Put(Put(...))). In the worst case, nesting could increase the number of rewrites needed to restore balance from O(log n) to $O((\log n)^2)$, where n is the number of nodes in the tree. It can also increase the complexity and number of cases in pattern matching. We show a particular algorithm here making use of an additional constructor Taken and discuss variations later.

3.1. Taken nodes

The constructor

Taken : tree23 → tree23

is introduced to signify a subtree with depth one less than that required for a balance. The functions tr2, tr3 and checktop can be modified to absorb Taken nodes as well as Put nodes, as they restore the balance. The additional rules for tr2 and tr3 are given in Appendix B and presented diagrammatically in Figs. 3 and 4. Note that not all combinations of Taken and Put nodes are dealt with because, as we will see later, not all cases will arise. tr2 rewrites only deal with a single possible Taken node in an immediate subtree, whereas tr3 rewrites deal with up to three Taken nodes in the three immediate subtrees (in parallel but not nested below each other). The first rule to apply should be chosen when there are overlaps (see remarks in Section 2.3).

Notice that in the diagrams, one can check by eye that the height of a subtree is preserved by the rewrite, and hence that balance is being preserved as Taken nodes are absorbed or pushed up the tree. This acts as an informal check and is the basis for a formal argument presented in Section 4.

3.2. Removals

Appendix C contains the equations for the main removal operation remove23 which involves the auxiliary functions: merge, remove, leftPut and rightPut (as well as tr2, tr3 and checktop). (The insert23 and put functions for this version are exactly as before and do not introduce any Taken nodes.) One of the design decisions for this removal algorithm is that the function merge : tree23 × tree23 → tree23 should create an extra level when combining two subtrees with the same depth. In Fig. 5, the merge operation is depicted with a double bar linking the two trees being merged. Once again, one can check by eye that the tree on the right-hand side of each rule is indeed one level greater than the argument trees on the left-hand side (i.e. at the level of the double bar) and the subtrees preserve their depth. Taken nodes are introduced where necessary to keep subtrees at the correct relative level.

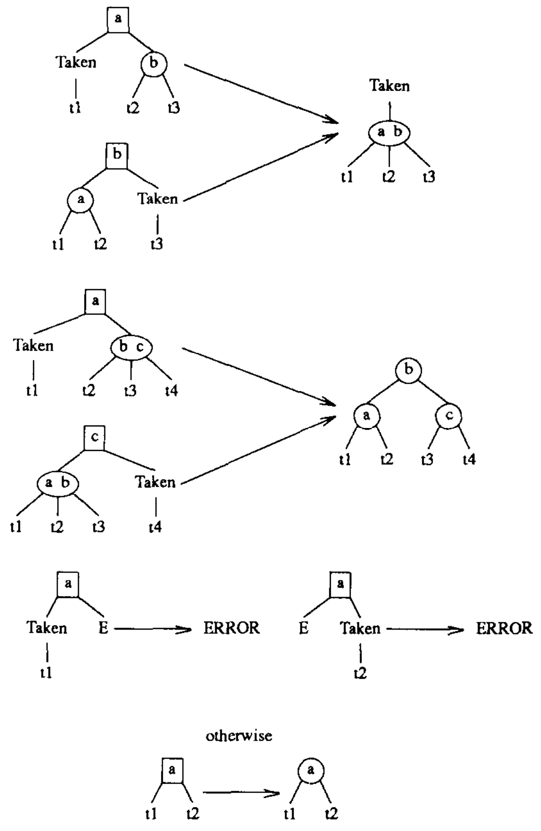

Fig. 3. Additional rules for tr2.

The function remove descends a tree to find an item to be removed, replacing visited Tr2 and Tr3 nodes by tr2 and tr3 nodes for rebalancing. When the item is found at a node, subtrees are combined by merging. This is a simple merge if the item is found in a 2-node and the merge will create an extra level to replace the one removed. If the item is found at a 3-node, a more complex merge takes place using leftPut or rightPut to combine the merged subtree with the remains of the partly removed 3-node. The rewrite rules of remove, leftPut and rightPut are depicted in Fig. 6.

In order to understand some of the design decisions, the reader should note the following: merge can only produce (a normal tree or) a tree with a Taken node as root, but not a tree with a Put node as root. Similarly, put can only produce (a normal tree or) a tree with a Put node as root, but not a tree with a Taken node as root. remove can produce a Taken or a Put (or a normal tree) and leftPut and rightPut expect a normal tree or a possible Taken as root of the merge argument to produce a normal tree or possibly a tree with Put as root. Thus leftPut and rightPut provide a crossover boundary which separates Takens and Puts. A Taken

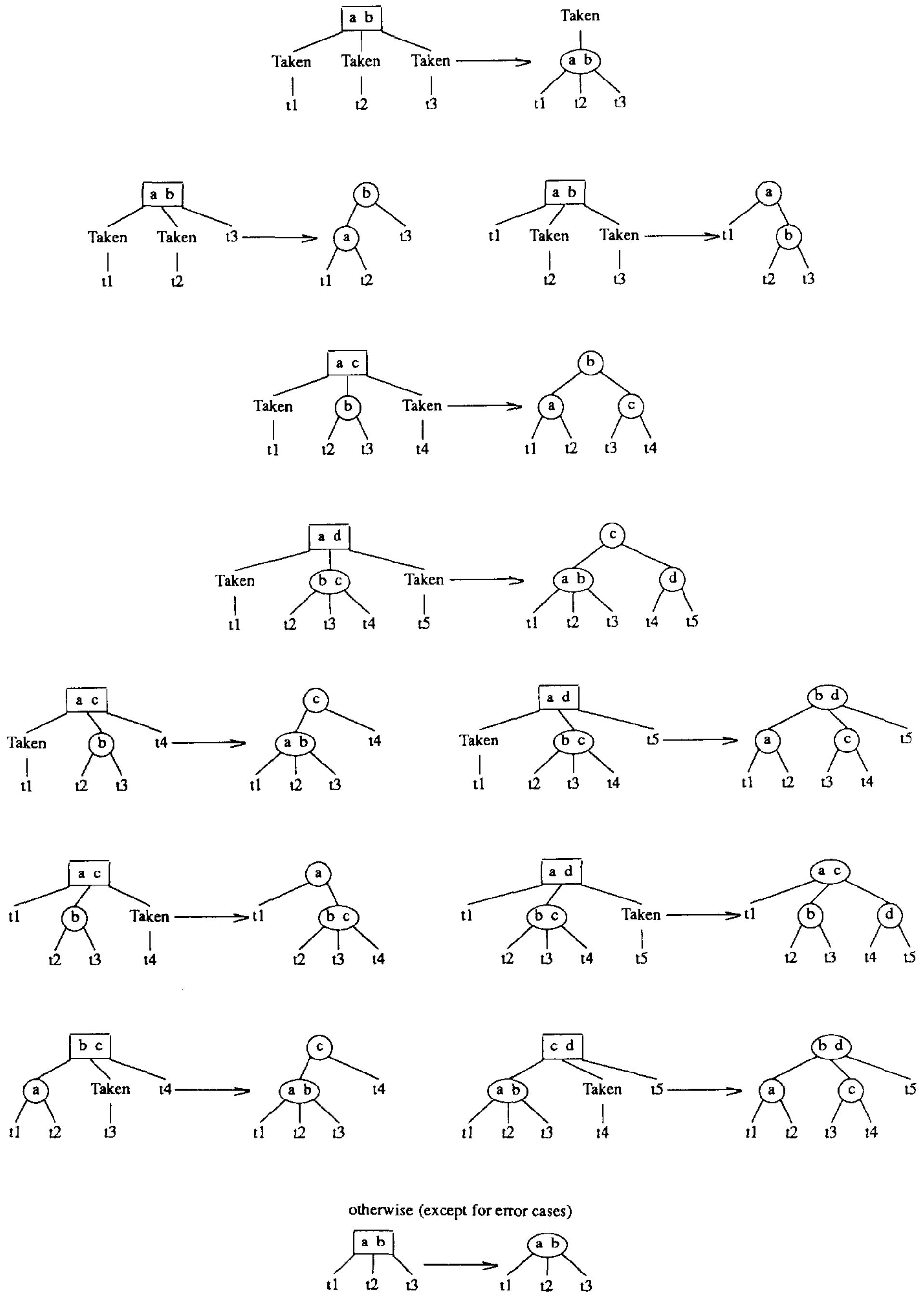

Fig. 4. Additional rules for tr3.

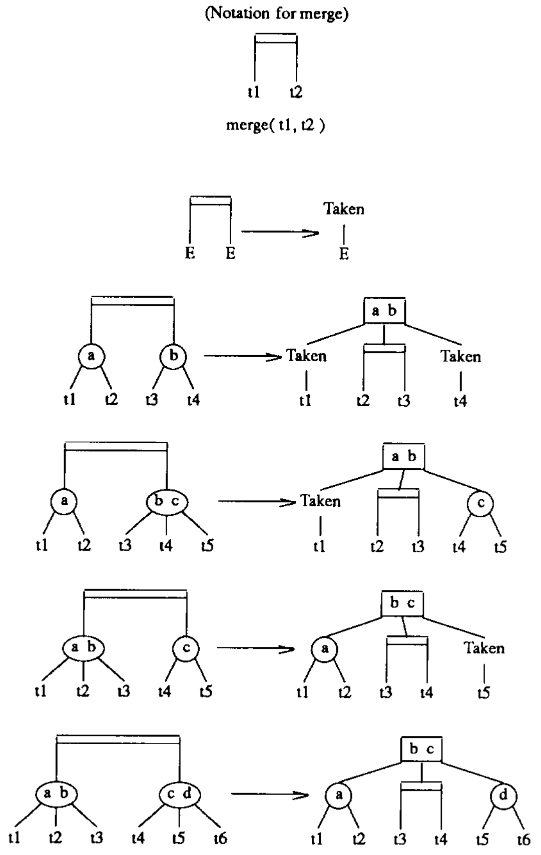

Fig. 5. Rewrite rules for merge.

bubbling up from a merge will be caught and removed by one of these, but the lack of a Taken bubbling up may cause the generation of a Put. This is then dealt with by higher tr3 and tr2 nodes. tr2 and tr3 need to be able to deal with either a single Put arising as (the root of) a subtree or Taken subtrees (but not both). In the case of tr2, only one argument can be rooted by Taken but up to three of tr3's, arguments can be rooted by Taken. A proof that all other cases (of both a Taken and a Put or nested Takens or two Puts) cannot arise, is required to justify the algorithm. We explore this in Section 4.

3.3. *Possible variations*

There are several alternatives which could be used, but the conventions about no nested auxiliary constructors may not hold when some alternatives are used. Several alternative solutions to the one presented were explored. For example, Put nodes can be generalised to allow three as well as two subtrees instead of introducing

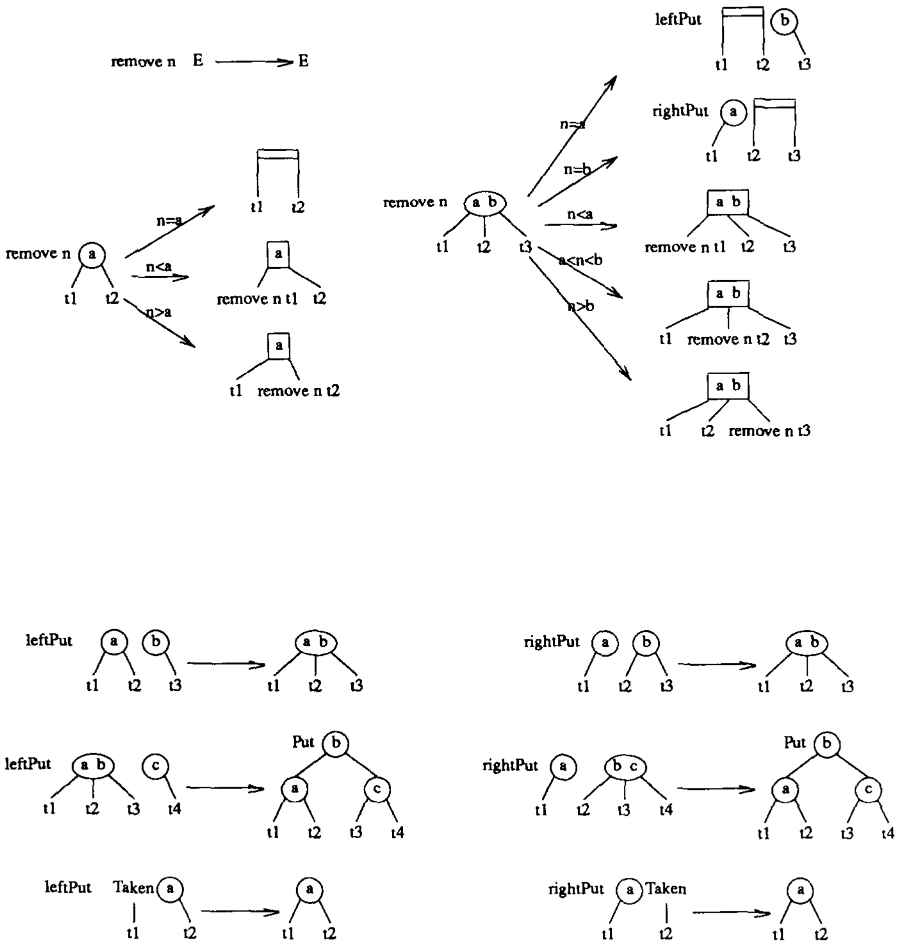

Fig. 6. Rewrite rules for remove, leftPut and rightPut.

Taken nodes and it is even possible to avoid the use of Put using Taken nodes in the definition of put as well as remove. The convention for such a version was that put increased tree depth by one (as the default) but there is a problematic case forcing us to consider nested Taken nodes. The functions tr2, tr3 and checktop can be adapted by ensuring that they remain active after rewriting (in some cases at least), or we could introduce tr2′, tr3′ and checktop′ to deal with the second level of Taken nodes which can arise.

In all the alternatives found, it seemed necessary to deal with nested auxiliary constructors. The chosen implementation is a careful selection of rules which work together, keeping to certain conventions and avoiding the need for nested Taken and/or Put nodes.

A different method for combining trees is to "pull up" a replacement value for the value being removed. This is discussed in [9] where it is shown to require only a small change to the definitions of the "merge" version.

3.4. *An abstract type intset*

Since there are several conventions about 2-3-trees and the functions applicable to them (i.e. keeping the trees balanced and ordered), the integrity of the trees should be protected by using an abstract type. We choose a type intset for (finite) integer sets.[1] Values of the abstract type intset are represented by values of type tree23 but integrity is guaranteed (e.g. by restricting the scope of an outer constructor Set). For example, using Standard ML notation:

```
abstype intset = Set of tree23
with
  emptyset             = Set E
  insertset n (Set t)  = Set(insert23 n t)
  removeset n (Set t)  = Set(remove23 n t)
  memberset n (Set t)= member23 n t
end
```

where member23 : int → tree23 → bool tests to see if a value is contained in a tree. Since it does not produce a tree, this function is not concerned with preservation of balance and order, but it does rely on the tree being ordered when scanning. The use of an abstract type to restrict scope ensures that only balanced, ordered trees (with normal nodes Tr2, Tr3, E) can be created in representations of intset values.

4. Proof of correctness

In order to show that the insertion and removal algorithms are correct, we need to establish that they preserve the invariant properties of trees being ordered and balanced (as well as performing the correct abstract functions of inserting or removing). This is the usual proof of correctness of an abstract type implementation first expressed by Hoare [5], but the difficulty faced here is that in order to establish the invariant property, many more intricate properties of auxiliary functions are relied upon. Formalising and establishing these properties are the main challenge.

4.1. *Subtype sets*

In order to formalise what is to be proved, some notation is introduced for classifying sets of values of the same type. These sets will be referred to as *subtype sets* and are equivalent to predicates on a type (the characteristic function of the

[1] Another application is given in [9] where 2-3-trees are used to implement a dictionary module where node values consist of a pair of a key and associated information. The type of associated information is arbitrary and the type of the keys has an ordering and is passed as a module parameter.

set) but set notation is more convenient. We extend the notation of product types and function types to subsets of such types as follows:

Definition 4.1. If X_i is a subtype set of type T_i $(i=1..n)$, then $X_1 \times X_2 \times \cdots \times X_n$ is the subtype set of type $T_1 \times T_2 \times \cdots \times T_n$ consisting of the values:

$$\{(x_1, x_2, \ldots, x_n) \mid x_1 \in X_1, x_2 \in X_2, \ldots, x_n \in X_n\}.$$

Definition 4.2. If X_i is a subtype set of type T_i $(i=1,2)$, then $X_1 \rightarrow X_2$ is a subtype set of type $T_1 \rightarrow T_2$, given by

$$X_1 \rightarrow X_2 \;\equiv\; \{f : T_1 \rightarrow T_2 \mid \forall x \in X_1 \bullet f(x) \in X_2\}.$$

That is, $X_1 \rightarrow X_2$ is the set of functions f of type $T_1 \rightarrow T_2$ such that when f is applied to a value in X_1 it produces a result in X_2. (This notation also makes sense with higher-order functions. For example, $f \in X_1 \rightarrow X_2 \rightarrow X_3$ means $f : T_1 \rightarrow T_2 \rightarrow T_3$ and given any $x \in X_1$, $f x \in X_2 \rightarrow X_3$. Thus given $y \in X_2$, $f x y \in X_3$.)

Some basic rules, properties and useful derived rules concerning subtype sets are summarised in Appendix D. Only sets, all of whose members have the same type, are considered, so the notation $s \in S$ is only used when the underlying type of S is the type of s. In the sequel we use juxtaposition of sets to denote binary intersection: $XY = X \cap Y$ (e.g. in (D.5) of Appendix D) and by default, we take the intersection of the empty collection of sets to be the full set of values of the type associated with the collection.

To begin with, the following sets of trees are introduced, *normal* or *normalised* trees, by which we mean (finite and total) balanced trees constructed only from E, Tr2, Tr3 and integers (i.e. excluding Put and Taken), along with similar sets which also allow a root node to be Put or Taken.

Definition (Normalised trees of depth k). Let $\mathcal{N}_k$, for $k \geq 0$, be the sets defined inductively by:

$$\begin{aligned}
&\mathcal{N}_0 = \{E\},\\
&\mathcal{N}_{k+1} = \{Tr2(t1, a, t2) \mid a \in \mathcal{Z},\ t1, t2 \in \mathcal{N}_k\}\\
&\qquad\cup\\
&\qquad \{Tr3(t1, a, t2, b, t3) \mid a, b \in \mathcal{Z},\ t1, t2, t3 \in \mathcal{N}_k\},
\end{aligned}$$

where $\mathcal{Z}$ has been used to denote the set of integers (well-defined values of type int). For later convenience, we will also define $\mathcal{N}_{-1} = \{\}$. The set of all normalised trees is denoted

$$\mathcal{N} \in \bigcup \{\mathcal{N}_i \mid i \geq 0\}.$$

Definition (Possible Put trees of depth k). Let $\mathcal{P}_k$, for $k \geq 0$, be the set of normalised trees of depth k extended to allow for a possible Put node at the root where the immediate subtrees are normal trees of depth k:

$$\mathcal{P}_k = \mathcal{N}_k \cup \{Put(t1, a, t2) \mid a \in \mathcal{Z},\ t1, t2 \in \mathcal{N}_k\}.$$

Definition (Possible Taken trees of depth k). Let $\mathcal{T}_k$, for $k \geq 0$, be the set of normalised trees of depth k extended to allow for a possible Taken node at the root where the subtree is normalised with depth $k-1$:

$$\mathcal{T}_k = \mathcal{N}_k \cup \{\text{Taken}(t) \mid t \in \mathcal{N}_{k-1}\} \quad (\text{for } k \geq 0).$$

In particular $\mathcal{T}_0 = \mathcal{N}_0 \cup \{\} = \{E\}$. Clearly, we have for each $k \geq 0$

$$\mathcal{N}_k \subseteq \mathcal{T}_k, \qquad \mathcal{N}_k \subset \mathcal{P}_k, \qquad \mathcal{N}_k = \mathcal{P}_k \mathcal{T}_k \quad (= \mathcal{P}_k \cap \mathcal{T}_k).$$

Now, for example, $\mathcal{N}_k \to \mathcal{T}_k$ means the set of functions of type tree23 → tree23, which when applied to a tree in $\mathcal{N}_k$, produces a tree in $\mathcal{T}_k$. So if $f \in \mathcal{N}_k \to \mathcal{T}_k$ for all $k \geq 0$, then f produces a possible Taken tree with the same depth as its argument tree (provided the argument tree is normal). It follows directly from the definitions of $\mathcal{N}_k$ that for $k \geq 0$

$$\text{Tr2} \in \mathcal{N}_k \times \mathcal{Z} \times \mathcal{N}_k \to \mathcal{N}_{k+1},$$

$$\text{Tr3} \in \mathcal{N}_k \times \mathcal{Z} \times \mathcal{N}_k \times \mathcal{Z} \times \mathcal{N}_k \to \mathcal{N}_{k+1}.$$

That is, when Tr2 is applied to a triple of a normalised tree, an integer and another normalised tree, of the same depth as the first (k), it produces a normalised tree of depth $k+1$.

We outline the structure of a proof of correctness of the implementation of abstract type intset using such subtype sets. (More detail can be found in [9].) The proof is separated out into three parts for clarity of exposition:

(i) showing that removal and insertion preserve balance (and absence of Put and Taken nodes);
(ii) showing that removal and insertion preserve order, and
(iii) showing that removal and insertion do remove and insert the required item and the membership if nothing else is affected.

In fact, proving the third of these properties requires that we consider remove23, insert23 and member23 simultaneously in showing the required abstract behaviour:

```
member23 n (insert23 n t)   = true
member23 n (insert23 m t)   = member23 n t,   if n ≠ m
member23 n (remove23 n t)   = false
member23 n (remove23 m t)   = member23 n t,   if n ≠ m
member23 n E                = false
```

for all integers n and m and all 2-3-trees t provided that they are of an appropriate form (i.e. balanced and ordered without Taken or Put nodes). Although we separate the proof into three parts, there is still a dependence of (ii) on (i) and of (iii) on (i) and (ii). Combining the parts into a single proof would avoid some repeated case analyses but stating and proving the goal for each case becomes more complex.

4.2. *Preservation of balance*

Preservation of balance can be expressed as

$$\text{remove23} \in \mathscr{X} \to \mathscr{N} \to \mathscr{N},$$
$$\text{insert23} \in \mathscr{X} \to \mathscr{N} \to \mathscr{N}.$$

These properties can be derived from some stronger results and properties of auxiliary functions listed in Appendix E. The proof of the main property of insert23 involves proving 12 other properties from Appendix E (about 30 cases in all) and one induction (for put). The proof of the main property for remove23 involves proving another 9 properties (about 60 new cases) with two more inductions (for merge and remove). The number of cases is determined by the number of equations for a function and the number of properties to be proved for that function. The order in which properties are proved (and the number of inductions) is determined by the dependency graph for the function definitions. (Any mutually dependent functions would involve simultaneous proofs of properties by induction.)

As an illustration, we show part of the proof that $\text{remove} \in \mathscr{X} \to \mathscr{N}_k \to (\mathscr{T}_k \cup \mathscr{P}_k)$ by induction on $k \geq 0$ assuming the properties listed in Appendix E for merge, leftPut, rightPut, tr2 and tr3 (for all $k \geq 0$).

Firstly, when $k = 0$,

$$\text{remove} \in \mathscr{X} \to \mathscr{N}_0 \to (\mathscr{T}_0 \cup \mathscr{P}_0)$$

follows immediately from the equation

$$\text{remove n E} = \text{E}$$

because $\mathscr{N}_0 = \{\text{E}\}$ and $\text{E} \in \mathscr{T}_0 \cup \mathscr{P}_0$. Secondly, we assume the induction hypothesis that

$$\text{remove} \in \mathscr{X} \to \mathscr{N}_k \to (\mathscr{T}_k \cup \mathscr{P}_k)$$

and establish that

$$\text{remove} \in \mathscr{X} \to \mathscr{N}_{k+1} \to (\mathscr{T}_{k+1} \cup \mathscr{P}_{k+1})$$

That is, for $\text{n} \in \mathscr{X}$ and $\text{t} \in \mathscr{N}_{k+1}$, $\text{remove n t} \in (\mathscr{T}_{k+1} \cup \mathscr{P}_{k+1})$. There are two cases to consider:

Case 1: $\text{t} = \text{Tr2(t1,a,t2)}$ where $\text{t1,t2} \in \mathscr{N}_k$ and $\text{a} \in \mathscr{X}$.

Case 1(a): $\text{n} = \text{a}$. Then, by definition,

$$\text{remove n t} = \text{merge(t1,t2)}.$$

The property stated for merge in Appendix E and the fact that $\text{t1,t2} \in \mathscr{N}_k$ mean that $\text{merge(t1,t2)} \in \mathscr{T}_{k+1}$. So $\text{remove n t} \in \mathscr{T}_{k+1}$ and hence $\text{remove n t} \in \mathscr{T}_{k+1} \cup \mathscr{P}_{k+1}$.

Case 1(b): $\text{n} < \text{a}$. Then, by definition,

$$\text{remove n t} = \text{tr2(remove n t1 ,a,t2)}.$$

The induction hypothesis and the fact that $\mathsf{t1} \in \mathcal{N}_k$ mean that $\mathsf{remove\ n\ t1} \in \mathcal{T}_k \cup \mathcal{P}_k$. By considering the two separate cases $\mathsf{remove\ n\ t1} \in \mathcal{T}_k$ and $\mathsf{remove\ n\ t1} \in \mathcal{P}_k$ along with the properties stated for tr2 in Appendix E, we get that either

$$\mathsf{tr2(remove\ n\ t1,a,t2)} \in \mathcal{T}_{k+1} \quad \text{or} \quad \mathsf{tr2(remove\ n\ t1,a,t2)} \in \mathcal{N}_{k+1}.$$

So

$$\mathsf{remove\ n\ t} \in \mathcal{T}_{k+1} \cup \mathcal{P}_{k+1}.$$

(Because $\mathcal{N}_{k+1} \subset \mathcal{T}_{k+1} \subset (\mathcal{T}_{k+1} \cup \mathcal{P}_{k+1})$).)

Case 1(c): $\mathsf{n} > \mathsf{a}$. (This is analogous to Case 1(b).)

Case 2: $\mathsf{t} = \mathsf{Tr3(t1,a,t2,b,t3)}$. This is proved similarly by considering five subcases ($\mathsf{n} = \mathsf{a}$, $\mathsf{n} = \mathsf{b}$, $\mathsf{n} < \mathsf{a}$, $\mathsf{a} < \mathsf{n} < \mathsf{b}$, $\mathsf{b} < \mathsf{n}$).

By induction, it follows that for all $\mathsf{k} \geq 0$, for all $\mathsf{n} \in \mathcal{Z}$ and $\mathsf{t} \in \mathcal{N}_k$,

$$\mathsf{remove\ n\ t} \in (\mathcal{T}_k \cup \mathcal{P}_k)$$

(i.e. that $\mathsf{remove} \in \mathcal{Z} \to \mathcal{N}_k \to (\mathcal{T}_k \cup \mathcal{P}_k)$ for all $\mathsf{k} \geq 0$—as required.)

4.3. *Proof of order preservation*

The fact that the functions remove23 and insert23 preserve order can be shown by a similar technique.

We begin by formalising the ordered trees as a subtype of tree23.

Definition. The set $\mathcal{O}$ of all ordered trees is given by

$$\mathcal{O} = \bigcup \{\mathcal{B}_{i,j} \mid \mathsf{i,j} \in \mathcal{Z}, \mathsf{i} < \mathsf{j}\}$$

where, for all $\mathsf{i,j} \in \mathcal{Z}$, $\mathsf{i} < \mathsf{j}$

$$\begin{aligned}\mathcal{B}_{i,j} = &\{\mathsf{Tr2(t1,a,t2)} \mid \mathsf{a} \in \mathcal{Z}, \mathsf{i} < \mathsf{a} < \mathsf{j}, \mathsf{t1} \in \mathcal{B}_{i,a}, \mathsf{t2} \in \mathcal{B}_{a,j}\}\\ &\cup\\ &\{\mathsf{Tr3(t1,a,t2,b,t3)} \mid \mathsf{a,b} \in \mathcal{Z}, \mathsf{i} < \mathsf{a} < \mathsf{b} < \mathsf{j}, \mathsf{t1} \in \mathcal{B}_{i,a}, \mathsf{t2} \in \mathcal{B}_{a,b}, \mathsf{t3} \in \mathcal{B}_{b,j}\}\\ &\cup\\ &\{\mathsf{E}\}.\end{aligned}$$

So $\mathcal{B}_{i,j}$ denotes the set of trees which are ordered and bounded by i and j as i and j range over all integers ($\mathsf{i} < \mathsf{j}$). This is a well-founded inductive definition for larger and larger intervals i..j, with base cases i..i+1. We have that $\mathcal{B}_{i,i+1} = \{\mathsf{E}\}$ and $\mathcal{B}_{i,j} \subseteq \mathcal{B}_{r,s}$ whenever $\mathsf{r} \leq \mathsf{i}$ and $\mathsf{j} \leq \mathsf{s}$ and for convenience, we also define

$$\mathcal{B}_{i,j} = \{\} \quad \text{whenever } \mathsf{j} \geq \mathsf{i}$$

The trees in $\mathcal{B}_{i,j}$ need not be balanced and they do not involve Taken and Put nodes. Accordingly, we extend the above sets $\mathcal{B}_{i,j}$ to allow for a Taken or Put node as root:

Definition (Extended, ordered trees). For $i,j \in \mathscr{Z}$

$$\begin{aligned}\mathscr{E}_{i,j} = \mathscr{B}_{i,j}& \\ &\cup \\ &\{\mathrm{Put}(t1,a,t2) \mid a \in \mathscr{Z}, i<a<j, t1 \in \mathscr{B}_{i,a}, t2 \in \mathscr{B}_{a,j}\} \\ &\cup \\ &\{\mathrm{Taken}(t1) \mid t1 \in \mathscr{B}_{i,j}\}.\end{aligned}$$

Writing {a} for the subset of $\mathscr{Z}$ containing just a, it follows from these definitions that for any $i,a,b,j \in \mathscr{Z}$:

$$\begin{aligned}&\mathrm{Tr2} \in \mathscr{B}_{i,a} \times \{a\} \times \mathscr{B}_{a,j} \to \mathscr{B}_{i,j},\\ &\mathrm{Tr3} \in \mathscr{B}_{i,a} \times \{a\} \times \mathscr{B}_{a,b} \times \{b\} \times \mathscr{B}_{b,j} \to \mathscr{B}_{i,j},\\ &\mathrm{Put} \in \mathscr{B}_{i,a} \times \{a\} \times \mathscr{B}_{a,j} \to \mathscr{E}_{i,j},\\ &\mathrm{Taken} \in \mathscr{B}_{i,j} \to \mathscr{E}_{i,j}.\end{aligned}$$

It is tempting to assert that $\mathrm{remove23} \in \mathscr{Z} \to \mathscr{O} \to \mathscr{O}$ and $\mathrm{insert23} \in \mathscr{Z} \to \mathscr{O} \to \mathscr{O}$. However, this turns out *not* to be true! The problem is that some of the intermediate functions will only work properly on balanced trees, so this needs to be taken into account when trying to establish that they also preserve order. The goal should be to show that

$$\begin{aligned}&\mathrm{remove23} \in \mathscr{Z} \to \mathscr{N}\mathscr{O} \to \mathscr{N}\mathscr{O},\\ &\mathrm{insert23} \in \mathscr{Z} \to \mathscr{N}\mathscr{O} \to \mathscr{N}\mathscr{O}.\end{aligned}$$

Combining the balance information with the properties of Tr2, Tr3, Put and Taken given above (using (Subset) and (D.1) of Appendix D gives (for any $i,a,b,j,k \in \mathscr{Z}$ where $k \geq 0$):

$$\begin{aligned}&\mathrm{Tr2} \in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{N}_{k+1}\mathscr{E}_{i,j},\\ &\mathrm{Tr3} \in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,b} \times \{b\} \times \mathscr{N}_k\mathscr{E}_{b,j} \to \mathscr{N}_{k+1}\mathscr{E}_{i,j},\\ &\mathrm{Put} \in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{P}_k\mathscr{E}_{i,j},\\ &\mathrm{Taken} \in \mathscr{N}_k\mathscr{E}_{i,j} \to \mathscr{T}_k\mathscr{E}_{i,j}.\end{aligned}$$

These properties can then be used to establish the properties of auxiliary functions. The properties listed in Appendix F can be proved analogously to the way properties of balance are proved and the goal stated above follows easily from these. (The number of cases in this proof is the same as for balance but each case involves more subcases so the size of proof is approximately double that for balance.)

4.4. Proof of abstract properties

The last part of the proof is to show that for all $m,n \in \mathscr{Z}$ and *appropriate* t:

$$\begin{aligned}&\mathrm{member23\ n\ (insert23\ n\ t)} &&= \mathrm{true}\\ &\mathrm{member23\ n\ (insert23\ m\ t)} &&= \mathrm{member23\ n\ t}, \quad \text{if } n \neq m\\ &\mathrm{member23\ n\ (remove23\ n\ t)} &&= \mathrm{false}\\ &\mathrm{member23\ n\ (remove23\ m\ t)} &&= \mathrm{member23\ n\ t}, \quad \text{if } n \neq m\\ &\mathrm{member23\ n\ E} &&= \mathrm{false}\end{aligned}$$

By *appropriate* we mean that t is ordered and also normalised. (The latter being necessary as before to ensure that merging and other auxiliary functions work properly.) We can simply formalise this as the invariant property

$$t \in \mathcal{NO}.$$

It should be noted that the naive approach of trying to prove these equations directly by structural induction on trees soon comes unstuck because auxiliary functions act on the result of inserts and removes as they pass down a tree and properties of the results (beyond their top level structure) need to be known. This requires proving many auxiliary properties as in the proof of the invariant. If we had chosen to prove correctness via an abstraction function [5] from normalised, balanced trees to sets (instead of directly showing overall behaviour is correct), the same problems are encountered. The invariant property and abstraction function do not provide sufficient information about the auxiliary functions for straightforward inductive proofs. Thus we continue with the use of subtype sets to express the properties needed to establish correctness.

We introduce some more subtypes ($\mathcal{C}_n$ and $\mathcal{D}_n$) for each $n \in \mathcal{Z}$ to denote trees which (roughly speaking) *contain* (respectively *do not contain*) the value n. More precisely, we define these sets as trees which produce true (respectively false) when an appropriate membership test is applied to them (they need not be ordered or balanced). These sets can be described as:

$$\mathcal{C}_n = \text{InverseImage}(\text{member23 } n, \{\text{true}\}),$$
$$\mathcal{D}_n = \text{InverseImage}(\text{member23 } n, \{\text{false}\}),$$

and the main properties we need to establish are (for every $n,m \in \mathcal{Z}$):

$$\text{remove23} \in \{n\} \to \mathcal{NO} \to \mathcal{D}_n,$$
$$\text{remove23} \in \{n\} \to \mathcal{C}_m.\mathcal{NO} \to \mathcal{C}_m \quad \text{provided } n \neq m,$$
$$\text{remove23} \in \{n\} \to \mathcal{D}_m.\mathcal{NO} \to \mathcal{D}_m,$$

$$\text{insert23} \in \{n\} \to \mathcal{NO} \to \mathcal{C}_n,$$
$$\text{insert23} \in \{n\} \to \mathcal{C}_m.\mathcal{NO} \to \mathcal{C}_m,$$
$$\text{insert23} \in \{n\} \to \mathcal{D}_m.\mathcal{NO} \to \mathcal{D}_m \quad \text{provided } n \neq m.$$

These can be established from similar results replacing $\mathcal{NO}$ by $\mathcal{N}_k\mathcal{E}_{i,j}$ and (a large number of) further properties of auxiliary functions. In turn these properties allow an easy proof of the equations

```
memberset n emptyset          = false
memberset n (insertset n s)   = true
memberset n (insertset m s)   = memberset n s   (when n ≠ m)
memberset n (removeset n s)   = false
memberset n (removeset m s)   = memberset n s   (when n ≠ m)
```

There are, however, three times as many properties to show for this part of the proof and detailed proofs considering all the cases for these were not carried out in full.

5. Discussion and conclusions

We discuss firstly the equational algorithm for removal (and variations), then the proof of correctness, and finally the proof method and notation more generally.

5.1. Variations on the algorithm

As was pointed out in Section 3.3, several variations on the definitions of functions used in removing and inserting could be used. The merge version discussed in this report was only arrived at after several alternatives had been explored and a "pulling-up" version was subsequently discovered to be just a small variation. None of the other versions found was as simple as the ones described here.

In the literature (e.g. Sedgewick [10]), there is an "improved" (procedural) version of the insertion algorithm which uses 4-nodes as well. The essential idea is to allow a 4-node to replace a Put, but to scan ahead when descending to ensure that such 4-nodes are gradually bubbled up, away from new inserts and thus avoiding any occurrence of a 4-node immediately below a 4-node. This then allows the tree structure to be determined on the way down (rather than after rewrites going back up) at the expense of a complicated look-ahead. This alteration is useful for procedural programming where trees are altered in-place by the operations. In the equational version this would simply replace some tr2 and tr3 rewriting in one step to a passive node by a direct passive node construction with a Tr2 or Tr3 (or Tr4) which is also a one-step construction. Thus there is no saving and considerable overhead in a much more complex pattern match. However, the 4-node version can allow for subsequent inserts to be started before previous ones are complete and so may perform better with parallel rewriting. A further optimisation replaces 3- and 4-nodes by 2-nodes with different colourings for the arcs (pointers) from a node to the subtrees. This change only seems appropriate for procedural programs where pattern matching has to be implemented as case analysis with pointer chasing by the programmer. In this case, a complex case analysis is "simplified" by using fewer cases for the nodes and introducing another level of cases for the colour of arcs. In functional languages and equational programming systems such as O'Donnell's, pattern matching is implemented automatically and can produce very efficient code (see Cardelli [2], O'Donnell [8, Section 18.2], Augustsson [1] and Wadler [12]).

5.2. Proof of the algorithm

Although an algorithm to insert values into balanced trees has been given elsewhere, a full proof of correctness has not been given to our knowledge. Perhaps this is not surprising because the insertion algorithm is fairly "obviously" correct by inspection. For our removal algorithm, however, it would be stretching the imagination a bit too far to claim it is "obviously" correct (although the diagrams help in showing this informally). The use of subtype sets we have introduced to solve this problem also allows us to prove the correctness of the original insertion algorithm.

The essence of the method is that it provides a way to express and reason about the proliferation of auxiliary properties which are needed to establish the invariant and correct abstract behaviour. It seems unfortunate that we could not completely separate our proof of balance preservation from proof of order preservation and proof of the abstract properties of the trees. We might have considered extending the definitions of merge and other functions, so that they worked for unbalanced trees as well. Sometimes a proof can suggest a cleaner algorithm, and changing the algorithm to simplify the proof might well be a good idea in some circumstances. However, the removal algorithm will become more complex with such extensions which are allowing for arguments which are inappropriate and which we should not have to deal with. Such changes seem inappropriate in these circumstances.

An important point concerning proofs and rigor seems worth making here. Earlier versions of the algorithm turned out to be incorrect despite "informal proofs". The errors were discovered during testing of the algorithm, and these led to the discovery of an erroneous case in the proof which had been "checked". Without machine assistance, systematic proofs with a large number of very similar cases lead all too easily to such human errors. Yet the full encoding of the proof seems to be a large task making the difference in effort between fully formal and just rigorous proofs quite noticeable. Mechanical support for such reasoning and the potential for partial automation of similar proofs are topics currently being investigated by the author.

5.3. Subtype set proofs

The proof method is convenient for reasoning about many properties of functions in equational and functional programs. In particular, the sets of values we consider can either include or exclude partial values (such as $\perp$ and Tr2($\perp$,3,$\perp$)) and even infinite values (such as t where t = Tr2(t,3,t)). Constructions of sets such as $\mathcal{N}$ explicitly excluded partial values and infinite values, but these can be used to express other properties. For example $f \in \{\perp\} \to \{\perp\}$ expresses the strictness of f and $f \in X \to Y$ where $\perp \notin Y$ expresses the totalness of f on the set X. As another example, suppose take is defined by:

```
take 0 x = []
take (Succ n) (a::x) = a::take n x
```

where :: and [] are the list constructors and 0 and Succ are natural number constructors. Then for $k \geqslant n \geqslant 0$, T any type, S any non-empty subtype set of T,

$$\begin{aligned}
&\mathsf{take} \in \{n\} \to \mathsf{S}\ \mathsf{list}_k \to \mathsf{S}\ \mathsf{list}_n,\\
&\mathsf{take} \in \{n\} \to \mathsf{S}\ \mathsf{Plist}_k \to \mathsf{S}\ \mathsf{list}_n,\\
&\mathsf{take} \in \{k\} \to \mathsf{S}\ \mathsf{list}_n \to \mathsf{S}\ \mathsf{Plist}_n \quad (k \neq 0),\\
&\mathsf{take} \in \{n\} \to \mathsf{S}\ \mathsf{Inflist} \to \mathsf{S}\ \mathsf{list}_n,
\end{aligned}$$

where S list_k is the set of lists of length k with items in S (defined in a similar way to $\mathcal{N}_k$); S Plist_k is the set of partial lists with k items (each in S)—i.e. terminating in $\perp$ instead of [] after the kth item; and S Inflist is the set of infinite lists with items in S.

There are several dangers in casual use of arbitrary sets, such as unsound reasoning with the empty set, failure of monotonicity for types involving function spaces, interaction with polymorphism, order of pattern matching with lazy semantics. These issues show the need for a fully formalised logic/type system to underpin this kind of reasoning and to investigate the potential for mechanical assistance. Indeed, such a formal system might be based on type systems such as that of Martin-Löf (see e.g. Nordström et al. [7]), but there are some differences. We are interested in establishing relatively simple properties of possibly non-terminating functional/equational programs whereas only terminating programs can be constructed within Martin-Löf's Type theory. Other type theories with both subtypes and recursion such as those investigated by Cardelli [3, 4] have been aimed at explaining programming language features such as class abstraction and inheritance rather than program correctness. These relationships to type systems are still being investigated and the author would like to thank Hong Zhu for discussions concerning this.

Appendix A. Equations for insert23, put, tr2, tr3 and checktop

```
insert23 n t  = checktop(put n t)

put n E = Put(E,n,E)
put n (Tr2(t1,a,t2))
    = if n=a  then  Tr2(t1,a,t2)           else
      if n<a  then  tr2(put n t1,a,t2)     else
      (* n>a *)     tr2(t1,a,put n t2)
put n (Tr3(t1,a,t2,b,t3))
     =if n=a  then  Tr3(t1,a,t2,b,t3)            else
      if n=b  then  Tr3(t1,a,t2,b,t3)            else
      if n<a  then  tr3(put n t1,a,t2,b,t3)      else
      if n<b  then  tr3(t1,a,put n t2,b,t3)      else
      (*n>b*)       tr3(t1,a,t2,b,put n t3)
put n other = error "put of un-normalised tree"
```

```
tr2(Put(t1,a,t2),b,t3) = Tr3(t1,a,t2,b,t3)
tr2(t1,a,Put(t2,b,t3)) = Tr3(t1,a,t2,b,t3)
tr2 other              = Tr2 other

tr3(Put(t1,a,t2),b,t3,c,t4) = Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3(t1,a,Put(t2,b,t3),c,t4) = Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3(t1,a,t2,b,Put(t3,c,t4)) = Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3 other                   = Tr3 other

checktop (Put(t1,a,t2)) = Tr2(t1,a,t2)
checktop other          = other
```

Appendix B. Extended equations for tr2, tr3 and checktop

```
tr2(Put(t1,a,t2),b,t3)             = Tr3(t1,a,t2,b,t3)
tr2(t1,a,Put(t2,b,t3))             = Tr3(t1,a,t2,b,t3)
tr2(Taken t1,a,E)                  = error "tr2(Taken_,_,E)"
tr2(Taken t1,a,Tr2(t2,b,t3))       = Taken(Tr3(t1,a,t2,b,t3))
tr2(Taken t1,a,Tr3(t2,b,t3,c,t4))  = Tr2(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr2(E,a,Taken t1)                  = error "tr2(E,_,Taken_)"
tr2(Tr2(t1,a,t2),b,Taken t3)       = Taken(Tr3(t1,a,t2,b,t3))
tr2(Tr3(t1,a,t2,b,t3),c,Taken t4)  = Tr2(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr2 other                          = Tr2 other

tr3(Put(t1,a,t2),b,t3,c,t4)              = Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3(t1,a,Put(t2,b,t3),c,t4)              = Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3(t1,a,t2,b,Put(t3,c,t4))              = Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3(Taken t1,a,Taken t2,b,Taken t3)      = Taken(Tr3(t1,a,t2,b,t3))
tr3(Taken t1,a,Taken t2,b,t3)            = Tr2(Tr2(t1,a,t2),b,t3)
tr3(t1,a,Taken t2,b,Taken t3)            = Tr2(t1,a,Tr2(t2,b,t3))
tr3(Taken t1,a,Tr2(t2,b,t3),c,Taken t4)  = Tr2(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
tr3(Taken t1,a,Tr3(t2,b,t3,c,t4),d,Taken t5)
                                         = Tr2(Tr3(t1,a,t2,b,t3),c,Tr2(t4,d,t5))
tr3(Taken t1,a,E,b,t2)                   = error "tr3(Taken_,_,E,_,_)"
tr3(Taken t1,a,Tr2(t2,b,t3),c,t4)        = Tr2(Tr3(t1,a,t2,b,t3),c,t4)
tr3(Taken t1,a,Tr3(t2,b,t3,c,t4),d,t5)   = Tr3(Tr2(t1,a,t2),b,Tr2(t3,c,t4),d,t5)
tr3(t1,a,E,b,Taken t2)                   = error "tr3(_,_,E,_,Taken_)"
tr3(t1,a,Tr2(tr2,b,t3),c,Taken t4)       = Tr2(t1,a,Tr3(t2,b,t3,c,t4))
tr3(t1,a,Tr3(t2,b,t3,c,t4),d,Taken t5)   = Tr3(t1,a,Tr2(t2,b,t3),c,Tr2(t4,d,t5))
tr3(E,a,Taken t1,b,t2)                   = error "tr3(E,_,Taken_,_,_)"
tr3(Tr2(t1,a,t2),b,Taken t3,c,t4)        = Tr2(Tr3(t1,a,t2,b,t3),c,t4)
tr3(Tr3(t1,a,t2,b,t3),c,Taken t4,d,t5)   = Tr3(Tr2(t1,a,t2),b,Tr2(t3,c,t4),d,t5)
tr3 other                                = Tr3 other
```

```
checktop (Put(t1,a,t2))=Tr2(t1,a,t2)
checktop (Taken t)     =t
checktop other         =other
```

Appendix C. Equations for remove23, remove, merge, leftPut and rightPut

```
remove23 n t =checktop(remove n t)

remove n E =E
remove n (Tr2(t1,a,t2))
        = if n=a then  merge(t1,t2)                else
          if n<a then  tr2(remove n t1,a,t2)       else
          (* n>a *)    tr2(t1,a,remove n t2)
remove n (Tr3(t1,a,t2,b,t3))
        = if n=a then  leftPut(merge(t1,t2),b,t3)     else
          if n=b then  rightPut(t1,a,merge(t2,t3))    else
          if n<a then  tr3(remove n t1,a,t2,b,t3)     else
          if n<b then  tr3(t1,a,remove n t2,b,t3)     else
          (* n>b *)    tr3(t1,a,t2,b,remove n t3)
remove n other=error "remove of un-normalised tree"

merge(E,E)   =Taken E
merge(Tr2(t1,a,t2),Tr2(t3,b,t4))
              =tr3(Taken t1,a,merge(t2,t3),b,Taken t4)
merge(Tr2(t1,a,t2),Tr3(t3,b,t4,c,t5))
              =tr3(Taken t1,a,merge(t2,t3),b,Tr2(t4,c,t5))
merge(Tr3(t1,a,t2,b,t3),Tr2(t4,c,t5))
              =tr3(Tr2(t1,a,t2),b,merge(t3,t4),c,Taken t5)
merge(Tr3(t1,a,t2,b,t3),Tr3(t4,c,t5,d,t6))
              =tr3(Tr2(t1,a,t2),b,merge(t3,t4),c,Tr2(t5,d,t6))
merge other  =error "merge of inappropriate trees"

leftPut(Tr2(t1,a,t2),b,t3)       =Tr3(t1,a,t2,b,t3)
leftPut(Tr3(t1,a,t2,b,t3),c,t4)  =Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
leftPut(Taken t1,a,t2)           =Tr2(t1,a,t2)
leftPut(E,a,t1)                  =error "leftPut(E,_,_)"
leftPut(Put t,a,t1)              =error "leftPut(_,_,_)"

rightPut(t1,a,Tr2(t2,b,t3))      =Tr3(t1,a,t2,b,t3)
rightPut(t1,a,Tr3(t2,b,t3,c,t4))=Put(Tr2(t1,a,t2),b,Tr2(t3,c,t4))
rightPut(t1,a,Taken t2)          =Tr2(t1,a,t2)
rightPut(t1,a,E)                 =error "rightPut(_,_,E)"
rightPut(t1,a,Put t)             =error "rightPut(_,_,Put_)"
```

Appendix D. Some properties of subtype sets

$$\frac{s \in S \quad S \subseteq S'}{s \in S'} \qquad \text{(Subset)}$$

$$\frac{\forall i \in I\,(s \in S_i)}{s \in \bigcap_{i \in I} S_i} \quad \frac{s \in \bigcap_{i \in I} S_i}{\forall i \in I\,(s \in S_i)} \qquad \text{(Intersection)}$$

$$\frac{f \in X \to Y}{\forall x \in X (f\, x \in Y)} \quad \frac{\forall x \in X (f\, x \in Y)}{f \in X \to Y} \qquad \text{(Application)}$$

$$\bigcap_{i \in I} (X_i \to Y_i) \subseteq \bigcap_{i \in I} X_i \to \bigcap_{i \in I} Y_i \qquad \text{(D.1)}$$

$$\bigcap_{i \in I} (X \to Y_i) = X \to \bigcap_{i \in I} Y_i \qquad \text{(D.2)}$$

$$\bigcap_{i \in I} (X_i \to Y_i) \subseteq \bigcup_{i \in I} X_i \to \bigcup_{i \in I} Y_i \qquad \text{(D.3)}$$

$$\bigcap_{i \in I} (X_i \to Y) = \bigcup_{i \in I} X_i \to Y \qquad \text{(D.4)}$$

$$(X_1 \times \cdots \times X_n)(Y_1 \times \cdots \times Y_n) = (X_1 Y_1) \times \cdots \times (X_n Y_n) \qquad \text{(D.5)}$$

$$\frac{X' \subseteq X \quad Y \subseteq Y'}{X \to Y \subseteq X' \to Y'} \qquad \text{(D.6)}$$

$$\frac{\forall i \in I\,(t \in X_i \to Y)}{t \in \bigcup_{i \in I} X_i \to Y} \qquad \text{(D.7)}$$

$$\frac{\forall i \in I\,(t \in X \to Y_i \to Z_{f(i)}) \quad \forall i \in I\,(f(i) \in J)}{t \in X \to \bigcup_{i \in I} Y_i \to \bigcup_{j \in J} Z_j} \qquad \text{(D.8)}$$

$$\frac{I \subseteq J}{\bigcup_{i \in I} Z_i \subseteq \bigcup_{j \in J} Z_j} \qquad \text{(D.9)}$$

Appendix E. Balance properties of functions

remove23 $\in \mathscr{X} \to \mathscr{N} \to \mathscr{N}$
insert23 $\in \mathscr{X} \to \mathscr{N} \to \mathscr{N}$

(for $k \geqslant 0$)

remove23 $\in \mathscr{X} \to \mathscr{N}_k \to (\mathscr{N}_k \cup \mathscr{N}_{k-1} \cup \mathscr{N}_{k+1})$
insert23 $\in \mathscr{X} \to \mathscr{N}_k \to (\mathscr{N}_k \cup \mathscr{N}_{k+1})$

tr2 $\in \mathscr{P}_k \times \mathscr{X} \times \mathscr{N}_k \to \mathscr{N}_{k+1}$
tr2 $\in \mathscr{N}_k \times \mathscr{X} \times \mathscr{P}_k \to \mathscr{N}_{k+1}$
tr2 $\in \mathscr{T}_k \times \mathscr{X} \times \mathscr{N}_k \to \mathscr{T}_{k+1}$
tr2 $\in \mathscr{N}_k \times \mathscr{X} \times \mathscr{T}_k \to \mathscr{T}_{k+1}$

tr3 $\in \mathscr{P}_k \times \mathscr{Z} \times \mathscr{N}_k \times \mathscr{Z} \times \mathscr{N}_k \to \mathscr{P}_{k+1}$
tr3 $\in \mathscr{N}_k \times \mathscr{Z} \times \mathscr{P}_k \times \mathscr{Z} \times \mathscr{N}_k \to \mathscr{P}_{k+1}$
tr3 $\in \mathscr{N}_k \times \mathscr{Z} \times \mathscr{N}_k \times \mathscr{Z} \times \mathscr{P}_k \to \mathscr{P}_{k+1}$
tr3 $\in \mathscr{T}_k \times \mathscr{Z} \times \mathscr{T}_k \times \mathscr{Z} \times \mathscr{T}_k \to \mathscr{T}_{k+1}$

Tr2 $\in \mathscr{N}_k \times \mathscr{Z} \times \mathscr{N}_k \to \mathscr{N}_{k+1}$
Tr3 $\in \mathscr{N}_k \times \mathscr{Z} \times \mathscr{N}_k \times \mathscr{Z} \times \mathscr{N}_k \to \mathscr{N}_{k+1}$
Put $\in \mathscr{N}_k \times \mathscr{Z} \times \mathscr{N}_k \to \mathscr{P}_k$
Taken $\in \mathscr{N}_k \to \mathscr{T}_{k+1}$

merge $\in \mathscr{N}_k \times \mathscr{N}_k \to \mathscr{T}_{k+1}$

leftPut $\in \mathscr{T}_{k+1} \times \mathscr{Z} \times \mathscr{N}_k \to \mathscr{P}_{k+1}$

rightPut $\in \mathscr{N}_k \times \mathscr{Z} \times \mathscr{T}_{k+1} \to \mathscr{P}_{k+1}$

checktop $\in \mathscr{T}_k \to (\mathscr{N}_k \cup \mathscr{N}_{k-1})$
checktop $\in \mathscr{P}_k \to (\mathscr{N}_k \cup \mathscr{N}_{k+1})$

put $\in \mathscr{Z} \to \mathscr{N}_k \to \mathscr{P}_k$

remove $\in \mathscr{Z} \to \mathscr{N}_k \to (\mathscr{T}_k \cup \mathscr{P}_k)$

Appendix F. Order properties of functions

remove23 $\in \mathscr{Z} \to \mathscr{N}\mathscr{O} \to \mathscr{O}$
insert23 $\in \mathscr{Z} \to \mathscr{N}\mathscr{O} \to \mathscr{O}$

(For every k,i,a,b,j,n $\in \mathscr{Z}$ with $k \geq 0$)

remove23 $\in \mathscr{Z} \to \mathscr{N}_k\mathscr{E}_{i,j} \to \mathscr{E}_{i,j}$
insert23 $\in \{n\} \to \mathscr{N}_k\mathscr{E}_{i,j} \to \mathscr{E}_{r,s}$
where $r = \min(n-1, i)$ and $s = \max(j, n+1)$

tr2 $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{P}_k\mathscr{E}_{a,j} \to \mathscr{N}_{k+1}\mathscr{E}_{i,j}$
tr2 $\in \mathscr{P}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{N}_{k+1}\mathscr{E}_{i,j}$
tr2 $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{T}_k\mathscr{E}_{a,j} \to \mathscr{T}_{k+1}\mathscr{E}_{i,j}$
tr2 $\in \mathscr{T}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{T}_{k+1}\mathscr{E}_{i,j}$

tr3 $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,b} \times \{b\} \times \mathscr{P}_k\mathscr{E}_{b,j} \to \mathscr{P}_{k+1}\mathscr{E}_{i,j}$
tr3 $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{P}_k\mathscr{E}_{a,b} \times \{b\} \times \mathscr{N}_k\mathscr{E}_{b,j} \to \mathscr{P}_{k+1}\mathscr{E}_{i,j}$
tr3 $\in \mathscr{P}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,b} \times \{b\} \times \mathscr{N}_k\mathscr{E}_{b,j} \to \mathscr{P}_{k+1}\mathscr{E}_{i,j}$
tr3 $\in \mathscr{T}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{T}_k\mathscr{E}_{a,b} \times \{b\} \times \mathscr{T}_k\mathscr{E}_{b,j} \to \mathscr{T}_{k+1}\mathscr{E}_{i,j}$

Tr2 $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{N}_{k+1}\mathscr{E}_{i,j}$
Tr3 $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,b} \times \{b\} \times \mathscr{N}_k\mathscr{E}_{b,j} \to \mathscr{N}_{k+1}\mathscr{E}_{i,j}$
Put $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \{a\} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{P}_{k+1}\mathscr{E}_{i,j}$
Taken $\in \mathscr{N}_k\mathscr{E}_{i,j} \to \mathscr{T}_{k+1}\mathscr{E}_{i,j}$

merge $\in \mathscr{N}_k\mathscr{E}_{i,a} \times \mathscr{N}_k\mathscr{E}_{a,j} \to \mathscr{T}_{k+1}\mathscr{E}_{i,j}$

$$\text{leftPut} \in \mathcal{T}_{k+1}\mathcal{E}_{i,a} \times \{a\} \times \mathcal{N}_k\mathcal{E}_{a,j} \to \mathcal{P}_{k+1}\mathcal{E}_{i,j}$$

$$\text{rightPut} \in \mathcal{N}_k\mathcal{E}_{i,a} \times \{a\} \times \mathcal{T}_{k+1}\mathcal{E}_{a,j} \to \mathcal{P}_{k+1}\mathcal{E}_{i,j}$$

$$\text{remove} \in \mathcal{X} \to \mathcal{N}_k\mathcal{E}_{i,j} \to (\mathcal{T}_k \cup \mathcal{P}_k)\mathcal{E}_{i,j}$$

$$\text{checktop} \in \mathcal{E}_{i,j} \to \mathcal{E}_{i,j}$$

$$\text{put} \in \{n\} \to \mathcal{N}_k\mathcal{E}_{i,j} \to \mathcal{P}_k\mathcal{E}_{r,s}$$
$$\text{where } r = \min(n-1, i) \text{ and } s = \max(j, n+1)$$

References

[1] L. Augustsson, Compiling pattern-matching, in: Jouannaud, ed., *Conference on Programming Languages and Computer Architecture,* Nancy, France, Lecture Notes in Computer Science 201 (Springer, Berlin, 1985) 368-381.

[2] L. Cardelli, Compiling a functional language, in: *Proceedings ACM Symposium on Lisp and Functional Programming,* Austin, TX (1984).

[3] L. Cardelli and G. Longo, A semantic basis for quest, Research Report No. 55, DEC Systems Research Center (1990).

[4] L. Cardelli and P. Wegner, On understanding types, data abstraction and polymorphism, *Comput. Surv.* **17** (4) (1985) 471-522.

[5] C.A.R. Hoare, Proof of correctness of data representations, *Acta Inform.* **1** (1972) 271-281.

[6] C.M. Hoffman and M.J. O'Donnell, Programming with equations, *ACM Trans. Programming Languages Syst.* **4** (6) (1982) 83-112.

[7] B. Nordström, K. Petersson and J.M. Smith, *Programming in Martin-Löfs Type Theory,* International Series of Monographs in Computer Science (7) (Clarendon Press, Oxford, England, 1990).

[8] M.J. O'Donnell, *Equational Logic as a Programming Language* (MIT Press, Cambridge, MA, 1985).

[9] C.M.P. Reade, Balanced trees with removals: an exercise in rewriting and proof, Tech. Report CSTR-91-4, Department of Computer Science, Brunel University, Uxbridge (1991).

[10] R. Sedgewick, *Algorithms* (Addison-Wesley, Reading, MA, 1983).

[11] S. Thatte, A refinement of strong sequentiality for term rewriting with constructors, *Inform. Comput.* **72** (1) (1987) 46-55.

[12] P. Wadler, Efficient compilation of pattern-matching, in: S. Peyton-Jones, ed., *The implementation of Functional Programming Languages,* Prentice-Hall Series in Computer Science (Prentice-Hall, Englewood Cliffs, NJ, 1987) Chapter 5.