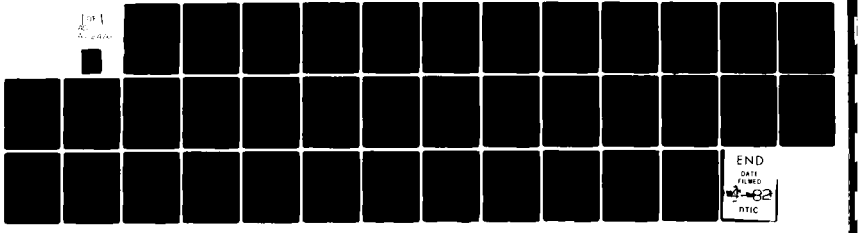
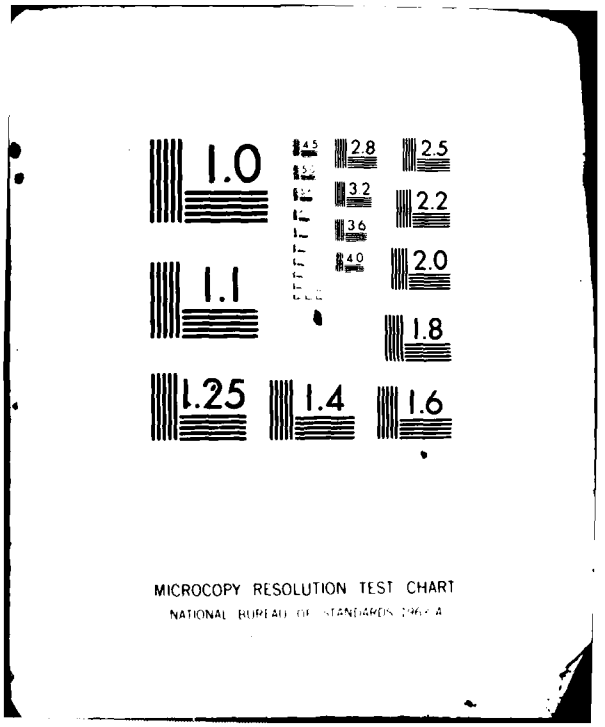
AD-A112 492 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
ON PROGRAM TRANSFORMATIONS FOR ABSTRACT DATA TYPES AND CONCURRE--ETC(U)
OCT 81 P PEPPER N00014-76-C-0607
UNCLASSIFIED STAN-CS-81-883 NL



END
DATE FILMED
4-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

October 1981

Report. No. STAN-CS-81-883

On Program Transformations for Abstract Data Types and Concurrency

by

P. Pepper

Department of Computer Science

Stanford University
Stanford, CA 94305

On
Program Transformations
for
Abstract Data Types
and Concurrency

by

P. Pepper

Computer Science Department
Stanford University
Stanford, Ca 94305

Abstract

We study transformation rules for a particular class of abstract data types, namely types that are representable by recursive mode declarations. The transformations are tailored to the development of efficient tree traversal and they allow for concurrency. The techniques are exemplified by an implementation of concurrent insertion and deletion in 2-3-trees.

This research has been supported by the *Deutsche Forschungsgemeinschaft* and by the Office of Naval Research under Contract N00014-76-C-0687.

1. Data Types, Transformatios and Concurrency

The purpose of this study is threefold: We seek transformations for data structures. We want these transformations to allow for concurrency. And we want to apply them to a nontrivial example, viz. concurrent operations on data bases. We do *not* talk here about notations for transformations nor do we discuss their automatic application. And we avoid detailed semantical considerations (which are replaced by references to the pertinent literature).

Program transformations shall capture frequently employed programming techniques — be they newly invented or taken from the literature — into a schematic form that allows a fairly easy application to concrete problems. It usually does not make sense to aspire to rules that are applicable to any program[1]. One rather has to concentrate on special classes of programs to achieve reasonably powerful and flexible transformations (e.g. recursion removal, loop optimization, etc.). For this reason we focus on the particular class of those data structures which are defined by recursive mode declarations. The transformations presented include the derivation of traversal operations and of pointer implementations.

Concurrency is viewed here as an optimization issue rather than as an inherent property of the given problem. This view is certainly admissible when the concurrency comes into existence by allowing several processes to execute a known sequential task simultaneously. In this case it is also appropriate to work with the so-called *multiprogramming assumption*, i.e. to semantically model parallelism as sequential interleaving of atomic actions.

Data base operations such as insertion, removal and search meet the above requirements. These operations are well-known in sequential applications and they are now to be transferred into concurrent environments. For the implementation of data bases one frequently uses tree structures, for example the B-trees suggested in [4]. To shorten the writing-down we will restrict ourselves to the special case of 2-3-trees (cf. [1]).

The paper has two major parts. Section 3 lists a number of transformations for recursive data structures. These transformations essentially capture standard programming techniques within the formalism of abstract data types and also make an attempt to cope with concurrency and protection problems. The use of abstract data types leads to a valuable accuracy of the specifications. But one has to pay a price, viz. an increased length of code. (This makes the generation of abstract data types by transformations even more desirable.) Even if we restrict our attention to the class of recursive type declarations there remains a vast abundance of conceivable transformations. The choice made in this paper is primarily motivated by the intended sample development. But, vice versa, the design of the algorithm is also influenced by observations about the available rules — an interesting feedback between transformations and decision making on the programmer's side.

Section 4 contains a development of the operations insert and remove for 2-3-trees, allowing for concurrency. The algorithm differs somewhat from the one given in [4] and its improved version in [11]. This difference may be roughly described as follows: B-trees as well as their special case 2-3-trees have two characterisitc properties, viz. the special tree structure, which is responsible for the correctness of all operations, and the balancing (all paths have the same length), which is responsible for the efficiency. The operations insert and remove at least temporarily violate these properties. In [4] the decision was to violate the particular tree structure and to keep the balancing intact. We feel, however, that the correctness issue deserves precedence and therefore keep the

[1] Well, compilers are an exception.

tree structure intact while tolerating a temporary disturbance of the balancing. As a consequence, locks are only needed for a very few nodes at a time and reading as well as writing operations may almost unrestrictedly coexist in the same (sub)trees.

There is one severe problem: Transformations ought to be "correct". But any notion of correctness is vain without a precise semantical definition of the language under consideration. Our topics here are abstract data types and concurrency, both issues that are currently attacked (with varying success) in an abundance of articles. It clearly is far beyond the scope of this paper to discuss two such heavy problems incidentally. We will therefore nolens volens rely on a more intuitive idea of the correctness of transformations. The focus of our attention will be the technical feasibility and the usability of the rules, assuming that they are semantically justified. (It will only be in the appendix that we try to at least sketch the underlying semantic modelling.)

2. The Basic Problem

The problem domain we have chosen allows a clear partitioning into two subtasks: The operations to be performed on the given data structure are specified completely independently from any concurrency considerations. The parallelism comes in by simply allowing several sequential processes to operate on the common data structure simultaneously. The situation is therefore characterized by the fact that the the processes communicate for no other reason but protection against mutual interferences.

2.1. The Data Type DICTIONARY

In a very abstract form, data bases may be thought of as sets on which the basic operations "insert an element", "find an element" and "remove an element" can be performed. These operations are formally specified by the following abstract data type DICTIONARY (as it is termed in [1]). The notation will be explained below.

type DICTIONARY (**type** ELEM) **declares** set,insert,remove,member :

based on: ELEM, BOOL

sorts: set

opns:

empty :	elem	→	set	
insert :	set × elem	→	set	
remove :	set × elem	→	set	
member :	set × elem	→	bool	

axioms: ∀ set s, elem x, y :

member(empty, y)	= false	
member(insert(s, x), y)	= true	if x = y
	= member(s, y)	otherwise
remove(insert(s, x), y)	= remove(s, y)	if x = y
	= insert(remove(s, y), x)	otherwise

required: ∀ elem x :

s ≠ empty **for** remove(s, x)

end of type

The notation used here is a syntactically sugared mixture of those found in [7] and [13]: The parametrization takes into account that sets can be defined for any type of elements. As suggested by the keyword **declares**[2] a number of identifiers are provided to the environment; all functions not listed here are "hidden". (Thus the operation **empty** is hidden. Actually, without this operation it is impossible to generate an initial set for further use. But we assume that there is some mechanism such as the ADA package initialization, which is of no concern to us here. We deal only with processes that are allowed to use the remaining operations on an existing set.) The type is based on two other types, viz. the parameter ELEM and BOOL, and it introduces a new sort of objects, called **sets**. The list of operation symbols together with their functionalities completes the "syntactic interface".

The axioms, a set of universally quantified conditional equations, specify the behavior of the operations and objects defined by the type. The requirements play a particular role: It is understood that all other equations only hold if none of the requirements is violated ("implicit condition"). In the literature one finds essentially two ways of handling partially defined functions such as **remove**. One is to introduce special **error**-elements. In this setting we would write

$$\forall \textbf{set}\ s,\ \textbf{elem}\ x : \mathbf{remove}(s,x) = \mathbf{error} \quad \textbf{if} \quad s = \mathbf{empty}.$$

Alternatively, one may work with partial algebras, which leads to notations such as

$$\forall \textbf{set}\ s,\ \textbf{elem}\ x : \mathbf{undefined}(\mathbf{remove}(s,x)) \quad \textbf{if} \quad s = \mathbf{empty},$$

or equivalently

$$\forall \textbf{set}\ s,\ \textbf{elem}\ x : s \neq \mathbf{empty} \quad \textbf{if} \quad \mathbf{defined}(\mathbf{remove}(s,x)).$$

For some of the applications in this paper the last form will be the most convenient one for expressing the restriction of the domain of a function. To stress this particular usage we employ the special notation[2]

$$\forall \textbf{set}\ s,\ \textbf{elem}\ x : s \neq \mathbf{empty} \quad \textbf{for} \quad \mathbf{remove}(s,x).$$

From our practical point of view we need not care about the two (theoretically quite different) possible interpretations of partially defined functions.

In later examples we will use some further constructs. The expression **enrich...by** instead of **based on...** means that we only add new operations to the type, but that these operations do not generate further objects. As pointed out by Burstall and Goguen (cf. [7]) this allows the use of, say, existential quantifiers in enrichments. Finally, the expression $A \oplus B$ builds the "disjoint union" of the types A and B except that common subtypes are not duplicated.

According to the principles described in [13] or [5] any model of this data type is acceptable as an implementation. Though never making explicit use of it we will orient our whole development towards the so-called "terminal model". Without going into any theoretical details this model may be roughly characterized by the following definition of the equality of its objects. The only way to distinguish two sets "from the outside" is by checking their elements using the operation **member**. This means that two sets s and s' are "indistinguishable" or "visibly equivalent" if for all x the equality $\mathbf{member}(s,x) = \mathbf{member}(s',x)$ holds. When applying this criterium to tree implementations we have to consider two trees as being equivalent if they have the same leaves (even though they may exhibit completely different internal structures).

---

[2]Specification languages exhibit a tendency towards colloquialism.

2.2. CONCURRENCY

The second major aspect of our task is concurrency: There are several processes performing the above operations on the given set simultaneously. For the protection of these processes against mutual interferences when accessing the common set **s** we employ a locking mechanism. The operation **r-lock(s)** sets a read-lock on the variable **s** and the operation **w-lock(s)** sets a write-lock.

Our program consists of n processes executing set operations of the kind

$$[\![\ \ldots \ \| P_\mu :: \ \ldots \ \texttt{r-lock(s); b := member(s,x); r-unlock(s);} \ \ldots \ \| \ldots \ \| P_\pi :: \ \ldots \ \texttt{w-lock(s); s := insert(s,x); w-unlock(s);} \ \ldots \ \| \ \ldots \ ]\!]$$

In the above specification of the concurrent processes and their interaction the operations **insert**, **remove** and **member** are taken to be "atomic". This is, of course, unrealistic and therefore the goal of the subsequent program development will be to increase the degree of concurrency by implementing these operations in terms of more elementary ones.

3. TECHNIQUES, TOOLS AND TRANSFORMATIONS

The underlying idea of program transformation is to capture repeatedly occurring programming techniques in schematic rules such that it becomes fairly easy to apply these techniques to concrete problems. For classical language constructs (recursive functions, loops, assignments etc.) pairs of program schemata have already proved good (cf. e.g [6]), but in connection with data structures and their operations the task becomes more complex. We will try here to attack the issue within the framework of abstract data types.

Systems such as CLEAR or OBJ (cf. [7], [8]) provide valuable tools for modularizing a development process and for describing relationships between various types. (In addition they are based on a sound mathematical theory.) However, the systems of types are still "static", i.e. fixed upon writing-down. A stepwise program development process also needs a dynamic component, i.e. tools for altering given type systems. Transformations are such a dynamic tool. We are not attempting to cast the transformations presented in this chapter into a formal notation. The paper should rather be seen as a test whether the development of such a formalism seems worthwhile.

We try here to identify those transformations that are needed to achieve a particular goal, viz. tree operations that can do without exhaustive search. Therefore we develop specialized abstract data types that contain just the properties we need. These properties are found by analysing algorithms known from the literature and by separating their essential characterisitics from minor particularities.

3.1. Recursive Type Declarations

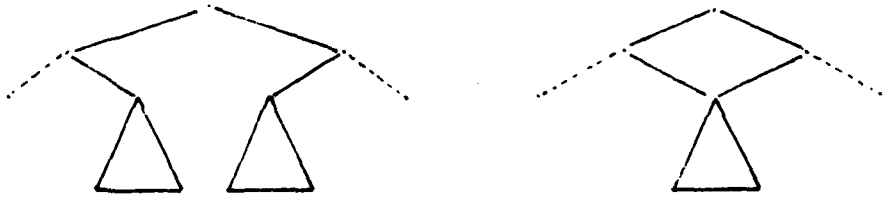
There is a particular class of abstract data types that are both easily understood and frequently occurring in applications. In fact, due to their simplicity these types are usually written in the shorter notation of recursive mode declarations. A classical example are binary trees:

type tree = leaf(elem item) | cons(tree son1, tree son2)

This can be viewed as a shorthand notation for a type TREE that introduces in addition to the operations **leaf**, **item**, **cons**, **son1** and **son2** with their obvious axioms also the test functions **. is leaf** and **. is cons** (cf.[3]).

Because of its shortness we will from now on work with this example, but it should be understood that the techniques apply to any type that is defined by such a recursive declaration involving direct product and direct sum. In this sense the subsequent sections should be seen as presenting rules that apply to arbitrary recursively defined data types.

A note of caution: If a tree contains two identical subtrees the specification allows implementations according to either of the following figures:



So one has to be careful when applying arguments that intuitively refer to the left model. The subsequent considerations work for both cases, but in general it is recommendable to use trees without multiple subtrees.

We are aiming at a particular class of algorithms, namely those that traverse trees efficiently without the need for exhaustive searches such as depth-first or breadth-first search. Therefore we assume that there exists a predicate P that determines for any given tree whether the search should continue in the left or in the right son. (This special case is the one we are interested in and for which we therefore formulate a transformation.) To express this special goal we introduce an enrichment of the type TREE that reflects the property that P is true for exactly one son if it is true for the father.

type PTREE declares P :

enrich TREE by

opns: P : tree → bool

required: ∀ **tree t,** :

$$(P(son1(t)) \vee P(son2(t))) = true \quad \textbf{if} \quad t \text{ is cons} \wedge P(t) = true$$
$$(P(son1(t)) \wedge P(son2(t))) = false \quad \textbf{if} \quad t \text{ is cons} \wedge P(t) = true$$

end of type

The generation of this type could be done automatically by some of the later transformations. But the presentation is more straightforward if we do it right here, for this indicates that the whole development relies on the existence of such a P.

3.2. SUBTREES AND SUBSTITUTION

Basic notions for trees are the predicate "is subtree of" and the operation "within t substitute s by u". For easier readability we will use the infix notations $s \preceq t$ for the subtree relation and $t[s \Leftarrow u]$ for the substitution operation. Both can be defined by a simple data type extension. (Note that substitute replaces all occurrences of s within t and that the restriction for the substitution is induced by the type PTREE.)

type TREE$^+$ **declares** $. \preceq .$, $.[. \Leftarrow .]$:

enrich PTREE **by**

opns: $. \preceq . :$ **tree** $\times$ **tree** $\rightarrow$ **bool**

$.[. \Leftarrow .] :$ **tree** $\times$ **tree** $\times$ **tree** $\rightarrow$ **tree**

axioms: $\forall$ **tree** s, t, u :

$s \preceq t = \text{true}$ **if** $s = t$

$= \text{true}$ **if** t **is cons** $\wedge$ ($s \preceq \text{son1}(t) \vee s \preceq \text{son2}(t)$)

$= \text{false}$ **otherwise**

$t[s \Leftarrow u] = u$ **if** $t = s$

$= \text{cons}(\text{son1}(t)[s \Leftarrow u], \text{son2}(t)[s \Leftarrow u])$ **if** $t \neq s \wedge$ t **is cons**

$= t$ **otherwise**

required: $\forall$ **tree** t, u :

$P(t) \Leftrightarrow P(u)$ **for** $t[t \Leftarrow u]$

end of type

Now assume that we have operations that transform trees into trees. As a running example we will use the addition of a leaf.

type TREE1 **declares** add :

enrich PTREE **by**

opns: add : **tree** $\times$ **elem** $\rightarrow$ **tree**

axioms: $\forall$ **tree** s, **elem** y :

$\text{add}(s, y) = s$ **if** $\text{item}(s) = y$

$= \text{cons}(s, \text{leaf}(y))$ **otherwise**

required: $\forall$ **tree** s, **elem** y :

s *is leaf* **for** $\text{add}(s, y)$

end of type

The leaf to which the operation **add** shall be applied is in general contained in some enclosing tree. Therefore we need an extension **Add** that applies **add** to a suitable subtree of a large given tree. The problem here lies in the word "suitable": Since the restrictions above do not determine an admissible subtree uniquely we have to make a choice. Making choices means to introduce nondeterminism. Within the classical framework of abstract data types the only way of doing this is by way of disjunctions and existential quantifiers. (For more details see appendix Λ.) The following type therefore specifies that the addition should take place at any subtree **s** of **t** which is a leaf. (Note: If **s** occurs several times in **t** the new leaf is added several times; but in our aforementioned equivalence relation this does not matter.)

type EXTENDED TREE **declares** **Add** :

enrich $\text{TREE}^+ \oplus \text{TREE1}$ **by**

opns: **Add** : **tree** × **elem** → **tree**

axioms: $\forall$ **tree t**, **elem y**, $\exists$ **tree s** :

$s \preceq t \wedge s \text{ is leaf} \wedge \text{Add}(t,y) = t[s \Leftarrow \text{add}(s,y)]$

end of type

This type illustrates a typical aspect of transformation systems. Certain rules are designed in a relatively complicated fashion to ensure correctness in any case. But they will only effect improvements when applied to programs exhibiting special properties. In our case this means that the above transformation is best applied in those cases where the restrictions of the type TREE1 determine the subtree uniquely.

Transformation: The transformation has the form MAKE-EXTENDED-TREE(TREE,TREE1), where TREE refers to (some enrichment of) a recursive type and TREE1 specifies the special operations such as **add**. The result is a type of the kind EXTENDED TREE.

The transformation is indeed mechanically performable, since all parts of the equations are syntactically derivable from the given types. In particular, the condition in the type EXTENDED TREE is just the requirement for the operation **add**, i.e. the substitution is applied at some permissible point.

If we make corresponding enrichments for deletion and membership test (see section 4) this type provides an implementation for the type DICTIONARY. To show this one needs an equivalence relation on trees (cf. [5]). The appropriate one here is: Two trees t_1 and t_2 are **equivalent** if they have the same set of leaves. (The internal structure of the trees as well as the multitude of leaves are neglected.) Using this relation we could prove that all axioms of the type DICTIONARY are fulfilled by the implementation. As a matter of fact, the resulting model is the terminal one, since the equivalence coincides with the visible equivalence of DICTIONARY.

This will be the only point in a program development along the lines described here, where one needs proofs on a meta-level (employing equivalence relations, homomorphisms and the like). From then on the development can make use of the transformations to be presented below.

3.3. TREE TRAVERSAL

Though being very convenient for specification purposes — e.g. for the proof that the type EXTENDED TREE is an implementation of the type DICTIONARY — the above notion of "subtree" does not indicate *how* the subtree may be localized in an efficient way. To get a more operational specification that describes "tree traversal" we introduce the idea of a "current node" into our types. (We will use the terms "current node" and "current (sub)tree" synonymously.) Conceptually this may be specified by a pair of two trees one of which is a subtree of the other.

The predicate P introduced in the type PTREE of section 3.1 is now used to direct the search through the tree. For this reason the subsequent type is based on PTREE.

type TRAVERSABLE TREE **declares** ttree,reset,down,up,current :

enrich PTREE **by**

sorts: ttree

opns:
- init : tree → ttree
- reset : ttree → ttree
- down : ttree → ttree
- up : ttree → ttree
- root : ttree → tree
- current : ttree → tree

axioms: ∀ ttree tt, tree s :
- root(init(s)) = s
- root(reset(tt)) = root(tt)
- root(down(tt)) = root(tt)
- root(up(tt)) = root(tt)
- current(reset(tt)) = root(tt)
- current(down(tt)) = son_i(current(tt)) **if** P(son_i(current(tt))) = true
- up(down(tt)) = tt

required: ∀ ttree tt :
- current(tt) $\preceq$ root(tt)
- P(current(tt)) = true
- current(tt) ≠ root(tt) **for** up(tt)
- ¬(current(tt) is leaf) **for** down(tt)

end of type

Note that the operations init and root are hidden, i.e. any process working with the type can only use the operations reset etc. (As in the type DICTIONARY we assume some kind of initialization mechanism.) The operation up is only specified as being the inverse of down. Note also that the first two requirements hold by construction. But they will become important in a later extension.

As a straightforward extension we now add the operation "substitute" in such a way that it is automatically applied to the current subtree.

type TRAVERSABLE TREE$^+$ **declares** subst :

enrich TRAVERSABLE TREE ⊕ TREE$^+$ **by**

opns: subst : ttree × tree → ttree

axioms: ∀ **ttree tt, tree s :**

$\mathsf{root(subst(tt,s))} = \mathsf{root(tt)[current(tt) \Leftarrow s]}$

$\mathsf{current(subst(tt,s))} = \mathsf{s}$

required: ∀ **ttree tt, tree s :**

$\mathsf{P(s)} = \mathsf{true}$ **for** $\mathsf{subst(tt,s)}$

end of type

The requirement is enforced by the type TRAVERSABLE TREE: The predicate P has to be invariantly true for the current subtree.

Transformation: In the transformation MAKE-TRAVERSABLE-TREE(TREE,P) the parameter TREE again refers to (some enrichment of) a recursively defined type and P is the predicate for directing the search. The result of the transformation is the type TRAVERSABLE TREE$^+$ (including the type PTREE, which may be automatically generated here).

Note that the only references to the underlying recursive type are the use of son_i and the predicate **isleaf** in the specification of **down**. Therefore the transformation is indeed mechanically executable.

3.4. Implementing Substituion by Tree Traversal

The type EXTENDED TREE uses a complex substitution operation that applies to all subtrees having a certain property. The purpose of tree traversal is to localize such subtrees, i.e. to make them into current trees. If no further information exists this traversal has to follow a depth-first search or a breadth-first search strategy. We will, however, formulate a transformation for the case where the subtree can be uniquely determined using the predicate P mentioned in section 3.3.

In section 3.2 we have enriched the basic type TREE by special operations such as **add**. As an intermediate step we do the same enrichment for traversable trees, now applying the operation to the current subtree. Hence, the new operation shall have the property $\mathsf{add'(tt,y)} = \mathsf{subst(tt, add(current(tt),y))}$, which according to the definition of the original **add** leads to the specification[3]

type TRAVERSABLE TREE1 **declares add :**

enrich TRAVERSABLE TREE **by**

opns: $\mathsf{add} : \mathsf{ttree} \times \mathsf{elem} \rightarrow \mathsf{ttree}$

axioms: ∀ **ttree tt, elem y :**

$\mathsf{add(tt,y)} = \mathsf{tt}$ **if** $\mathsf{item(current(tt))} = \mathsf{y}$

$= \mathsf{subst(tt\ ,\ cons(current(tt), leaf(y)))}$ **otherwise**

required: ∀ **ttree tt, elem y :**

$\mathsf{current(tt)}$ **is leaf** **for** $\mathsf{add(tt,y)}$

end of type

The following type combines the above substitution with the tree traversal that finds a suitable subtree. (Note that the predicate P is part of the definition of **down**.)

[3]The use of the same identifier does no harm since the environment always distinguishes them.

type EXTENDED TRAVERSABLE TREE **declares** **Add** :

enrich TRAVERSABLE TREE1 **by**

opns: **Add** : **ttree** × **elem** → **ttree**

axioms: ∀ **ttree tt, elem y** :

$\mathsf{Add(tt,y)} = \mathsf{Add(down(tt),y)}$ **if** ¬(**current(tt) is leaf**)

$\mathsf{Add(tt,y)} = \mathsf{add(tt,y)}$ **if** **current(tt) is leaf**

end of type

It is clear how in a procedural version of the program the function **Add** would become a simple loop. The important question is, however, under which conditions the above operation **Add** implements the one in section 3.2 (where implementation means that it yields one of the results contained in the nondeterministic choice). In other words, when does the predicate P lead to a subtree **s** compatible with the restrictions for **add** in the type TREE1?

Let R(s) stand for the restriction, in our case R(s) ="**s is leaf**". Then the applicability condition for the transformation is

$$\forall\, \mathsf{t} : \Big(\mathsf{P}(\mathsf{son}_i(\mathsf{t})) \wedge \big(\exists\, \mathsf{s} : \mathsf{s} \preceq \mathsf{t} \wedge \mathsf{R}(\mathsf{s})\big)\Big) \Rightarrow \Big(\exists\, \mathsf{s} : \mathsf{s} \preceq \mathsf{son}_i(\mathsf{t}) \wedge \mathsf{R}(\mathsf{s})\Big)$$

or in prenex form

$$\forall\, \mathsf{t}, \mathsf{s} :\ \exists\, \mathsf{s}' : \big(\mathsf{s} \preceq \mathsf{t} \wedge \mathsf{R}(\mathsf{s}) \wedge \mathsf{P}(\mathsf{son}_i(\mathsf{t}))\big) \Rightarrow \big(\mathsf{s}' \preceq \mathsf{son}_i(\mathsf{t}) \wedge \mathsf{R}(\mathsf{s}')\big)$$

This means that if there exists an admissible subtree at all then there must be one on the chosen path. For the particular predicate **s is leaf** of our example this is trivially true for arbitrary predicates P.

Transformation: MAKE-EXTENDED-TRAVERSABLE-TREE(PTREE,TREE1) uses (some enrichment of) the basic recursive type TREE together with the predicate P and the special operation **add** defined in TREE1 to produce the type EXTENDED TRAVERSABLE TREE (including the type TRAVERSABLE TREE1), provided that the above applicability condition is met.

Again the transformation only uses syntactic information available from the two given types to construct the new one. Hence it can be realized mechanically (except for the checking of the applicability condition.)

There are, of course, variants of this transformation. In particular, the search for an admissible subtree need not start at the root (using the operation **reset**) but may begin at the current subtree, since after the execution of one operation it is frequently known that the subsequent one will be applicable in the immediate vicinity (cf. appendix A).

The state we have reached by now is as follows:
Under the assumption that the earlier (nondeterministic) type EXTENDED TREE indeed implements the original type DICTIONARY the above type EXTENDED TRAVERSABLE TREE is an implementation, too. (**insert(s, x)** corresponds to **Add(reset(s), x)**). All the user has to do to generate this type is to give the initial recursive declaration of trees, the predicate P and the special *operation* **add**. The rest is achieved by the transformations.

3.5. Implementing Trees by Pointer Structures

A standard technique for implementing trees is to use pointer structures in some kind of "store" – be it disk storage, the central memory or an array. Since we want to stay within the framework of abstract data types we have to mimic this concept of a store by specifying a mapping that associates names ("indices", "addresses", "references") to values. The situation is very much the same as for "recursion removal" in classical transformation systems. One does not go all the way to loops but only changes the kind of recursion from, say, nested recursion to tail recursion. The direct correspondence between the latter and loops is taken for granted. Thus one can stay within the framework of applicative formulations and still approaches an operationally motivated goal.

As a further advantage of an abstract view of a store we need not distinguish between developments aiming at pointers (as is possible in, say, PASCAL or ALGOL68) or at arrays (as is necesssary in FORTRAN) or at external storage devices.

type STORE (**type** INDEX, **type** CONTENTS) **declares** **mapping**,access,alter,newindex :

based on: INDEX,CONTENTS

sorts: **mapping**

opns:
initialize : → **mapping**
access : **mapping** × **index** → **content**
alter : **mapping** × **index** × **content** → **mapping**
newindex : **mapping** → **index**

axioms: ∀ **mapping** m, **index** i, j, **content** x :
access(alter(m, i, x), i) = x
access(alter(m, i, x), j) = access(m, j) **if** i ≠ j

∀ **mapping** m : ∃ **index** i :
newindex(m) = i

required: ∀ **mapping** m, **index** i :
m ≠ initialize **for** access(m, i)
i ≠ newindex(m) **for** access(m, i)

end of type

Note that the operation **newindex** is incompletely specified (it should be viewed as an enrichment); all we require is that it is a totally defined operation and that the index differs from all those occurring in the given object m. Furthermore, the type STORE is parameterized since we may use it for various kinds of indices and contents.

It is clear that this type corresponds to classical notions in programming languages such as ALGOL or PASCAL. Without going into details we will briefly sketch here what the correspondences look like in the style of, say, PASCAL (cf. [10]).

(a) If the type **mapping** is to be interpreted as a file we get the following correspondences (where the notation of [10] is extended to the case of direct access files):
The type declaration reads: **type mapping = file of content;**
Further correspondences are: access(f, i) ⇔ get(f, i)
f := alter(f, i, x) ⇔ put(f, i, x)

(b) If the interpretation is made in terms of pointers, the correspondence looks slightly confusing since the object **mapping m** becomes anonymous (i.e. is not listed explicitly).
There is no type delaration for **mapping**.

Declaration of **index**:			**type index** = ↑**content**;
Further correspondences:	**var i : index** := **newindex**(m)	⇔	**var i** : ↑**content**;
	access(m, i)	⇔	i↑
	m := **alter**(m, i, x)	⇔	i↑ := x

(c) Finally, there is an interpretation in terms of (unbounded) arrays:

Declaration of **mapping**:			**type mapping** = **array of content**;
Further correspondences:	**access**(a, i)	⇔	a[i]
	a := **alter**(a, i, x)	⇔	a[i] := x

Because of its notational beauty we will stick to this last interpretation for the rest of this paper.

The type STORE can be used to specify an implementation of trees. Since the description of the formal treatment is quite lengthy we have moved it to the appendix. Here we will only give the basic idea: In pointer implementations the objects themselves are replaced by references to them. (In ALGOL68 this is the only way of writing down something like a recursive type declaration.) This leads to the two type declarations (where **index** stands for the references).

type tree′ = **leaf**(**elem item**) | **cons**(**index son1**, **index son2**)
type tree = (**mapping A**, **index r**)

where **mapping** is defined by the instantiation STORE(**index**, **tree′**) of the above type scheme.

As is well-known, an implementation of e.g. **cons**(a, b) means to build the disjoint union of two mappings and analogously **son1**(a) means to extract a submapping. But we are not interested in this full generality. We rather seek a transformation that implements a class of important special cases efficiently. Consider the example
t′ = t[s ⇐ u]
with s = **cons**(**cons**(a, b), c) and u = **cons**(**cons**(a, **leaf**(x)), **cons**(b, c)) .

Here the new tree u is built up from subtrees of the orignal tree s and primitives such as **leaf**(x). Consequently the mapping representing the new overall tree t′ will only slightly differ from the one representing t. Let m be the mapping representing the old tree t in the example above and let m′ be the new mapping representing t′. Then m′ is given by the expression (where the auxiliary identifiers shall not only increase readability but also indicate in which order the computations will take place in a procedural implementation). Note that all the identifiers s, a, etc. row stand for indices.

n_1 = **newindex**(m)
m_1 = **alter**(m, n_1, **cons**(b, c))
n_2 = **newindex**(m_1)
m_2 = **alter**(m_1, n_2, **leaf**(x))
m_3 = **alter**(m_2, **son1**(m(s)), **cons**(a, n_2))
m′ = **alter**(m_3, r, **cons**(**son1**(m(s)), n_1))

The appendix shows that the transition from the expression **substitute**(...) to an expression **alter**(...) indeed can be done in a fully mechanical and formal way. The resulting transformation simply captures a standard, yet burdensome and error-prone implementation technique in a

schematic form which then can be safely applied. And the above example is a convincing argument for the desirability of such a mechanical transition.

Transformation: The transformation POINTER-IMPLEMENTATION(TREE1) yields a new type where the special operations such as **add** of the type TREE1 are specified in terms of **alter**(...), provided they meet the requirements sketched above.

The mechanical nature of this transformation will be demonstrated in the appendix.

3.6. Pointer Implementations of Traversable Trees

It is trivial to combine the last two transformations. In the case of a traversable tree we get the declaration (**tree'** remaining the same)

type tree = (mapping A, index r, index c)

The operations **reset**, **up** and **down** only change the index **c** but leave the mapping invariant. The operation **subst** alters the mapping but leaves the index **c** untouched.

Transformations: TRAVERSABLE-POINTER-IMPLEMENTATION(TREE,TREE1,P) generates a type that provides operations such as **up**, **down**, **Add** etc. and specifies them in terms of the operations **access**, **alter** etc. of the type STORE.

The mechanization of this transformation is trivial once one knows how to do the transformation POINTER-IMPLEMENTATION. This means that our automatic program development has proceeded considerably further: Again assuming that the type EXTENDED TREE implements the type DICTIONARY we have now arrived at a pointer implementation of the type DICITONARY. And still the user only had to provide the tree declaration and the operations **add**, **delete**, etc.

3.7. Tree Traversal for Several Processes

As soon as we work in a concurrent environment there arises the need for several "current subtrees", which is, of course, a straightforward generalization of the type TRAVERSABLE TREE. Conceptually this leads to an (n + 1)-tuple of trees, where n trees are subtrees of the n + 1st one. The only complication will be that we have to keep the subtree relations between various current trees intact. The auxiliary type **name** denotes a set of names for the processes working on the tree (without loss of generality we may again take the integral numbers here). The predicates P_μ are as in section 3.3, the meaning of the underlining will be explained in the next section.

type MULTIPLE TREE **declares** **mtree,reset,down,up,current** :

based on: PTREE

sorts:	**mtree**			
opns:	init :	**tree**	→	**mtree**
	reset :	**name × mtree**	→	**mtree**
	down :	**name × mtree**	→	**mtree**
	up :	**name × mtree**	→	**mtree**
	root :	**mtree**	→	**tree**
	current :	**name × mtree**	→	**tree**

axioms: $\forall$ **tree s, mtree tt, name** π, μ :

$\mathsf{root}(\mathsf{init}(s)) = s$

$\mathsf{root}(\underline{\mathsf{reset}}_\mu(tt)) = \mathsf{root}(tt)$

$\mathsf{root}(\underline{\mathsf{down}}_\mu(tt)) = \mathsf{root}(tt)$

$\mathsf{current}_\mu(\underline{\mathsf{reset}}_\mu(tt)) = \mathsf{root}(tt)$

$\mathsf{current}_\pi(\underline{\mathsf{reset}}_\mu(tt)) = \mathsf{current}_\pi(t)$ **if** $\pi \neq \mu$

$\mathsf{current}_\mu(\underline{\mathsf{down}}_\mu(tt)) = \mathsf{son}_i(\mathsf{current}_\mu(tt))$ **if** $\mathsf{P}_\mu(\mathsf{son}_i(\mathsf{current}_\mu(tt))) = \mathbf{true}$

$\mathsf{current}_\pi(\underline{\mathsf{down}}_\mu(tt)) = \mathsf{current}_\pi(tt))$ **if** $\pi \neq \mu$

$\underline{\mathsf{up}}_\mu(\underline{\mathsf{down}}_\mu(tt)) = tt$

$\underline{\mathsf{up}}_\pi(\underline{\mathsf{down}}_\mu(tt)) = \underline{\mathsf{down}}_\mu(\underline{\mathsf{up}}_\pi(tt))$ **if** $\pi \neq \mu$

required: $\forall$ **mtree tt, name** π, **tree s** :

$\mathsf{current}_\pi(tt) \preceq \mathsf{root}(tt)$

$\mathsf{P}_\pi(\mathsf{current}_\pi(tt)) = \mathbf{true}$

$\mathsf{current}_\pi(tt) \neq \mathsf{root}(tt)$ **for** $\underline{\mathsf{up}}_\pi(tt)$

$\neg(\mathsf{current}_\pi(tt)$ **is leaf**$)$ **for** $\underline{\mathsf{down}}_\pi(tt)$

end of type

The same extension as for simple traversable trees adds an operation "substitute" that replaces one of the current subtrees. However, the requirement that all other current trees remain subtrees adds some further restrictions.

type MULTIPLE TREE$^+$ **declares subst :**

enrich MULTIPLE TREE $\oplus$ TREE$^+$ **by**

opns: subst : **name** $\times$ **mtree** $\times$ **tree** $\rightarrow$ **mtree**

axioms: $\forall$ **mtree tt, tree s, name** μ, π :

$\mathsf{root}(\mathsf{subst}_\mu(tt, s)) = \mathsf{root}(tt)[\mathsf{current}_\mu(tt) \Leftarrow s]$

$\mathsf{current}_\mu(\mathsf{subst}_\mu(tt, s)) = s$

$\mathsf{current}_\pi(\mathsf{subst}_\mu(tt, s)) = \mathsf{current}_\pi(tt)[\mathsf{current}_\mu(tt) \Leftarrow s]$ **if** $\mu \neq \pi$

required: $\forall$ **mtree tt, tree s, name** μ, π :

$\mathsf{P}_\mu(s) = \mathbf{true}$ **for** $\mathsf{subst}_\mu(tt, s)$

$\mathsf{current}_\pi(tt) \prec \mathsf{current}_\mu(tt) \Rightarrow \mathsf{current}_\pi(tt) \preceq s$ **for** $\mathsf{subst}_\mu(tt, s)$

end of type

The third equation and the restriction are both enforced by the requirement that invariantly $\mathsf{current}_\pi(tt) \preceq \mathsf{root}(tt)$. Note that the restriction allows the substitution of a subtree that is current for two processes. (The symbol $\prec$ means "is subtree but not equal".)

As in section 3.4 we now get operations add_μ and Add_μ:

type MULTIPLE TREE1 **declares add :**

enrich MULTIPLE TREE **by**

opns: add : **name** $\times$ **mtree** $\times$ **elem** $\rightarrow$ **mtree**

axioms: $\forall$ **mtree tt, elem y, name** μ :

$\mathsf{add}_\mu(\mathsf{tt},\mathsf{y}) = \mathsf{tt}$ **if** $\mathsf{item}(\mathsf{current}_\mu(\mathsf{tt})) = \mathsf{y}$

$= \mathsf{subst}_\mu(\mathsf{tt}\,,\ \mathsf{cons}(\mathsf{current}_\mu(\mathsf{tt}),\mathsf{leaf}(\mathsf{y}))\,)$ **otherwise**

required: $\forall$ **mtree tt, elem y, name** μ :

$\mathsf{current}_\mu(\mathsf{tt})$ **is leaf** **for** $\mathsf{add}_\mu(\mathsf{tt},\mathsf{y})$

end of type

type EXTENDED MULTIPLE TREE **declares Add :**

enrich MULTIPLE TREE1 **by**

opns: Add : **name** $\times$ **mtree** $\times$ **elem** $\rightarrow$ **mtree**

axioms: $\forall$ **mtree tt, elem y, name** μ :

$\mathsf{Add}_\mu(\mathsf{tt},\mathsf{y}) = \mathsf{Add}_\mu(\underline{\mathsf{down}}_\mu(\mathsf{tt}),\mathsf{y})$ **if** $\neg(\,\mathsf{current}_\mu(\mathsf{tt})$ **is leaf** $)$

$\mathsf{Add}_\mu(\mathsf{tt},\mathsf{y}) = \mathsf{add}_\mu(\mathsf{tt},\mathsf{y})$ **if** $\mathsf{current}_\mu(\mathsf{tt})$ **is leaf**

end of type

The pointer implementation, too, is a straightforward extension of the principles used in section 3.5 and 3.6. Now we get

type tree = $(\mathbf{mapping}\ \mathsf{A}, \mathbf{index}\ \mathsf{r}, \mathbf{index}\ \mathsf{c}_1, \ldots, \mathsf{c}_n)$

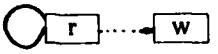
where the operations reset_μ, up_μ and down_μ only change the index c_μ while subst_μ alters the mapping A.

The type MULTIPLE TREE (respectively its implementation) specifies the overall behavior of the concurrent processes including their shared and private variables. In this sense the axioms and requirements correspond to the "always"-operator $\Box$ of temporal logic. In the next section we have to discuss the relationship between this type and concurrent programs. In particular, it must be decided how the interleaving of sequences of operations takes place, for so far every process completes its operation, say Add_μ, before another one starts a new operation, say Delete_π.

3.8. Concurrency and Protection

We deal here with a particular class of concurrency problems: Synchronization is only needed to keep processes from interfering with each other while accessing a common variable. Therefore it is justified that we base our approach on the so-called multiprogramming-assumption, where parallelism is modelled by sequential interleaving. The essential point here is that the operations are assumed to be *atomic*. Whenever an implementation be it hardware or software realizes such a "conceptually atomic" operation by a sequence of more elementary operations, there must be a mechanism that treats this sequence as an indivisible unit. In the literature various such mechanisms are known (semaphores, locks, conditional critical regions, etc.) We will use here the one that has been tailored to the use in large structered objects, viz. locks. In their simplest form, locks act like semaphores that are associated to the shared variables of processes. Let v be such a variable. Then lock(v) blocks that variable (no two processes may hold locks on a variable simultaneously) and unlock(v) frees it. This mechanism gains considerable expressive power by

introducing different kinds of locks, for example read- and write-locks. The compatibility relation between these locks is usually specified by graphs such as



where a solid arc states that two different processes may hold the respective locks on the variable simultaneously, and a dotted arc means that a process may convert its lock from one kind to the other. Such behaviors are easily described by abstract data types. But there are additional problems that are more intrinsic and quite lengthy; therefore we have moved this discussion to the appendix and treat the subject here in a more traditional way.

The goal of our development is best described by an example indicating the initial and the final state of our intended development. (The names μ may be omitted in these protocols since they are implicitly given by the respective processes.)

Example 1: We are given processes that issue operations of the type DICTIONARY.

$$[\![\ \ldots \ \| \ P_\mu :: \ \ldots \ \text{w-lock}(s);\ s := \text{insert}(s,x);\ \text{w-unlock}(s) \ \ldots \ \| \ \ldots \ ]\!]$$

In the final program this shall become (where according to the pointer implementation of section 3.6 the index c denotes the current subtree of the process under consideration and an array notation is used)

```
[[ ... || P_μ :: ... r-lock(s[c]);
                     until s[c] is leaf
                     do c' := c; c := down(s[c]); r-unlock(s[c']); r-lock(s[c]) od;
                     w-lock(s[c]); s[c] := add(s[c], x); w-unlock(s[c])
          ... || ... ]]
```

(*End of example*)

The purpose of the transformation to be found is therefore twofold: When the recursive function insert is implemented by a loop, the locking shall not continue throughout the iteration. Furthermore, not the whole object but only the currently accessed part is to be locked. In the sequel we will deal with both problems in turn.

Recall the equivalence relation of section 3.1, where two trees are equivalent if they have the same set of leaves. Under this equivalence relation the operations reset, up and down are the identity and therefore can be taken out of the critical region. We delay the details to the appendix and assume for the time being that certain operations – – marked here by underlining allow the following (informal) transformations

(T1)
$$\frac{\text{lock}(t);\ t := \underline{\text{down}}(t)}{\Downarrow}$$
$$\text{r-lock}(t);\ t := \underline{\text{down}}(t);\ \text{r-unlock}(t);\ \text{lock}(t)$$

This means that a down-operation that immediately follows a locking allows an interruption of the locking after its execution. The analogous rule holds for an unlocking after the down-operation. But note that such an interruption is not permitted in the middle of a critical region. In addition we need technical rules such as distributivity over conditionals that have to be extended from classical programming constructs to locks:

(T2)
$$\frac{\text{lock}(t);\ \textbf{if}\ p(t)\ \textbf{then}\ A\ \textbf{else}\ B\ \textbf{fi}}{\Downarrow}$$
$$\text{r-lock}(t);\ \textbf{if}\ p(t)\ \textbf{then}\ \text{lock}(t);\ A\ \textbf{else}\ \text{lock}(t);\ B\ \textbf{fi}$$

In both transformations lock stands for either read-locks or write-locks.

These rules together with the meanwhile classical fold/unfold techniques of Burstall and Darlington suffice to restrict the locking to the actually necessary periods of time: We start from the sequence of operations that stems from implementing **insert** by **Add**

w-lock(t); t := Add(t, x); w-unlock(t)

and make it into a the procedure

proc Add* = (**mtree** t, **elem** x)**mtree** : w-lock(t); t := Add(t, x); w-unlock(t)

Unfolding of the recursive definition of **Add** given in the type EXTENDED TRAVERSABLE TREE yields after a few minor simplifications

```
proc Add* = (mtree t, elem x)mtree :
    w-lock(t);
    if ¬(t is leaf) then t := down(t); t := Add(t, x); w-unlock(t)
                    else t := add(t, x); w-unlock(t)                fi
```

Application of the above transformations and some simplifications yield the version

```
proc Add* = (mtree t, elem x)mtree :
    r-lock(t);
    if ¬(t is leaf) then t := down(t); r-unlock(t); w-lock(t); t := Add(t, x); w-unlock(t)
                    else w-lock(t); t := add(t, x); w-unlock(t)                          fi
```

Now we have an instance of the original definition of **Add***, which allows folding and thus produces a version that directly corresponds to a loop:

```
proc Add* = (mtree t, elem x)mtree :
    r-lock(t);
    if ¬(t is leaf) then t := down(t); r-unlock(t); Add*(t, x)
                    else w-lock(t); t := add(t, x); w-unlock(t) fi
```

The second problem to be addressed here is that of locking only the actually accessed nodes instead of the whole tree. After the application of the transformation of sections 3.5, 3.6 we have a collection of individual objects in the place of the monolithic single tree. (We will use the array notation here.) The following transformation describes the transition from the locking of the whole array to the locking of single entries:

(T3)
$$\frac{\text{lock}(a);\ a[i] := e;\ \text{unlock}(a)}{\Downarrow}$$
$$\text{lock}(a[i]);\ a[i] := e;\ \text{unlock}(a[i])$$

If several elements are accessed in the critical region they have to be locked simultaneously. (As is well known such a simultaneous locking can be implemented sequentially by establishing an order relation between the single elements. In our case "father before son" would be an obvious choice.) This simultaneous locking even can distinguish read- and write-locks. Of course, an application of this transformation is only reasonable if the number of elements accessed in the critical region is limited.

There remains one issue. The abstract data type MULTIPLE TREE requires that a substitution must not take place if certain subtrees are the current trees of other processes. In a sequential environment violation of such a requirement causes the program execution to abort. In a concurrent environment we would rather see the process wait until some other process has resolved the violation. (If no other process is kind enough to do this we have infinite waiting[4].) The simple solution is a re-interpretation of the undefinedness expressed in the restrictions of our data types. We will adopt this solution here for the sake of brevity although it has severe drawbacks from a practical point of view: A process repeatedly locks and unlocks the variable in question just to see whether the violation still exists. In this way, it continuously interferes with the other processes. Furthermore, testing of the violation means that it has to know what the current subtrees of the other processes are. Both problems are typical candidates for a solution by locks. Therefore we might introduce two new kinds of locks, one standing for "I am here" and the other standing for "I am the only one here". The former is compatible with both read- and write-locks, the latter with no lock. Although this leads to the most practical solution we will not pursue the subject here further.

3.9. Summing up the Transformations

What have we gained by the lengthy considerations of this chapter?

Assume that the aforementioned tools and transformations indeed are at our disposal (maybe even aided by a mechanical system). If we now have to develop a program that applies some fairly complex operations to a certain tree structure i.e. to a recursive data stucture then all we have to do is specify the individual tree transformations (for example, how a new leaf may be added, how restructerings can be done etc.) Since this specification is not burdened by implementation details, it very clearly reveals the underlying algorithmic ideas.

Once this step is done, the rest of the implementation can be done almost automatically by applying our transformation rules. The result is a program that does the desired operations in a pointer implementation and possibly even in a concurrent environment.

The rest of this paper will present an application, viz. the development of concurrent operations on 2-3-trees. According to the aforementioned principles it will suffice to describe the various individual operations as basic tree operations.

---

[4]Compare the ongoing dispute in the area of programming language semantics whether it is admissible to identify abortion and nontermination.

4. An Application: Implementing Sets by 2-3-Trees

How do the aforementioned techniques apply in a concrete program development? First of all, recall that it suffices to cast one's ideas about the algorithm into basic tree transformations. The rest of the development is done using the rules of the previous chapter. Therefore we will develop here a sequence of more and more refined versions of trees together with suitable enrichments until all our intentions are met. These intentions are essentially those given in the literature on 2-3-trees and B-trees and they will be explained in due course. This leads to a nicely structured development process where we cope with one issue at a time.

4.1. Basic 2-3-Trees

The average performance of the tree operations is best when the trees are "balanced', i.e. when all paths have the same length. Obviously addition as well as deletion violate this property unless a restructering of the tree takes place. The costs of these restructerings can be kept low — in the order *log N* — if 2-3-trees (or more generally B-trees) are used. In a 2-3-tree every node (except for the leaves) has either two or three sons. The type 2-3-TREE is therefore defined by

```
type tree = leaf(elem item) |
            cons2(tree son1, tree son2) |
            cons3(tree son1, tree son2, tree son3)
```

(We will often use the notation . **is cons** as an abbreviation for . **is cons**2 ∨ . **is cons**3.) In data base applications one usually works with the more general B-trees (cf. [4]), where every node has between m and $2m - 1$ sons. Clearly 2-3-trees are just the special case $m = 2$. For simplifying the presentation we will restrict ourselves to this special case.

4.2. Ordered Trees

A first decision that considerably increases the efficiency of the intended implementation is to use only "ordered trees". The straightforward specification of this requirement would be to use functions **min** and **max** that yield the smallest and largest leaf of the tree. But the values of these functions may change on deletion or addition. Therefore it is better to work with upper and lower bounds. This leads to the following redefinition of the basic 2-3-trees into

```
type tree = leaf(elem low, elem item, elem high) |
            cons2(tree son1, tree son2) |
            cons3(tree son1, tree son2, tree son3)
```

Using these bounds the ordering requirement is easily specified. (By +1 we denote the successor of an element of the type **elem**, which is from now on assumed to be linearly ordered.)

type ORDERED 2-3-TREE **declares** lwb,upb :

enrich 2-3-TREE **by**

opns: upb : tree → elem
lwb : tree → elem

axioms: ∀ tree t :

$$\begin{aligned} lwb(t) &= low(t) && \textbf{if } t \textbf{ is leaf} \\ &= lwb(son1(t)) && \textbf{if } t \textbf{ is cons} \\ upb(t) &= high(t) && \textbf{if } t \textbf{ is leaf} \\ &= upb(son2(t)) && \textbf{if } t \textbf{ is cons2} \\ &= upb(son3(t)) && \textbf{if } t \textbf{ is cons3} \end{aligned}$$

required: ∀ tree t :

$$upb(son1(t)) + 1 = lwb(son2(t))$$
$$upb(son2(t)) + 1 = lwb(son3(t)) \quad \textbf{if } t \textbf{ is cons3}$$

end of type

Note that this is a specification and does not prescribe any particular implementation. The optimal solution, where the information about the bounds is distributed all over the tree (cf. [1]), is still included here.

The ordering also provides us with a predicate P as required in section 3.1[5]: For an arbitrary value y - *actually the one to be searched in the tree* — *the predicate* $P_y(t)$ is $lwb(t) \leq y \leq upb(t)$. Due to the definitions of lwb and upb this is true for exactly one of the sons if it is true for the father. Thus we introduce the type

type PTREE **declares** P :

enrich ORDERED 2-3-TREE **by**

opns: P : tree → bool

axioms: ∀ tree t, elem y :

$$\begin{aligned} P_y(t) &= true && \textbf{if } lwb(t) \leq y \leq upb(t) \\ &= false && \textbf{otherwise} \end{aligned}$$

required: ∀ tree t :

$$(P(son1(t)) \vee P(son2(t))) = true \quad \textbf{if } t \textbf{ is cons} \wedge P(t) = true$$
$$(P(son1(t)) \wedge P(son2(t))) = false \quad \textbf{if } t \textbf{ is cons} \wedge P(t) = true$$

end of type

The requirements are fulfilled by definition.

[5] As a matter of fact, this requirement motivates the introduction of an ordering.

4.2. ADDITION AND DELETION OF LEAVES

The basic operations needed for implementing sets in terms of trees are addition and deletion of leaves. This can be done according to the following figures.

Example 2: Consider the addition of the value 2 to the trees given below; this leads to the following tree transformations

Case 1: addition of the value 2

Case 2: addition of the value 2

(End of example)

Example 3: Assume that we want to delete the value 2 from the tree given below.

Case 3: removal of the value 2

The problem with this solution is that the node r disappears when it has only two sons one of which is to be deleted. This is unpleasant when it comes to concurrency, since such problems are candidates for deadlock situations. To shorten the presentation we adopt another solution and introduce the notion of an empty tree (a leaf without an item). This means that we add the variant **empty(elem low, elem high)** to the recursive type declaration above and extend the type ORDERED 2-3-TREE accordingly. By **s is terminal** we abbreviate (**s is leaf** $\vee$ **s is empty**). Thus we get the transformation (where no nodes need to be eliminated for the time being).

Case 4: deletion of the value 2

(End of example)

The specification of these operations according to the above figures is easy. To shorten the writing-down we neglect the optimal solutions of cases 1 and 3 and treat all situations according to cases 2 and 4.

type TREE1 **declares** add,delete :

enrich PTREE **by**

opns: add : tree × elem → tree
delete : tree × elem → tree

axioms: ∀ elem x, y, l, h :

add(leaf(l, x, h), y)	= cons 2(leaf(l, x, x), leaf(x + 1, y, h))	**if** x < y
	= leaf(l, x, h)	**if** x = y
	= cons 2(leaf(l, y, y), leaf(y + 1, x, h))	**if** x > y
add(empty(l, h), y)	= leaf(l, y, h)	
delete(leaf(l, x, h), y)	= leaf(l, x, h)	**if** x ≠ y
	= empty(l, h)	**if** x = y
delete(empty(l, h), y)	= empty(l, h)	

required: ∀ tree s, elem y :

s is terminal ∧ lwb(s) ≤ y ≤ upb(s)	**for** add(s, y)
s is terminal ∧ lwb(s) ≤ y ≤ upb(s)	**for** delete(s, y)

end of type

The transformations of section 3.2 now produce a type EXTENDED TREE or a type EXTENDED TRAVERSABLE TREE, which are both implementations of the type DICTIONARY. Note that as a byproduct of the ordering the subtree where the addition and deletion take place is uniquely determined, thus making the potential nondeterminism in the transformation of section 3.2 harmless. Also, the extension to multiple trees is easy since no substitution erases a subtree.

5. Repairing Degenerate Trees

The operators defined above produce trees that may be degenerate in either of two ways: There may be empty subtrees as the result of a deletion; these subtrees should be erased. And there may be search paths through the tree that are considerably longer than other paths; but for a better overall performance of the tree operations all paths should have the same length. The purpose of working with 2-3-trees instead of ordinary binary trees is to allow the re-balancing of degenerate trees without too much overhead.

The principal idea of the repairing can – by abusing a terminology of artificial intelligence – be sketched as follows: The operations add and delete search the leaf in question and then apply the necessary changes. These changes may cause a disturbance of the balancing. Whenever such a degeneration exists "a demon is triggered" that tries to repair the tree. The actions of this demon may lead to further disturbances which are treated in turn until the tree is in perfect shape again. This principle leaves a number of choices: The "demon" may be one or more processes that are part of the data strucuture itself such that the requesting processes are not even aware of something going on in the data base after they have finished their task. But one also may charge the process responsible for the degeneration with the task of the demon by forcing it to do all necessary repairs before leaving the data base. In both cases there is a further decision possible, viz. the decision

whether the treatment of one degeneration may produce at most one or several new disturbances. These issues will be treated in detail later on.

5.1. BALANCED TREES

To express the balancing requirement we need an operation that determines the "height" of a tree, i.e. its distance from the leaves. However, in the case of unbalanced trees such an operation is not well-defined. To overcome this deficiency we associate to each (sub)tree a "disturbance" that is $\geq +1$ if the tree is too high in comparison with its environment and ≤ -1 if it is too short; otherwise the disturbance is 0. Then we can replace our previous notion of height by a "virtual height", which is corrected by the disturbances. This calls for another modification of the original type 2-3-TREE:

type tree = **empty**(**int dist**, **elem low**, **elem high**) |
leaf(**int dist**, **elem low**, **elem item**, **elem high**) |
cons 2(**int dist**, **tree son1**, **tree son2**) |
cons 3(**int dist**, **tree son1**, **tree son2**, **tree son3**)

The balancing requirement now can be formulated by the type

type BALANCED 2-3-TREE **declares height :**

enrich 2-3-TREE **by**
opns: **height** : **tree** → **int**
axioms: ∀ **tree t** :
height(t) = −**dist**(t) **if** t **is terminal**
= **height**(**son1**(t)) + 1 − **dist**(t) **if** t **is cons**
required: ∀ **tree t** :
height(**son1**(t)) = **height**(**son2**(t)) **if** t **is cons**
height(**son2**(t)) = **height**(**son3**(t)) **if** t **is cons 3**
end of type

The height gives for any path p from the root to a leaf the value "number of edges on p"−"sum of all disturbances along p". For easier readibility we will from now on use the notation $\mathbf{cons}\,2^{\delta}(\mathbf{r}, \mathbf{s})$ instead of $\mathbf{cons}\,2(\delta, \mathbf{r}, \mathbf{s})$ (analogously for **leaf** and **cons 3**).

The change in the definition of 2-3-trees requires an according modification of the operations **add** and **delete**:

type 2-3-TREE1 **declares add**,**delete** :

enrich PTREE ⊕ BALANCED 2-3-TREE **by**
opns: **add** : **tree** × **elem** → **tree**
delete : **tree** × **elem** → **tree**

axioms: $\forall$ **elem** x, y, l, h, **int** α :

$$\begin{aligned}
\mathsf{add}(\mathsf{leaf}^{\alpha}(\mathsf{l},\mathsf{x},\mathsf{h}),\mathsf{y}) &= \mathsf{cons}\,2^{\alpha+1}(\mathsf{leaf}^{0}(\mathsf{l},\mathsf{x},\mathsf{x}),\mathsf{leaf}^{0}(\mathsf{x}+1,\mathsf{y},\mathsf{h})) && \textbf{if}\quad \mathsf{x}<\mathsf{y}\\
&= \mathsf{leaf}^{\alpha}(\mathsf{l},\mathsf{x},\mathsf{h}) && \textbf{if}\quad \mathsf{x}=\mathsf{y}\\
&= \mathsf{cons}\,2^{\alpha+1}(\mathsf{leaf}^{0}(\mathsf{l},\mathsf{y},\mathsf{y}),\mathsf{leaf}^{0}(\mathsf{y}+1,\mathsf{x},\mathsf{h})) && \textbf{if}\quad \mathsf{x}>\mathsf{y}\\
\mathsf{add}(\mathsf{empty}^{\alpha}(\mathsf{l},\mathsf{h}),\mathsf{y}) &= \mathsf{leaf}^{\alpha}(\mathsf{l},\mathsf{y},\mathsf{h})\\
\mathsf{delete}(\mathsf{leaf}^{\alpha}(\mathsf{l},\mathsf{x},\mathsf{h}),\mathsf{y}) &= \mathsf{leaf}^{\alpha}(\mathsf{l},\mathsf{x},\mathsf{h}) && \textbf{if}\quad \mathsf{x}\neq\mathsf{y}\\
&= \mathsf{empty}^{\alpha}(\mathsf{l},\mathsf{h}) && \textbf{if}\quad \mathsf{x}=\mathsf{y}\\
\mathsf{delete}(\mathsf{empty}^{\alpha}(\mathsf{l},\mathsf{h}),\mathsf{y}) &= \mathsf{empty}^{\alpha}(\mathsf{l},\mathsf{h})
\end{aligned}$$

required: $\forall$ **tree** s, **elem** y :

$$\begin{aligned}
&\mathsf{s\ is\ terminal} \wedge \mathsf{lwb}(\mathsf{s}) \le \mathsf{y} \le \mathsf{upb}(\mathsf{s}) && \textbf{for}\quad \mathsf{add}(\mathsf{s},\mathsf{y})\\
&\mathsf{s\ is\ terminal} \wedge \mathsf{lwb}(\mathsf{s}) \le \mathsf{y} \le \mathsf{upb}(\mathsf{s}) && \textbf{for}\quad \mathsf{delete}(\mathsf{s},\mathsf{y})
\end{aligned}$$

end of type

By the same construction as before this becomes an implementation of DICTIONARY. Note that height(add(s,y)) = height(s) and height(delete(s,y)) = height(s).

5.2. LOCAL REBALANCING

The following figures shall give a first intuitive idea of the operations that are used to repair degenerate trees. The notation $\mathsf{r}_{(\rho)}$ indicates the disturbance of r.

Example 4: Empty subtrees have to be eliminated. The only problematic case arises when there is only one brother. The resulting shortening of the tree has to be compensated by the disturbances. (Note that a may be empty as well.)

Case 1: Eliminating empty subtrees

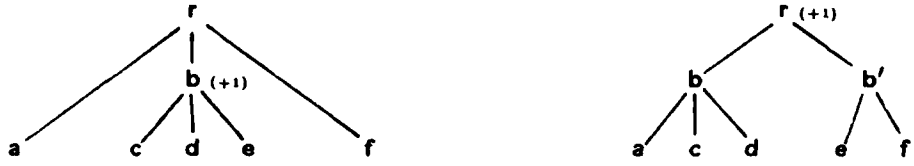
(*End of example*)

For the rest of this section we will be concerned with repairing disturbed balances. For simplicity we assume in the illustrations below that the node b (or both a and b) has a disturbance +1 while the other ones have disturbances 0.

Example 5: Assume that the node b in the following trees is unbalanced, i.e. has a true height that is (by 1) greater than that of its brothers.

Case 2: rebalancing in case of two sons and one brother

Case 3: rebalancing in case of three sons and one brother



Case 4: rebalancing in case of two brothers

Case 5: rebalancing in case of two high subtrees

Case 6: rebalancing in case of two high subtrees

Note: Case 5 essentiallly also applies if b has only two sons (a having either two or three sons). Of course, there also are various symetrical situations where the roles of brothers are exchanged.

How do more complex situations of disturbances influence these pictures? Let us consider one representative example in full detail, using the algebraic tools provided by the data type technique. The intuitive idea reflected in figure 4 leads to the general equation (with the indeterminates $\alpha', \ldots, \sigma'$):

$r_{old} = \mathsf{cons}3^{\rho}(\, a^{\alpha}\,,\ \mathsf{cons}3^{\beta}(c^{\gamma}, d^{\delta}, e^{\epsilon})\,,\ f^{\varsigma}\,)$

$r_{new} = \mathsf{cons}2^{\rho'}(\, \mathsf{cons}3^{\beta'}(a^{\alpha'}, c^{\gamma'}, d^{\delta'})\,,\ \mathsf{cons}2^{\sigma'}(e^{\epsilon'}, f^{\varsigma'})\,)$

To determine the new disturbances we use the following design criteria:

- It must be possible to substitute the new tree for the old one in any given environment. Therefore we have to require $\mathsf{height}(r_{new}) = \mathsf{height}(r_{old})$.

- The node which is the focus of attention — in our case **b** — has to be completely repaired. This means $\beta' = 0$.

- The transition should affect as few nodes as possible. This means in particular that we do not want to alter the disturbances of **c**, **d** and **e**. Therefore we require $\gamma' = \gamma$, $\delta' = \delta$, $\epsilon' = \epsilon$ (provided that this does not lead to an inconsistency).

The calculations center around the axioms and restrictions of the type BALANCED 2-3-TREE. The disturbance σ' of the node $\mathbf{b}'$ is determined by the decision $\beta' = 0$ and by the two restrictions

$\mathsf{height}(\mathsf{cons}\,3^{\beta'}(\ldots)) = \mathsf{height}(\mathsf{cons}\,2^{\sigma'}(\ldots)) \wedge \mathsf{height}(\mathbf{e}^{\epsilon'}) = \mathsf{height}(\mathbf{e}^{\epsilon}) = \mathsf{height}(\mathbf{d}^{\delta}) = \mathsf{height}(\mathbf{d}^{\delta'})$
$\vdash \quad \mathsf{height}(\mathbf{d}^{\delta'}) + 1 - \beta' = \mathsf{height}(\mathbf{e}^{\epsilon'}) + 1 - \sigma'$
$\vdash \quad \sigma' = \beta' = 0.$

Analogously, the equations $\mathsf{height}(\mathbf{d}^{\delta}) = \mathsf{height}(\mathbf{d}^{\delta'})$ and $\beta' = 0$ allow us to determine ρ':

$\mathsf{height}(\mathbf{r}_{old}) = \mathsf{height}(\mathbf{r}_{new})$
$\vdash \quad \mathsf{height}(\mathbf{d}^{\delta}) + 1 - \beta + 1 - \rho = \mathsf{height}(\mathbf{d}^{\delta'}) + 1 - \beta' + 1 - \rho' = \mathsf{height}(\mathbf{d}^{\delta'}) + 2 - \rho'$
$\vdash \quad \rho' = \rho + \beta$

It remains to consider α' and ζ'. Using the last result $\rho' = \rho + \beta$ this becomes:

$\mathsf{height}(\mathbf{r}_{old}) = \mathsf{height}(\mathbf{r}_{new})$
$\vdash \quad \mathsf{height}(\mathbf{a}^{\alpha}) + 1 - \rho = \mathsf{height}(\mathbf{a}^{\alpha'}) + 1 - \beta' + 1 - \rho' = \mathsf{height}(\mathbf{a}^{\alpha'}) + 2 - \rho - \beta$
$\vdash \quad \alpha' = \alpha - \beta + 1 \qquad$ and analogously $\qquad \zeta' = \zeta - \beta + 1.$

When doing the same computations for the other cases it turns out that the last equation ($\alpha' = \alpha - \beta + 1$) always holds for the "lower" brothers. The father **r** always gets a new disturbance of the kind $\rho' = \rho + \beta$, except for case 2 where it is $\rho' = \rho + \beta - 1$. (As a rule of thumb use the fact that for every path in the pictures the value "sum of all disturbances minus number of edges" must be the same in both trees.)

Before we cast the above results into an abstract data type we should address a few issues here.

- Assume that we start from an initial tree of the form $\mathsf{leaf}^0(\ldots)$. This tree has height 0. Since all our operations are designed such that they leave the height invariant and since this height is (for any path **p** from the root to a leaf) equal to
  "number of edges of **p**" − "sum of disturbances along **p**"
  it is impossible to set all disturbances to 0 Therefore a "truly balanced" tree contains at most one nonzero disturbance, viz. the one at the root, and this disturbance states the length of all paths. (The identifier "height" has thus lost its mnemonic meaning for the root.)

- Working with both positive and negative disturbances minimizes the number of nodes to be changed. But we could equally well use only positive disturbances: Consider a node **r** with disturbance $-\delta < 0$. If we subtract δ from the father and add δ to all brothers the disturbance of **r** becomes 0. This doesn't change anything in the cases 2 — 6.

- The sum of all disturbances of the tree decreases in cases 2 — 6 with two exceptions, namely cases 3 and 4 when $\beta = 1$. In this case the sum remains invariant but the disturbance moves closer to the root. Therefore the rebalancing will eventually terminate (provided that no further additions or deletions take place).

- Each of the repair operations keeps the set of leaves of the tree invariant and therefore is the identity in our underlying equivalence relation.

5.3. An Abstract Data Type for Rebalancing

According to the figures of the previous section a subtree with root r is degenerated if at least one of its sons is empty or carries a nonzero disturbance. (Note that disturbances of the root of the whole tree are neglected.) To cope with the explosion of case distinctions we introduce a few notations and definitions. In the figures above there were three levels of nodes: the father r, the son b (or the sons a and b) with the highest disturbance, and finally the grandsons and the sons with smaller disturbances. The tuple $\langle t_1, \ldots, t_n \rangle$ shall denote the nodes on the lowest level from left to right. For $r = \mathsf{cons}3(a, b, c)$ this can be formalized as follows (analogously for cons2):

$\langle t_1, \ldots, t_n \rangle = \langle \mathcal{T}(a), \mathcal{T}(b), \mathcal{T}(c) \rangle$
where for any son s^σ of r

$$\mathcal{T}(s^\sigma) = \begin{cases} \langle\rangle, & \text{if s is empty;} \\ s^{\sigma - maxdist + 1}, & \text{if } \sigma < \mathsf{maxdist}; \\ \langle \mathsf{son1}(s), \mathsf{son2}(s), \mathsf{son3}(s) \rangle, & \text{if } \sigma = \mathsf{maxdist}. \end{cases}$$

$$\mathsf{maxdist} = \begin{cases} \max(\mathsf{dist}(a), \mathsf{dist}(b), \mathsf{dist}(c)) + 1, & \text{if } \mathsf{dist}(a) = \mathsf{dist}(b) = \mathsf{dist}(c) \\ \max(\mathsf{dist}(a), \mathsf{dist}(b), \mathsf{dist}(c)), & \text{otherwise} \end{cases}$$

Note that all t_i have the same height. (The above formulae are just the generalization of the sample calculation carried out in the previous section.) Also, the strange definition of maxdist allows us to include the case where all disturbances are equal in a uniform way (the disturbance is just moved to the father).

The form of the transformed tree now only depends on the number n of elements in the tuple, e.g.

$$t = \langle t_1, t_2, t_3 \rangle \quad \Rightarrow \quad r_{new} = \mathsf{cons}3^{\rho'}(t_1, t_2, t_3) \quad \text{where } \rho' = \rho + \mathsf{maxdist} - 1$$

Using these shorthand notations we can specify the operation rebalancing by the following type:

type 2-3-TREE2 **declares** rebalance :

enrich PTREE ⊕ BALANCED 2-3-TREE **by**

opns: rebalance : **tree** → **tree**

axioms: ∀ **tree** a, b, c, **int** α, β, γ :
let $\langle t_1, \ldots, t_n \rangle$ and m = maxdist be defined as above **in**

$$\begin{aligned}
\mathsf{rebalance}(r^\rho) &= \mathsf{empty}^{\rho-1}(\mathsf{lwb}(r), \mathsf{upb}(r)) && \text{if } n = 0 \\
&= t_1^{m-2} && \text{if } n = 1 \\
&= \mathsf{cons}2^{\rho+m-1}(t_1, t_2) && \text{if } n = 2 \\
&= \mathsf{cons}3^{\rho+m-1}(t_1, t_2, t_3) && \text{if } n = 3 \\
&= \mathsf{cons}2^{\rho+m}(\mathsf{cons}2^0(t_1, t_2), \mathsf{cons}2^0(t_3, t_4)) && \text{if } n = 4 \\
&= \mathsf{cons}2^{\rho+m}(\mathsf{cons}3^0(t_1, t_2, t_3), \mathsf{cons}2^0(t_4, t_5)) && \text{if } n = 5 \\
&= \mathsf{cons}2^{\rho+m}(\mathsf{cons}3^0(t_1, t_2, t_3), \mathsf{cons}3^0(t_4, t_5, t_6)) && \text{if } n = 6 \\
&= \mathsf{cons}3^{\rho+m}(\mathsf{cons}3^0(t_1, t_2, t_3), \mathsf{cons}2^0(t_4, t_5), \mathsf{cons}2^0(t_6, t_7)) && \text{if } n = 7
\end{aligned}$$

required: ∀ **tree** t :
$\mathsf{dist}(\mathsf{son}_i(t)) \neq 0 \vee \mathsf{son}_i(t)$ **is empty** **for** rebalance(t)

end of type

The peculiar forms of the first three cases take into account that empty subtrees may occur side by side with disturbances. Note that the operation **rebalance** is the identity under our equivalence relation for trees.

The transformation of section 3.2 now produces the nondeterministic specification

type EXTENDED TREE2 **declares** Rebalance :

opns: Rebalance : **tree** → **tree**

axioms: $\forall$ **tree t**, $\exists$ **tree s** :

$$\text{Rebalance}(t) = t[s \Leftarrow \underline{\text{rebalance}}(s)] \wedge s \preceq t \wedge \big(\text{dist}(\text{son}_i(s)) \neq 0 \vee \text{son}_i(s) \text{ is empty}\big) \quad \textbf{if} \quad \exists \text{ tree } s : s \preceq t \wedge \big(\text{dist}(\text{son}_i(s)) \neq 0 \vee \text{son}_i(s) \text{ is empty}\big)$$
$$= t \quad \textbf{otherwise}$$

end of type

This type is nondeterministic: The restrictions do not determine which degeneration is treated if there exist several ones in the given tree.

5.4. Demons vs. Individual Responsibility

We somehow have to implement the nondeterministic operation **Rebalance** of the previous section. The goal can be described using the terminology of a "demon". When there is some degeneration in the tree a "demon is triggered" which miraculously repairs it. If this repair causes another degeneration the demon is triggered anew, etc.

One solution is to have (one or several) special processes that act as demons. To keep them from doing exhaustive searches we have to provide these processes with a list of (potentially) degenerated nodes. Since the operations **add** and **delete** also affect this list it is a shared variable (which we could model by associating it to the tree).

Another solution is to charge the process causing the degeneration with its repair. In this case the following property is particularly pleasant: Assume that we restrict all disturbances to the three values $-1, 0, +1$. Then we get from the property (e.g. in case 4) $\beta > \alpha$ the three possibilities $(\alpha, \beta) = (0, +1)$, $(\alpha, \beta) = (-1, 0)$, $(\alpha, \beta) = (-1, +1)$. In the first and second case we get the new disturbances $\alpha' = \zeta' = 0$, in the third one $\alpha' = \zeta' = -1 = \zeta = \alpha$. The disturbance of r becomes either $\rho' = \rho$ (in case 2) or $\rho' = \rho + 1$. The latter case enforces the additional restriction $\rho \leq 0$ for the applicability of the operation **rebalance**. Analogously, in the cases $n = 0, 1, 2$ we have to avoid disturbances < -1. This may require that we shift a negative disturbance up to the father r before we eliminate an empty tree. Also, to avoid deadlocks some of the cases 2 – 6 need a splitting into subcases. The pleasant effect of this setting is that the number of nonzero disturbances never increases through a rebalancing. Furthermore, an addition or deletion effects at most one disturbance. Hence, the number of nonzero disturbances is never greater than the number of processes. (Of course, the root of the overall tree has to be treated individually, since here the disturbances accumulate.)

Applying the sequence of transformations described in section 3 we arrive at a type "multiple 2-3-tree" specifying the operations $\underline{\text{rebalance}}_\mu(t)$ which repair the current tree μ. Since we can localize the degenerated tree from our knowledge of its lower and upper bound we could even generate the necessary tree traversal with our transformations. But this is not reasonable, for the search would start at the root every time anew. Yet, we know that the only new degeneration possibly arising

from the repair is localized at the father. In appendix A we briefly discuss suitable variants of the transformations for introducing tree traversal. In our application we get

axiom $\forall$ **ttree t :**

$$F(t) = \begin{cases} F(up(subst(t, \underline{rebalance}(current(t))))) & \textbf{if } R(son_i(current(t))) \\ = t & \textbf{otherwise} \end{cases}$$

where $R(s) \equiv dist(s) \neq 0 \vee s$ **is empty**

Finally, to show deadlock-freedom we have to consider the possible violations of those restrictions that refer to other processes. Such restrictions only occur in the operation **subst** of the type MULTIPLE TREE$^+$. Therefore we have to examine all cases of the operation **rebalance** with respect to these requirements of **subst**. The only critical nodes are the "higher" sons of r (see the figures 2 – 6), they must not be the current trees of other processes. For downward processes – **Add** and **Delete** – there is no problem: they are not restricted in any way and thus will eventually move on. An upward process – **repair** – does not conflict with r either and therefore will eventually move up to r (where an arbitrary number of processes are permitted). In other words, the type EXTENDED MULTIPLE TREE has the invariant property that that there is at least one μ for which no requirement is violated.

6. CONCLUSION

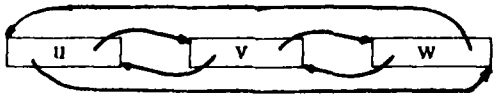
This paper exemplifies the typical approach that is taken for the derivation of new transformation rules. One considers the solution of a special problem and analyzes it carefully. The goal of this analysis is to extract the essential aspects of the solution, filtering out all unimportant details. The second step is to put this extracted knowledge into a generalized schematic form that is mechanically applicable. The final step has been omitted here, viz. the design of a convenient and precise formalism into which the technical details can be cast.

Such a schematic and formalized development has an important side-effect: Though sometimes looking quite intrinsic and complex the various applicability conditions of the transformations as well as certain enrichments of types indicate what is actually needed to perform a safe development. These proofs are necessary to guarantee correctness – in whatever technique or notation they are carried out. Even if there remains a lot of work to be done on the user's side, the transformations tell him at least which problems he has to attack.

The future research on this subject has to address two points (and probably in that order): First, it is necessary to study other special type transformations. For example, how do the rules in this paper work for systems of mutually recursive type declarations? What are the possibilities if we allow infinite objects in the style of Dana Scott? Consider the system of equations

$$u = cons(v, w), \quad v = cons(w, u), \quad w = cons(u, v).$$

The fixpoint of these equations describes the doubly linked list



The transformations for pointer implementations apply here in a straightforward manner. Finally, how do other types that are not recursive declarations fit into the approach? Note for instance that

the type DICTIONARY is "almost" recursively describable and so are sequences, queues, stacks, etc.

The second issue is to back the transformations more rigorously from a semantical point of view. The most severe problem arises here in connection with parallelism. In particular the possible connections between abstract data types and concurrency need further exploration. Types such as MULTIPLE TREE allow a specification of the overall behavior of the ensemble of processes according to the multiprogramming assumption. In this respect the universally quantified equations and restrictions show a close relationship to the $\Box$-operator of temporal logic and the existential quantifiers correspond to the eventuality operator $\Diamond$. When these connections are better understood, one can search for more general transformations leading from such global specifications to the individual protocols of the processes under consideration.

Acknowledgement I am grateful to Zohar Manna for providing the pleasant environment in which this research was carried out. Pierre Wolper was a stimulating partner in numerous discussions on the subject.

APPENDIX

This appendix addresses a few points that were too lengthy for being included into the main part of the paper. Again the discussion will not be overly formal.

APPENDIX A: NONDETERMINISM IN ABSTRACT DATA TYPES

Nondeterminism as it is used in some of today's more advanced programming languages is characterized by its completely "erratic" nature. This means in the first place that one cannot predict the result of a computation. But in addition it may even happen that two executions of the *same* expression under the *same* circumstances yield different results. Within the classical framework of abstract data types we can only mirror the former aspect, the latter requires an extension to relational theories. (Though being aware of this gap we will still speak of nondeterminism here. After all, we only want to apply abstract data types and do not model the semantics of programming languages.) This limitation even has its advantages: In our restricted nondeterminism we are still allowed to use an equation such as **x** = **e** to substitute **x** by the expression **e** without problems.

The general case of the transformation in section 3.2 is characterized as follows: We are given an operation **f** together with a restriction **R** of its domain and we want to substitute some admissible subtree **s** of a given tree **t** by **f(s)**. Let us call this substitution **F(t)**. Since we must not apply **F** unless there actually exists a subtree **s** with **R(s)**, the axiom reads

$$\forall\,\textbf{tree t}: \left(\exists\,\textbf{tree s}: \mathbf{s} \preceq \mathbf{t} \wedge \mathsf{R}(\mathbf{s})\right) \Rightarrow \left(\exists\,\textbf{tree s}: \mathbf{s} \preceq \mathbf{t} \wedge \mathsf{R}(\mathbf{s}) \wedge \mathsf{F}(\mathbf{t}) = \mathbf{t}[\mathbf{s} \Leftarrow \mathbf{f}(\mathbf{s})]\right)$$

According to our use of restrictions we can split this up into an axiom and a restriction:

axiom $\forall\,\textbf{tree t}: \exists\,\textbf{tree s}: \mathbf{s} \preceq \mathbf{t} \wedge \mathsf{R}(\mathbf{s}) \wedge \mathsf{F}(\mathbf{t}) = \mathbf{t}[\mathbf{s} \Leftarrow \mathbf{f}(\mathbf{s})]$

required $\forall\,\textbf{tree t}: \exists\,\textbf{tree s}: \mathbf{s} \preceq \mathbf{t} \wedge \mathsf{R}(\mathbf{s})$ **for** $\mathsf{F}(\mathbf{t})$

In the **add**-example of section 3.2 the requirement is **R(s) = s is leaf**. Since there always exists a subtree **s** of **t** fulfilling this requiremet the restriction can be omitted.

As a second example consider the definition

delete(leaf(y), y) = empty with the restriction **s = leaf(y)** **for** **delete(s, y)**

Here the restriction determines the application point uniquely and therefore we get the simplified version

axiom $\forall\,\textbf{tree t, elem y}: \mathsf{Delete}(\mathbf{t}, \mathbf{y}) = \mathbf{t}[\mathbf{leaf(y)} \Leftarrow \mathbf{empty}]$

required $\forall\,\textbf{tree t, elem y}: \mathbf{leaf(y)} \preceq \mathbf{t}$ **for** $\mathsf{Delete}(\mathbf{t}, \mathbf{y})$

Remark: This example shows that it will be useful to have the transformation in two flavors. In one which we have used here - the new operation **F** is undefined if there does not exist a suitable subtree, in the other form **F** is made into the identity in that case.

There is a second major instance of nondeterinism: Certain operations that are applicable at several points in the structure shall be executed "as often as possible" but in arbitrary order (the "demon" of section 5). The order of applications may decisively influence the result, in particular when the operations may generate new application points. In this case we modify the previous rule such that **f** is applied repeatedly. The axiom then reads

axiom $\forall$ **tree t** : $\exists$ **tree s** :

$s \preceq t \wedge R(s) \wedge F(t) = F(t[s \Leftarrow f(s)])$ **if** $\exists$ **tree s** : $s \preceq t \wedge R(s)$

$F(t) = t$ **otherwise**

Note that F now is, of course, the identity when there is no application point.

The rebalancing example in section 5 shows the need for a generalized variant of this rule. There a process should not treat all degenerations but only those that were reachable for it, i.e. those on the path from the current node back to the root. This leads to a *modification* of the transformation rule: Assume that the restriction R of the operation f is of the form

required $\forall$ t : $R(son_i(t))$ **for** $f(t)$

Assume further that f may establish the truth of R on its result, i.e. $R(son_i(t))$ may cause $R(f(t))$. Then we simply have to move upwards. In this case F is defined by

axiom $\forall$ **ttree t** :

$F(t) = F(up(subst(t, f(current(t))))$ **if** $R(son_i(current(t)))$

$= t$ **otherwise**

APPENDIX B: MORE ON POINTER IMPLEMENTATIONS

This part of the appendix describes the technical derivation of the pointer implementation of section 3.5 in greater detail and also gives a brief accont on the underlying model that justifies the transformation. The transformation probably is intuitively clear to anyone who has ever done programming with pointer structures. But if we want to do better than referring to the reader's intuition we have to go into some elementary mathematics.

A finite mapping can be represented by a finite set of pairs. If we use pairs of the form (i, x) with $i \in$ **index** and $x \in$ **contents** we have a (terminal) model of the type STORE. Trees are special graphs, which in turn can be represented by mappings. In this setting the operation **cons(a, b)** is implemented by

$$A \uplus B \uplus \{(n_1, (r_A, r_B))\}$$

where the mappings A and B represent the trees **a** and **b**, $\uplus$ is the "disjoint union" of mappings (all indices of B are consistently renamed), r_A, r_B are the roots of A and B and n_1 is a "new" index. Consequently, the function **substitute(A, B, C)** yields a new mapping A′ that is derived from A by replacing the submapping B by C (obeying the possibly required renaming of indices).

An efficiency measure suggests itself. Assume that A′ is as similar to A as possible (i.e. the indices are accordingly renamed). Then the differences $D = A - (A \cap A')$ and $D' = A' - (A \cap A')$ tell us how many pairs in A are altered or added to achieve A′. In the following constructions the sizes of D and D′ depend directly on the length of the given expressions (and not on the sizes of A, B and C). Therefore the cost is constant. Consider the example of section 3.5:

$A' =$ **substitute(A, S, U)**

where $S =$ **cons(cons(A, B), C)**

$= A \uplus B \uplus C \uplus \{(n_1, (r_A, r_B))\} \uplus \{(n_0, (n_1, r_C))\}$

$U =$ **cons(cons(A, L), cons(B, C))**

$= A \uplus L \uplus B \uplus C \uplus \{(n_2, (r_A, r_L))\} \uplus \{(n_1, (r_B, r_C))\} \uplus \{(n_0, (n_1, n_2))\}$

$L =$ **{(n3, leaf(x))}**

The only elements not in $A \cap A'$ are

$\{(n_1, (r_A, r_B))\}, \{(n_0, (n_1, r_C))\}$
$\{(n_3, \mathsf{leaf}(x))\}, \{(n_2, (r_A, r_L))\}, \{(n_1, (r_B, r_C))\}, \{(n_0, (n_1, r_2))\}$

Consequently, it suffices to exchange the right-hand sides of n_0 and n_1 and to add the pairs for n_2 and n_3.

How is this mechanized in a purely syntactic fashion? The following algorithm derives the necessary information by inspecting the given expression. Since the tree **s** to be substituted frequently is not given in the **cons**(...) form we reformulate the above expression:

$$t' = \mathsf{substitute}(t\ ,\ s\ ,\ \mathsf{cons}(\ \mathsf{cons}(\mathsf{son1}(\mathsf{son1}(s)), \mathsf{leaf}(x))\ ,\ \mathsf{cons}(\mathsf{son2}(\mathsf{son1}(s)), \mathsf{son2}(s))\ )$$

By using auxiliary identifiers this reads

$$\begin{array}{ll} i_0 = s & j_0 = \mathsf{cons}(j_1, j_3) \\ i_1 = \mathsf{son1}(i_0) = \mathsf{son1}(s) & j_1 = \mathsf{cons}(i_3, j_2) \\ i_2 = \mathsf{son2}(i_0) = \mathsf{son2}(s) & j_2 = \mathsf{leaf}(x) \\ i_3 = \mathsf{son1}(i_1) = \mathsf{son1}(\mathsf{son1}(s)) & j_3 = \mathsf{cons}(i_4, i_2) \\ i_4 = \mathsf{son2}(i_1) = \mathsf{son2}(\mathsf{son1}(s)) & \end{array}$$

The above detailization follows the rules: Every subterm of the kind $\mathsf{son}_i(\ldots)$ gets its own identifier (corresponding to its index in the original mapping). Every **cons**(...) and **leaf**(...) gets an index, too. All identifiers i_k of the left-hand side that do not occur on the right-hand side — in our case i_0 and i_1 — can be replaced by the corresponding j_k. All j_n not covered in this way stand for new indices — in our case j_2 and j_3. Translating this into a sequence of **alter**-expressions of the type STORE leads to the program of section 3.5. The mechanical nature of this derivation is obvious. The applicability condition is that the expression for the new tree **u** can be expressed in terms of the old tree **s** and **leaf**-operations.

There are optimizations possible. For example, if **son2**(**son1**(**s**)) would not occur in the expression for the new tree but several leaves are added then the above algorithm assigns new indices (i.e. storage locations) to many of the leaves without noticing that all indices belonging to the subtree **son2**(**son1**(**s**)) are available. To minimize garbage collection more elaborate variants are therefore needed.

Note that all operations in the paper — addition, deletion, rebalancing — meet the above requirements.

APPENDIX C: ABSTRACT DATA TYPES AND LOCKING

In section 3 we have adopted a purely formalistic view of transformations for locks. Now we will be more precise and also study to what degree abstract data types can be used and where the problems arise.

To begin with, the multiprogramming-assumption explains parallelism in terms of sequential interleaving: If there are two processes P_1 and P_2 issuing sequences of "atomic" operations $\sigma = \sigma_1 \cdots \sigma_k$, $\tau = \tau_1 \cdots \tau_m$ then the semantics of their concurrent operation is explained by the set of all possible mergings of the sequences σ and τ. This set is called **shuffle** in [12].

Remark: Actually, the two processes generally influence each other. Therefore one has to model their individual behaviors by some kind of formal trees and the shuffle has to be generalized such that it merges trees into sequences.

Now assume that we started from a program where the operations σ and τ were taken to be atomic. The shuffle of this program therefore is the set

$$\{\sigma\tau, \tau\sigma\}$$

If σ and τ are then broken up into sequences of more elementary operations we must guarantee that the shuffle of the new program is equivalent to the set

$$\{\sigma_1 \cdots \sigma_k \tau_1 \cdots \tau_m \,,\, \tau_1 \cdots \tau_m \sigma_1 \cdots \sigma_k\}.$$

On the other hand, the size of the shuffle indicates the degree of concurrency. One safe way of increasing that size is enabled if the operation τ_1 is exchangeable with σ, i.e. $\sigma\tau_1 = \tau_1\sigma$. Then we get the equivalent shuffle

$$\{\sigma_1 \cdots \sigma_k \tau_1 \cdots \tau_m \,,\, \tau_1 \sigma_1 \cdots \sigma_k \tau_2 \cdots \tau_m \,,\, \tau_1 \cdots \tau_m \sigma_1 \cdots \sigma_k\}$$

Analogously for τ_m, σ_1, σ_k and then for τ_2, etc. In other words, we have to identify within the sequence σ the largest subsequence that does not allow the above interchanging. This leads to

$$\sigma = \sigma_1 \cdots \sigma_{i-1} [\sigma_i \cdots \sigma_j] \sigma_{j+1} \cdots \sigma_k$$

where the sequence in parantheses is a critical region that has to be viewed as being atomic.

The above interchanging is in particular enabled for those operations σ_i that are the identity in the given semantic interpretation. In the sequel we will treat this in the context of abstract data types. (For shortening the writing-down we will restrict ourselves to one kind of locks.)

type LOCK (**type** M) **declares** lock, unlock :

enrich M **by**

opns:
lock : **name** × **m** → **m**
unlock : **name** × **m** → **m**
locked : **name** × **m** → **bool**

axioms: $\forall$ m x, **name** μ, π :

$\mathsf{locked}_\mu(\mathsf{lock}_\mu(x))$	$= \mathsf{true}$	
$\mathsf{locked}_\mu(\mathsf{lock}_\pi(x))$	$= \mathsf{locked}_\mu(x)$	**if** $\mu \neq \pi$
$\mathsf{locked}_\mu(\mathsf{unlock}_\mu(x))$	$= \mathsf{false}$	
$\mathsf{locked}_\mu(\mathsf{unlock}_\pi(x))$	$= \mathsf{locked}_\mu(x)$	**if** $\mu \neq \pi$
$\mathsf{locked}_\mu(f(x))$	$= \mathsf{locked}_\mu(x)$	for all operations f of M
$\mathsf{locked}_\mu(c)$	$= \mathsf{false}$	for all constants c of M
$f(\mathsf{lock}_\mu(x))$	$= f(x)$	for all operations f of M
$f(\mathsf{unlock}_\mu(x))$	$= f(x)$	for all operations f of M

required: $\forall$ m x, **name** μ, π :

$\neg\mathsf{locked}_\mu(x)$ **for** $\mathsf{lock}_\pi(x)$ **if** $\pi \neq \mu$

end of type

This type specifies the essentials of locks. But it has its problems: A minor one is that the last four axioms are generic. They just express the facts that the basic operations of the type M do not alter locks and that, vice versa, locks act like the identity on all operations. This is exactly what we need for the sequences occurring in the shuffle. But the severe problem is that these last equations also can be used to eliminate any lock from any program before the shuffle is built. Thus we are confronted with the situation that we need equations to specify the meaning of certain operations but that we never must apply these equations to a program. In other words, types not only need hidden functions but also hidden axioms.

If we accept the above extension of the type mechanism we have a valuable tool that makes the "underlining" of section 3.8 more accurate. For particularly chosen functions such as **up** and **down** we may add the equations

$\mathsf{down}(\mathsf{lock}(t)) = \mathsf{lock}(\mathsf{unlock}(\mathsf{down}(\mathsf{lock}(t))))$
$\mathsf{unlock}(\mathsf{down}(t)) = \mathsf{unlock}(\mathsf{down}(\mathsf{lock}(\mathsf{unlock}(t))))$

Now we can do the same development that we have done for the procedure **Add*** in section 3.8, but in the applicative style of abstract data types. The operation **Add'** with the initial definition

$\mathsf{Add}'(t, x) = \mathsf{unlock}(\mathsf{Add}(\mathsf{lock}(t), x))$

then gets the derived specification

$\mathsf{Add}'(t, x) = \mathsf{Add}'(\mathsf{unlock}(\mathsf{down}(\mathsf{lock}(t))), x)$ **if** $\mathsf{lock}(t)$ **is cons**
$= \mathsf{unlock}(\mathsf{add}(\mathsf{lock}(t), x))$ **otherwise**

The advantage of this more formal treatment is that we can state more precisely where read- and write-locks will occur, for example

$\mathsf{down}(\mathsf{w\text{-}lock}(t)) = \mathsf{w\text{-}lock}(\mathsf{r\text{-}unlock}(\mathsf{down}(\mathsf{r\text{-}lock}(t))))$
$\mathsf{rebalance}(\mathsf{w\text{-}lock}(t)) = \mathsf{w\text{-}lock}(\mathsf{w\text{-}unlock}(\mathsf{rebalance}(\mathsf{w\text{-}lock}(t))))$

The intrinsic problem remains, of course, to determine which operations allow the addition of such equations. For the operations such as **up** and **down** this is done as a byproduct of the earlier transformation rules. For other operations such as rebalance the user has to provide a proof e.g. that the operation is the identity in a suitably chosen equivalence relation. In addition, it must be clarified whether a read- or a write-lock is needed. The remaining details then are of a purely mechanic nature such that we could leave them to a transformation rule INTRODUCE-LOCKS.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, J. D. ULLMAN: The Design and Analysis of Computer Algorithms. Reading, Mass.: Addison-Wesley, 1974.

[2] F. L. BAUER, H. WÖSSNER: Algorithmic Language and Program Development. Berlin-Heidelberg-New York: Springer (to appear).

[3] F. L. BAUER ET AL.: Report on the Wide Spectrum Language CIP-L. *Technische Universität München, Institut für Informatik 1981.*

[4] R. BAYER, M. SCHKOLNICK: Concurrency of Operations on B-Trees. *Acta Informatica* **9**,1-21 (1979).

[5] M. BROY, B. MÖLLER, P. PEPPER, M. WIRSING: A Model-Independent Approach to Implementations of Abstract Data Types. In: A. SALWICKI (ED.): Proc. Symp. on Algorithmic Logic and the Programming Language LOGAN. *Lecture Notes in Computer Science,* Berlin-Heidelberg-New York: Springer 1981

[6] M. BROY, P. PEPPER: Program Development as a Formal Activity. *IEEE Transactions on Software Engineering, SE-7*:1, 14-22 (1980).

[7] R. M. BURSTALL, J. A. GOGUEN: The Semantics of CLEAR, a Specification Language. Proc. 1979 Copenhagen Winter School on Abstract Software Specification. *Lecture Notes in Computer Science* **86**, 1980.

[8] J. A. GOGUEN, J. MESSEGUER: OBJ-1, A Study in Executable Algebraic Formal Specification. SRI International, Menlo Park, July 1981.

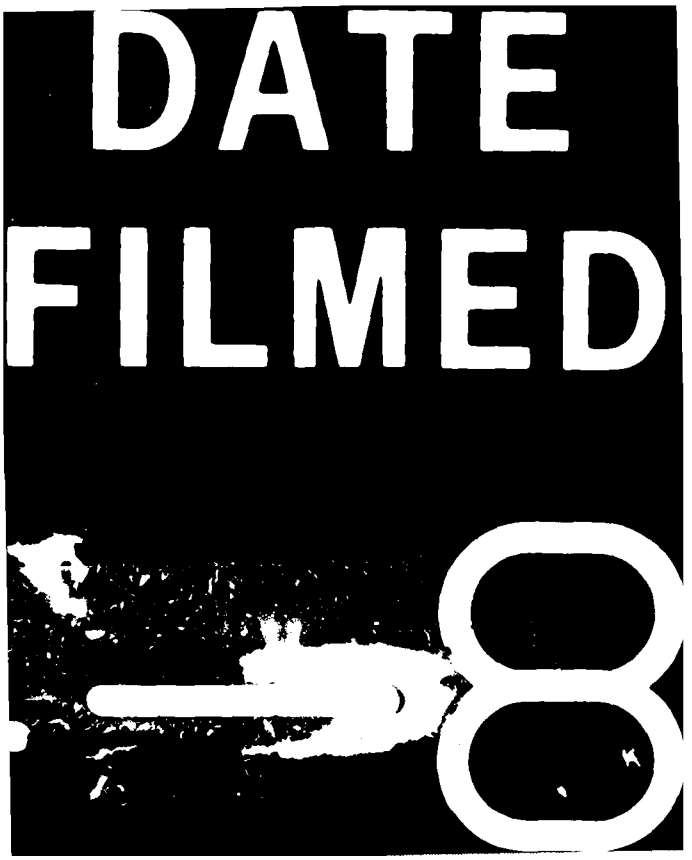
[9] C. A. R. HOARE: Communicating Sequential Processes. *Comm. ACM* **21**:8, 666-677 (1978).

[10] K. JENSEN, N. WIRTH: PASCAL User Manual and Report. New York-Heidelberg-Berlin: Springer, 1974.

[11] Y. S. KWONG, D. WOOD: *Concurrent Operations in Large Ordered Indexes.* In: B. ROBINET (ed.): Proc. 4th Int. Symposium on Programming, Paris, 1980. *Lecture Notes in Computer Science* **83**,202-222 (1980)

[12] S. GINSBURG: The Mathematical Theory of Context-Free Languages. Mc Graw Hill (1966).

[13] M. WIRSING, P. PEPPER, H. PARTSCH, W. DOSCH, M. BROY: On Hierarchies of Abstract Data Types. *Technische Universität München, Institut für Informatik TUM-I8007, 1980.*


DATE
FILMED
8