2

Calculating With Pointer Structures

Bernhard Möller
Institut für Informatik, Universität Augsburg
Universitätsstr. 14, D-86135 Augsburg, Germany.
email: moeller@uni-augsburg.de

Abstract

Based on the algebra of relations and maps we present some techniques for safe manipulation of pointer structures, with a special emphasis on tree-like structures. We investigate sufficient criteria for preservation of substructures under selective updating. The approach is illustrated with some simple examples.

Keywords

Pointer structures, sharing, destructive updating, program transformation, relational calculus

1 INTRODUCTION

Although pointer algorithms are very error-prone they lie at the very heart of many implementations. Yet they have received surprisingly little attention in work on formal derivation and verification of programs. If they are treated, mostly formulas from predicate logic are used, which tend, however, to be very complex and unwieldy. First attempts at a more algebraic approach were presented in our previous previous work (Berger et al. 1991, Möller 1991–1993). However, the calculation steps there were still comparatively small, and even simple algorithms had somewhat lengthy derivations.

The present paper generalizes that approach and develops additional techniques for calculating safe pointer implementations of operations that are specified at an abstract functional level. As usual, pointer level and abstract level are related via an abstraction function. We investigate the particular class of *reasonable* abstraction functions that depend only on the reachable part of the overall store. This allows reducing many questions about changes in a data structure to an analysis of the changes in reachability. Hence the treatment becomes independent of particular data structures such as lists or trees. As a result, we are able to prove the relevant properties once and for all and to condense the derivations considerably, which brings us closer to the goal that simple algorithms should have simple derivations.

2 RELATIONAL NOTATION

Our prominent mathematical tool will be binary relations by which we model the directed graph underlying a pointer structure and describe accessibility and sharing. Given a set X we denote its power set by $\wp(X)$. Now the set of all *binary relations* between sets M and N is $M \leftrightarrow N \stackrel{\text{def}}{=} \wp(M \times N)$. We use the notations $R : M \leftrightarrow N$ and $R \in M \leftrightarrow N$ synonymously.

By $\mathsf{dom}R$ and $\mathsf{ran}R$ we denote domain and range of $R : M \leftrightarrow N$. The *converse* $R^\smile : N \leftrightarrow M$ of R is given by $R^\smile \stackrel{\text{def}}{=} \{(y,x) : (x,y) \in R\}$. The *image* of set $L \subseteq M$ under R is $R(L) \stackrel{\text{def}}{=} \{y : \exists\, x \in L : (x,y) \in R\}$.

Particularly for analyzing the reachable part of a pointer structure we shall use the *domain restriction* of R to a subset $L \subseteq M$ given by $L \lhd R \stackrel{\text{def}}{=} R \cap L \times N$. Dually, the *range restriction* of R to a subset $L \subseteq N$ is $R \rhd L \stackrel{\text{def}}{=} R \cap M \times L$. Useful properties are

$$\begin{array}{rclcrcl} \mathsf{dom}(L \lhd R) & = & L \cap \mathsf{dom}R\,, & & \mathsf{dom}(R \rhd L) & = & R^\smile(L) \\ \mathsf{ran}(L \lhd R) & = & R(L)\,, & & \mathsf{ran}(R \rhd L) & = & L \cap \mathsf{ran}R\,. \end{array} \tag{1}$$

This implies $L \subseteq \mathsf{dom}R \Rightarrow \mathsf{dom}(L \lhd R) = L$.

The *composition* $R;S : M \leftrightarrow P$ of two relations $R : M \leftrightarrow N$ and $S : N \leftrightarrow P$ is defined as $R;S \stackrel{\text{def}}{=} \{(x,z) : \exists\, y \in N : (x,y) \in R \wedge (y,z) \in S\}$. Left and right neutral elements for R w.r.t. this operation are provided by I_M and I_N, where for a set P one defines the *identity relation* $I_P : P \leftrightarrow P$ by $I_P \stackrel{\text{def}}{=} \{(x,x) : x \in P\}$. The index $_P$ will be omitted when P is clear from the context.

As usual, R^+ and R^* are the transitive and the reflexive transitive closures of relation R.

Relation $R \subseteq M \times N$ is called a *(partial) map* if each element of M is in relation with at most one element from N, i.e., if $(x,y) \in R \wedge (x,z) \in R \Rightarrow y = z$. This can be expressed more concisely as $R^\smile; R \subseteq I_N$. We write $R : M \rightsquigarrow N$ to indicate that R is a map.

For further notions and laws concerning relations consider e.g. Schmidt, Ströhlein (1993).

3 A MODEL OF POINTER STRUCTURES

3.1 Stores and Pointer Structures

A pointer structure consists of a set of records connected by pointers. Let $\mathcal{A}$ be a set of *records* (represented, say, by their initial addresses). We assume a distinguished element $\diamond \in \mathcal{A}$ which plays the role of nil in Pascal or `NULL` in C, i.e., serves as a terminal pseudo-node for the underlying graph. The elements of $\mathcal{A}\backslash\{\diamond\}$ are called *proper* records. Let, moreover, $(\mathcal{N}_j)_{j \in J}$ be a family of sets of *node values*, such as integers or Booleans.

Then a *record scheme* consists of a non-empty set K of *selectors* each with a type $\mathcal{A} \to \mathcal{A}$ or $\mathcal{A} \to \mathcal{N}_j$ for some $j \in J$. Given such a record scheme, a *store* is a family $S = (S_k)_{k \in K}$ of partial maps such that

1. $S_k : \mathcal{A} \rightsquigarrow \mathcal{A}$ if k has type $\mathcal{A} \to \mathcal{A}$,
2. $S_k : \mathcal{A} \rightsquigarrow \mathcal{N}_j$ if k has type $\mathcal{A} \to \mathcal{N}_j$,
3. $\diamond \notin \mathit{recs}(S)$,

where $\mathit{recs}(S) \stackrel{\text{def}}{=} \bigcup_{k \in K} \mathrm{dom} S_k$ is the set of records allocated in S.

A store may be viewed as a labeled directed graph: the selectors are the arc labels and S_k is the set of arcs labeled by k. We record these sets separately to be able to model updating along single selectors adequately. The requirement that the S_k be maps serves to model the uniqueness of selection in records. By the third requirement, in a store, $\diamond$ is not related to anything and hence cannot be "dereferenced". This implies that there can be no $\diamond$ record in the "interior" of a pointer structure; if present, $\diamond$ terminates the structure at that point. The relational operations are extended componentwise to stores.

As an example of a record scheme consider a single set $\mathcal{N}$ of node values and three selectors l, v, r with types $l : \mathcal{A} \to \mathcal{A}$, $v : \mathcal{A} \to \mathcal{N}$ and $r : \mathcal{A} \to \mathcal{A}$. Then a binary tree store BT consists of three partial maps $BT_l : \mathcal{A} \rightsquigarrow \mathcal{A}$, $BT_v : \mathcal{A} \rightsquigarrow \mathcal{N}$ and $BT_r : \mathcal{A} \rightsquigarrow \mathcal{A}$, where BT_l and BT_r give the roots of the left and right subtree, if any, of a node, whereas BT_v returns the node value.

Frequently we want to abstract from the node values of the records and consider just their interrelationship through the pointers, since this is the only source of problems in pointer algorithms. Given a store $S = (S_k)_{k \in K}$, this is modeled by the binary *access relation* $[S] \subseteq \mathcal{A} \times \mathcal{A}$ given by

$$[S] \stackrel{\text{def}}{=} \bigcup_{k \in J} S_k \,,$$

where $J \subseteq K$ is the set of all selectors k of type $\mathcal{A} \to \mathcal{A}$. In the graph view, this operation "forgets" the arc labels. For instance, the access relation for a binary tree store BT is $[BT] \stackrel{\text{def}}{=} BT_l \cup BT_r$.

Store S is *closed* if $\mathrm{ran}[S] \subseteq \mathit{recs}(S) \cup \{\diamond\}$. This means that there are no "dangling references" to addresses not allocated in S. Equivalently, S is closed if $\mathrm{ran}[S] \backslash \mathit{recs}(S) \subseteq \{\diamond\}$. One may wonder why closedness was not built into our definition of store. This is because non-closed stores will frequently be used in constructing larger stores from smaller ones.

Let now $\mathcal{S}$ denote the set of all stores for a given record scheme. The set of *entries* to pointer structures is $\mathcal{A}^+$, the set of all non-empty finite lists of elements of $\mathcal{A}$. We choose lists rather than sets or bags of entries, since in pointer algorithms both order and multiplicity of entries may be relevant.

Now a *pointer structure* is an element of $\mathcal{P} \stackrel{\text{def}}{=} \mathcal{A}^+ \times \mathcal{S}$. For convenience we introduce the functions

$$\begin{aligned} \mathit{ptr}(s, S) &\stackrel{\text{def}}{=} s \,, \\ \mathit{sto}(s, S) &\stackrel{\text{def}}{=} S \,, \\ \mathit{recs}(s, S) &\stackrel{\text{def}}{=} \mathit{recs}(S) \,. \end{aligned}$$

In denoting lists of entries we separate the elements by commas. So a pointer structure will be written in the form $x_1, \ldots, x_n, S$ with entries $x_1, \ldots, x_n$ and store S.

As an example, we define an operation for the allocation of new records. Let $\mathcal{V} \stackrel{\text{def}}{=} \prod_{k \in K} \mathcal{V}_k$ be the set of possible records, where $\mathcal{V}_k = \mathcal{N}_j$ if selector k has type $\mathcal{A} \to \mathcal{N}_j$ and $\mathcal{V}_k = \mathcal{A}$ otherwise. Then the relation $newrec \in (\mathcal{P} \times \mathcal{V}) \leftrightarrow \mathcal{P}$ is given by

$$(m, T) \in newrec(p, v) \stackrel{\text{def}}{\Leftrightarrow} m \notin recs(p) \cup \{\diamond\} \wedge T = sto(p) \cup N \ ,$$

where $N \stackrel{\text{def}}{=} \{(m, v_k)\}_{k \in K}$ is the (generally non-closed) store assigning the components of v to the newly allocated record address m. In our examples the set K of selectors will be finite. We shall assume a fixed linear order on it and write elements of $\mathcal{V}$ as ordered tuples.

3.2 Reachability and Sharing

In a pointer structure $(s, S) \in \mathcal{P}$ we can follow the pointers from the entries s to other records. This is modeled by the set

$$reach(s, S) \stackrel{\text{def}}{=} [S]^*(\text{set}\, s) \ ,$$

where $\text{set}\, s$ is the set of elements occurring in $s \in \mathcal{A}^+$.

Associated with this is the *reachability relation* $\vdash : \mathcal{P} \leftrightarrow \wp(\mathcal{E})$ given by

$$p \vdash L \stackrel{\text{def}}{\Leftrightarrow} reach(p) \cap \text{set}\, L \neq \emptyset \ ,$$

where $\text{set}\, L = \bigcup_{s \in L} \text{set}\, s$. So this relation holds iff some record in the entries in L is accessible from the entries of p. For singleton set L we will omit the set braces.

Moreover, we introduce a unary predicate *sharing* on $\mathcal{P}$ by setting

$$sharing(n_1, \ldots, n_k, S) \stackrel{\text{def}}{\Leftrightarrow} \bigvee_{i=1}^{k} \bigvee_{j=i+1}^{k} reach(n_i, S) \cap reach(n_j, S) \not\subseteq \{\diamond\} \ .$$

So a pointer structure shows sharing iff a proper record is reachable from two of its entries. Note that this predicate is independent of the order of the entries n_i but not of their multiplicity. So if a record occurs twice in a list of entries, there will be sharing, as expected.

Using this, we can, for instance, characterize pointer structures that are "independent of their surroundings" in the sense that the only pointers into their reachable part originate in that part itself. In pointer structure $(s, S) \in \mathcal{P}$ there is no sharing with records outside $reach(s, S)$ if $\forall\, n \in \mathcal{A} : sharing(n, s, S) \Rightarrow (s, S) \vdash n$.

The reachable set abstracts too much from the actual contents of the store in a pointer structure. Therefore we characterize additionally that part of store S that is reachable from s by the restriction

$$from(s, S) \stackrel{\text{def}}{=} (s,\ reach(s, S) \bowtie S) \ ,$$

i.e., the substructure in which only the contents of records reachable from the entries s are kept. The restriction is again taken componentwise, i.e., for all $k \in K$. Note that, for $L \subseteq \mathcal{A}$, we have $L \bowtie [S] = [L \bowtie S]$ and $[S] \bowtie L = [S \bowtie L]$. From the definitions it is immediate that

Lemma 1 *For any store S we have $reach(\diamond, S) = \{\diamond\}$ and $from(\diamond, S) = (\diamond, \emptyset)$.*

Moreover, we have the following properties of *reach* and *from* (see the Appendix for the proof):

Lemma 2 *1.* $from(s,S) = from(s,T) \Rightarrow reach(s,S) = reach(s,T)$.
2. $from(s, from(s,S)) = from(s,S)$ (Idempotence).
3. $(s,S) \nmid recs(T) \Rightarrow from(s, S \cup T) = from(s,S)$ (Localization I).

4 POINTER IMPLEMENTATIONS

4.1 Reasonable Abstraction Functions

We now consider implementations of abstract objects of some set $\mathcal{O}$ by pointer structures in such a way that each object is represented by a pointer structure $(n,S) \in \mathcal{P}$ with a single entry $n \in \mathcal{A}$. As usual (see e.g. Hoare (1972)), the relation between abstract and concrete levels is established by a partial abstraction function $F : \mathcal{A} \times \mathcal{P} \rightsquigarrow \mathcal{O}$ such that F is surjective. We do not need the more general concept of a linking invariant between abstract and concrete level. To allow representations of *tuples* of abstract objects, we extend F to a partial function $F : \mathcal{P} \rightsquigarrow \mathcal{O}^+$ on arbitrary pointer structures by setting

$$F(n_1 \cdots n_k, S) \stackrel{\text{def}}{=} F(n_1, S) \cdots F(n_k, S) .$$

As usual, F induces an equivalence relation $\sim$ on $\mathcal{P}$ by

$$p \sim q \stackrel{\text{def}}{\Leftrightarrow} F(p) = F(q) .$$

Note that the image set $F(p) = \emptyset$ for $p \notin \mathrm{dom} F$. So all pointer structures that do not represent an element of $\mathcal{O}$ are equivalent under $\sim$.

Since the pointer representation of an abstract object should be essentially determined by the entries to the structure, we say that an abstraction function is *reasonable* if for all $p, q \in \mathcal{P}$ we have

$$from(p) = from(q) \Rightarrow p \sim q .$$

This seemingly simple concept is the key idea that makes our treatment work uniformly and independently of particular data structures such as lists or trees. It allows us to reduce questions about the changes a selective updating effects to a much simpler analysis of the changes in reachability. In particular, we can use the well-established relational calculus for that analysis.

From the definitions and Lemma 2.2. we have immediately

Corollary 3 *For any reasonable abstraction function F, we have $F(p) = F(from(p))$, i.e., $p \sim from(p)$.*

A sufficient criterion for reasonableness (see the Appendix for the proof) is

Lemma 4 *If for all s, S, T we have $reach(s,S) \bowtie S = reach(s,S) \bowtie T \;\Rightarrow\; F(s,S) = F(s,T)$, then F is reasonable.*

4.2 Implementation of Operations

As usual (see e.g. Hoare (1972)), the general pattern for transferring operations from abstract level to pointer level is as follows.

Consider an operation of type $\mathcal{O}^n \rightsquigarrow \mathcal{B}$ that leads into a set $\mathcal{B}$ of "external" values such as integers or Booleans. We define an *implementation relation* $OPOI \in (\mathcal{P} \rightsquigarrow \mathcal{B}) \leftrightarrow (\mathcal{O}^n \rightsquigarrow \mathcal{B})$ by setting

$$pg \; OPOI \; g \stackrel{dsf}{\Leftrightarrow} pg = F ; g \; .$$

So the implementation pg has to mimic the specification g faithfully. Note the implicit use of the extended abstraction function F for the representation of tuples in $\mathcal{O}^n$.

For operations of type $\mathcal{O}^n \rightsquigarrow \mathcal{O}$ we are more liberal and allow the implementation to be non-deterministic, i.e., a relation rather than a map. This is reasonable, since different concrete objects may represent the same abstract object. A typical non-deterministic operation at the pointer level is *newrec* as defined in Section 3.1. Our notion of implementation will be parameterized by additional requirements on the implementing relation, such as preservation of certain aspects of the store. Such requirements are again formulated as relations between "old" and "new" pointer structures. Hence our *implementation relation* has type $POI : (\mathcal{P} \leftrightarrow \mathcal{P}) \rightarrow ((\mathcal{P} \leftrightarrow \mathcal{P}) \leftrightarrow (\mathcal{O}^n \rightsquigarrow \mathcal{O}))$ and is defined by

$$pf \; POI(req) \; f \stackrel{dsf}{\Leftrightarrow} pf ; F = F ; f \wedge pf \subseteq req \; .$$

Here req is the additional requirement, examples of which will be given later. The unconstrained relation $POI(ALL)$, where $ALL \stackrel{def}{=} \mathcal{P} \times \mathcal{P}$ is the universal relation, leaves complete freedom for realizing pf by copying or by re-use of relevant parts of its argument p. In particular, it does not exclude indirect side-effects on parts of p that point into the reachable part $from(p)$.

However, one frequently is interested in implementations that change as little as possible. We therefore want to give stronger specifications that guarantee that changes take place only in the relevant reachable part or outside the current store, i.e., on "new" records. To this end we define the set

$$noreach(p) \stackrel{def}{=} recs(p) \backslash reach(p) = recs(p) \backslash recs(from(p)) \; .$$

For closed S, the set $noreach(s,S)$ does not contain records outside the set $recs(S)$ of

allocated records. It is the set of all records that are not reachable from the entries s and hence should better be left alone by changes to S. Note that $n \in noreach(p) \Rightarrow p \not\models n$.

Now we can define two constraining relations $loc, pres \in \mathcal{P} \leftrightarrow \mathcal{P}$ by setting

$$p\ loc\ q \stackrel{\text{def}}{\Leftrightarrow} noreach(p) \bowtie sto(p) = noreach(p) \bowtie sto(q)\ ,$$
$$p\ pres\ q \stackrel{\text{def}}{\Leftrightarrow} noreach(p) \subseteq noreach(q)\ .$$

So *loc* requires that the part of the store that is unreachable in p is left untouched in q; by our definition this does, however, not exclude adding new records to the store.

However, *loc* also holds if in the "modified" structure q there are pointers into $noreach(p)$. So records that were unreachable in p may become reachable by the modification and hence accessible for subsequent modification. This potential source of problems for updates in q through $ptr(p)$ is excluded by postulating *pres*.

Now we can work with the strengthenings $POI(loc)$, $POI(pres)$ or even $POI(loc \cap pres)$ of $POI(ALL)$. They all still admit implementation by copying and by re-use.

4.3 Development Strategy

To calculate a pointer implementation pf of $f : \mathcal{O}^n \rightsquigarrow \mathcal{O}$, we start with the expression $f(F(p))$ and try to transform it by equational reasoning into an expression $F(E)$ such that $F(E) = f(F(p))$ and E does not contain F. Then we can define pf by setting $pf(p) \stackrel{\text{def}}{=} E$ and are sure that $pf\ POI(ALL)\ f$ holds. Design decisions are reflected by the particular choice of the applied equations and generally result in a reduction of nondeterminacy. For implementations of operations $g : \mathcal{O} \rightsquigarrow \mathcal{N}$ we may, more directly, start with the expression $g(F(p))$ and transform it in such a way that F is eliminated from it.

One design goal is to keep changes to a minimum. This has two aspects:

- preserve the entries to pointer structures, if possible;
- implement changes to single record components by selective updating, if possible.

We shall see these goals influence our example derivations. In particular, they will motivate the introduction of strengthened requirements as additional invariants.

5 OVERWRITING POINTER STRUCTURES

5.1 Overwriting, Selecting and Updating

An essential operation on stores is their selective updating. To describe this we define the operation of *overwriting* one relation with another one (see also e.g. Pepper, Möller (1991), Hehner (1993) and Spivey (1994) for the special case of maps). Given relations $R, S : M \leftrightarrow N$, we define the relation $R \mid S : M \leftrightarrow N$ (pronounced "R onto S") by

$$R \mid S \stackrel{\text{def}}{=} R \cup \overline{\text{dom}R} \bowtie S\ ,$$

where $\overline{\text{dom}R}$ is the complement of $\text{dom}R$. Hence $(x,y) \in R \mid S \Leftrightarrow (x,y) \in R \vee (x \notin \text{dom}R \wedge (x,y) \in S)$. Thus, $R \mid S$ results from S by changing the values associated with

the "arguments" from M according to the prescription of R (if any). For example, if S is a map then $\{(x,y)\} \mid S$ "updates" S to make y the value corresponding to x. One has

$$\mathsf{dom}(R \mid S) = \mathsf{dom}R \cup \mathsf{dom}S \,. \tag{2}$$

We use the convention that $\mid$ binds stronger than all set-theoretic operations. The set $M \leftrightarrow N$ forms a monoid under $\mid$ with $\emptyset$ as its neutral element. Moreover, the set of $M \rightsquigarrow N$ of maps is a submonoid of $M \leftrightarrow N$. Finally, for maps S, T we have the property

$$S \subseteq T \Rightarrow S \mid T = T \,. \qquad \text{(Annihilation)} \tag{3}$$

For further properties see Möller (1993b).

Consider now two stores S and T over the same record scheme. The *overwriting* $T \mid S$ is again defined componentwise. For pointer structure p and store S we set

$$S \mid p \stackrel{\text{def}}{=} (ptr(p), S \mid sto(p)) \,.$$

From (2) it follows that $recs(S \mid p) = recs(S) \cup recs(p)$. Moreover, Lemma 2.3. tells us that overwriting outside the substructure belonging to some entries does not change that substructure:

Corollary 5 $p \not\vdash recs(S) \Rightarrow from(S \mid p) = from(p)$ (Localization II).

In selective updating only one of the component maps of a store is overwritten properly. A store which models the update along selector k is $(x \overset{k}{\mapsto} y)$ given by

$$\begin{aligned} (x \overset{k}{\mapsto} y)_k &\stackrel{\text{def}}{=} \{(x,y)\} \,, \\ (x \overset{k}{\mapsto} y)_j &\stackrel{\text{def}}{=} \emptyset \qquad\qquad \text{for } j \neq k \,. \end{aligned}$$

To ease the notation and to keep with traditional programming languages, we introduce an operation $\_.\_ := \_ : \mathcal{P} \times K \times \mathcal{P} \to \mathcal{P}$ for selective updating. For selector k of type $\mathcal{A} \to \mathcal{A}$ and pointer structures (n,S) and (m,T) with $n, m \in \mathcal{A}$ we define

$$(n,S).k := (m,T) \stackrel{\text{def}}{=} (n, (n \overset{k}{\mapsto} m) \mid T) \,.$$

If k has type $\mathcal{A} \to \mathcal{N}_j$ and $x \in \mathcal{N}_j$ we set

$$(n,S).k := x \stackrel{\text{def}}{=} (n, (n \overset{k}{\mapsto} x) \mid S) \,.$$

Moreover, we define the selection operation $\_.\_ : \mathcal{P} \times K \rightsquigarrow (\mathcal{A} \cup \bigcup_{j \in K} \mathcal{N}_j)$ by

$$(n,S).k \stackrel{\text{def}}{=} (S_k(n), S)$$

if k has type $\mathcal{A} \to \mathcal{A}$ and $S_k(n)$ is defined

$$(n,S).k \stackrel{\text{def}}{=} S_k(n)$$

if k has type $\mathcal{A} \to \mathcal{N}_j$. Otherwise $(n, S).k$ is undefined. When using selections as arguments, undefinedness is assumed to propagate according to the strictness of relational semantics.

Selection and updating interact as expected:

Lemma 6 *1.* $ptr(p.k := q) = ptr(p)$.
2. $(p.k := p.k) = p$.
3. $j : \mathcal{A} \to \mathcal{N}_j \wedge j \neq k \Rightarrow (p.k := q).j = p.j$.

Proof. 1. is immediate from the definition.
2. We only treat the case of a selector $k : \mathcal{A} \to \mathcal{A}$.

$$
\begin{aligned}
& (n, S).k := (n, S).k \\
= \quad & \{\![\text{ definition of selection }]\!\} \\
& (n, S).k := (S_k(n), S) \\
= \quad & \{\![\text{ definition of updating }]\!\} \\
& (n, (n \overset{k}{\mapsto} S_k(n)) \mid S) \\
= \quad & \{\![\text{ by annihilation (3), since } (n \overset{k}{\mapsto} S_k(n)) \subseteq S]\!\} \\
& (n, S) \ .
\end{aligned}
$$

3. is proved similarly.
□

Again we have localization properties, which are immediate from Corollary 5 and the definitions:

Corollary 7 *Let* $n \overset{\text{def}}{=} ptr(p)$ *and* $r \overset{\text{def}}{=} (n, sto(q))$. *Then*

1. $q \not\vdash n \Rightarrow from((p.k := q).k) = from(q)$.
2. $j \neq k \wedge r.j \not\vdash n \Rightarrow from((p.k := q).j) = from(r.j)$.

We note how selection and updating interact with the *noreach* set:

Lemma 8 *1. For arbitrary* p *and all selectors* j *we have* $noreach(p) \subseteq noreach(p.j)$, *i.e.,* p *pres* $p.j$.
2. $noreach((n, S).k := (m, T)) = noreach(m, \overline{\{n\}} \bowtie T)$.

Proof. The first property is straightforward from the definitions, whereas the second one additionally needs Corollary 5 of Möller (1993b). □

So far we have considered only selections that involve a single selector. However, one also wants to consider longer selection paths. To this end we extend the selection notation to words $u \in K^*$, where K is the set of selectors. To smoothen the notation, concatenation

on K^* will be denoted by . while ε denotes the empty word. We define inductively, for $k \in K$ and $u \in K^*$,

$$\begin{aligned} p.\varepsilon &\stackrel{\text{def}}{=} p\,, \\ p.(k.u) &\stackrel{\text{def}}{=} (p.k).u \text{ if } k \text{ has type } \mathcal{A} \to \mathcal{A} \text{ or } u = \varepsilon. \end{aligned}$$

In all other cases the selection is undefined. We have

Lemma 9 $p.(u.v) = (p.u).v$.

Proof. If $v = \varepsilon$, the claim is trivial. For $v \neq \varepsilon$ we use induction on u. The base case $u = \varepsilon$ is again trivial. For the induction step we calculate

$$\begin{aligned} & p.(k.u.v) \\ = \quad & \{\!\!\{ \text{ definition of . since } v \neq \varepsilon \Rightarrow u.v \neq \varepsilon \,\}\!\!\} \\ & \begin{cases} (p.k).(u.v) & \text{if } k \text{ has type } \mathcal{A} \to \mathcal{A} \\ \text{undefined} & \text{otherwise} \end{cases} \\ = \quad & \{\!\!\{ \text{ induction hypothesis for } u \,\}\!\!\} \\ & \begin{cases} ((p.k).u).v & \text{if } k \text{ has type } \mathcal{A} \to \mathcal{A} \\ \text{undefined} & \text{otherwise} \end{cases} \\ = \quad & \{\!\!\{ \text{ definition of . and image } \}\!\!\} \\ & \begin{cases} ((p.k).u).v & \text{if } k \text{ has type } \mathcal{A} \to \mathcal{A} \\ \text{undefined} & \text{otherwise} \end{cases} \\ = \quad & \{\!\!\{ \text{ definition of . } \}\!\!\} \\ & (p.(k.u)).v\,. \end{aligned}$$

□

For that reason we shall omit the parentheses and write simply $p.u.v$ for the composite selection $p.(u.v)$.

5.2 Overwriting and Abstraction

For pointer implementations that use selective updating it usually is important that the updates work locally. This can be established using the following localization property which is immediate from Corollary 5:

Corollary 10 *For any reasonable abstraction function F we have*

$$p \not\downarrow \mathit{recs}(S) \Rightarrow S \mid p \sim p \qquad \text{(Localization III).}$$

From Corollary 7 we obtain

Corollary 11 *Assume that abstraction function F is reasonable and let $n \stackrel{\text{def}}{=} \mathit{ptr}(p)$ and $r \stackrel{\text{def}}{=} (n, \mathit{sto}(q))$. Then*

1. $q \not\vdash n \Rightarrow (p.k := q).k \sim q$.
2. $j \neq k \wedge r.j \not\vdash n \Rightarrow (p.k := q).j \sim r.j$.

6 ACYCLIC STORES AND FORESTS

6.1 Basic Properties

We have seen that many properties depend on the absence of sharing. This is guaranteed by forests, which are therefore of special interest. For their characterization we need two notions about binary relations. A relation $R : M \leftrightarrow N$ is *acyclic* iff $R^+ \cap I = \emptyset$. Hence R is acyclic iff no element is reachable from itself via a non-empty path. R is *injective* iff $R ; R^\smile \subseteq I$, i.e., iff no two distinct elements have a common successor under R.

Corollary 12 *If $Q \subseteq P$ and P is acyclic or injective, then so is Q.*

Proof. All operations involved in the characterizations of these notions are monotonic w.r.t. inclusion. □

These notions are carried over to stores as follows. A store S is called *acyclic* if $[S]$ is acyclic, and *injective* if $[S] \bowtie \overline{\{\diamond\}}$ is injective. This means that no two different records point to the same *proper* record or, equivalently, that the underlying directed graph has maximal in-degree 1, except perhaps at the pseudo-record $\diamond$. Finally, S is called a *forest* if it is acyclic and injective.

We now state several auxiliary properties of binary relations.

Lemma 13 *Let R be an injective binary relation. Then R^* is upwards locally linear, i.e.,*

$R^* ; (R^*)^\smile \subseteq R^* \cup (R^*)^\smile$.

For the proof see the Appendix.

Lemma 14 *Let R be an acyclic binary relation. Then*

1. $R^\smile \cap R^* = \emptyset$.
2. *If R is injective then* $R^\smile ; R \cap R^+ = \emptyset \wedge R^\smile ; R \cap (R^+)^\smile = \emptyset$.
3. $\overline{I} \cap R^* = R^+$.
4. *If R is injective then* $R^\smile ; R \cap \overline{I} \subseteq \overline{R^* ; (R^*)^\smile}$.

For the proof see the Appendix. It should be noted that Lemmas 13 and 14 hold in all abstract relation algebras as well. We can exploit these properties for acyclic stores or forests to show strong separation properties which will allow localization of side effects:

Lemma 15 *1. Let S be injective. Then for all $x, y \in \mathcal{A}$ we have*

$$\mathit{sharing}(x, y, S) \Rightarrow ((y, S) \vdash x \vee (x, S) \vdash y) .$$

2. *Let S be acyclic and assume $y \in [S]^+(x)$. Then $(y,S) \not\models x$.*
3. *Let S be acyclic and assume $y \in [S]^+(x)$. Then $\forall\, z \in \mathcal{A} : \neg\ sharing(z,x,S) \Rightarrow \neg\ sharing(z,y,S)$.*
4. *Let S be a forest and y,z two distinct successors of x under $[S]$, i.e., assume $y,z \in [S](x) \wedge y \neq z$. Then $\neg\ sharing(x,y,S)$.*
5. *Let S be a forest and assume $y \in [S](x)$. Then*

$$noreach(y,S) = noreach(x,S) \cup \{x\} \cup \bigcup_{z \in [S](x) \setminus \{y\}} reach(z,S)\ .$$

Proof. 1. We have $sharing(x,y,S)$ iff $(x,y) \in [S]^*;[S]^{*\smile} = [S]^*;[S]^{\smile *}$. Now we may apply Lemma 13.

2. is immediate from acyclicity.
3. The assumption implies $reach(y,S) \subseteq reach(x,S)$. Now the claim follows by monotonicity.
4. is immediate from Lemma 14.4.
5. Set $L \stackrel{\text{def}}{=} noreach(x,S) \cup \{x\} \cup \bigcup_{z \in [S](x) \setminus \{y\}} reach(z,S)$. We prove the claim by showing that $reach(y,S) \cup L = recs(S)$ and $reach(y,S) \cap L = \emptyset$. We have

$$\begin{aligned}
& reach(y,S) \cup L \\
= \quad & \{\![\text{ set theory }]\!\} \\
& noreach(x,S) \cup \{x\} \cup \bigcup_{z \in [S](x)} reach(z,S) \\
= \quad & \{\![\text{ fixpoint property of } _^* \ ]\!\} \\
& noreach(x,S) \cup reach(x,S) \\
= \quad & \{\![\text{ definition of } noreach \text{ and set theory }]\!\} \\
& recs(S)
\end{aligned}$$

and

$$\begin{aligned}
& reach(y,S) \cap L \\
= \quad & \{\![\text{ distributivity }]\!\} \\
& (reach(y,S) \cap noreach(x,S)) \ \cup\ (reach(y,S) \cap \{x\}) \ \cup \\
& \bigcup_{z \in [S](x) \setminus \{y\}} (reach(y,S) \cap reach(z,S))\ .
\end{aligned}$$

The first summand is $\emptyset$ by definition of *noreach*, the other two are $\emptyset$ by 2. and 4. □

So far we have considered only stores. A pointer structure (n,S) is called *acyclic, injective* or a *forest* if the store of its reachable part $from(n,S)$ is acyclic, injective or a forest, respectively.

6.2 Composition of Forests

Next we investigate how acyclicity and injectivity propagate through union:

Lemma 16 *Consider relations $R, S : M \leftrightarrow N$.*

1. *If $R \cup S$ is acyclic then so are R and S.*
2. *If R and S are acyclic and $\mathrm{ran}R \cap \mathrm{dom}S = \emptyset$ then $R \cup S$ is acyclic as well.*
3. *$R \cup S$ is injective iff R and S are injective and $R ; S^\smile \subseteq I$.*

For the proof see the Appendix. Again this also holds in all abstract relation algebras. Now we obtain

Corollary 17 *Consider $(m, T) \in newrec((n, S), v)$ and $N \stackrel{\mathrm{def}}{=} \{(m, v_k)\}_{k \in K}$.*

1. *If S and N are acyclic and $\mathrm{ran}[N] \cap \mathrm{dom}[S] = \emptyset$ then (m, T) is acyclic as well.*
2. *If S is injective and $[S] ; [N]^\smile \subseteq I$ then (m, T) is injective as well.*

For the proof of 2. note that N is always injective.

6.3 Forests and Updating

We are now in the position to formulate the strong preservation properties for updating acyclic structures and hence forests.

Corollary 18 *Assume that p is acyclic and $u \neq \varepsilon$. Then*

1. *$from((p.k := p.u).k) = from(p.u)$.*
2. *$j \neq k \Rightarrow from((p.k := p.u).j) = from(p.j)$.*

If, moreover, F is a reasonable abstraction function, then

3. *$(p.k := p.u).k \sim p.u$.*
4. *$j \neq k \Rightarrow (p.k := p.u).j \sim p.j$.*

Proof. This is an application of Corollaries 7 and 11. In this particular case we have $r = p$ and the preconditions are satisfied by Lemma 15. □

Another, although not so important, property is

Corollary 19 *If S is closed and S and T are acyclic then $S \mid T$ is acyclic as well.*

Proof. We have $S \mid T = S \cup R$ where $R = \overline{\mathrm{dom}[S]} \bowtie T$. Using the criterion from Lemma 16.2. we obtain

$$
\begin{array}{ll}
 & \mathsf{ran}[S] \cap \mathsf{dom} R \\
= & \quad \{\!\![\text{ by (1) }]\!\!\} \\
 & \mathsf{ran}[S] \cap \overline{\mathsf{dom}[S]} \cap \mathsf{dom} T \\
\subseteq & \quad \{\!\![\text{ since } S \text{ is closed }]\!\!\} \\
 & \{\diamond\} \cap \mathsf{dom} T \\
= & \quad \{\!\![\text{ definition of store }]\!\!\} \\
 & \emptyset.
\end{array}
$$

□

Moreover, from Lemma 16 we obtain

Corollary 20 *Let* $r = (p.k := q)$ *and set* $n \stackrel{\text{def}}{=} ptr(r) = ptr(p)$ *and* $R \stackrel{\text{def}}{=} \overline{\{n\}} \bowtie sto(r) = \overline{\{n\}} \bowtie sto(q)$.

1. *If* $ptr(q) \notin \{n\} \cup \mathsf{dom}[R]$ *and* R *is acyclic, then* r *is acyclic as well.*
2. *If* R *is acyclic and* $n \notin \{ptr(q)\} \cup sto(q)(\overline{\{n\}})$, *then* r *is acyclic as well.*
3. r *is injective iff* $ptr(q) \notin \mathsf{ran}[R]$ *and* R *is injective.*

Proof. 1. Immediate from Lemma 16.2.
2. Immediate from Lemma 16.2.
3. Set $m \stackrel{\text{def}}{=} ptr(q)$. Then $[(n \stackrel{k}{\mapsto} m)]\,;[R]^{\smile} = \{(n,m)\}\,;[R^{\smile}]$ is contained in I iff $\{(n,m)\}\,;[R]^{\smile} = \emptyset$, which is equivalent to $m \notin \mathsf{ran}[R]$. The remainder is clear from Lemma 16.3. and injectivity of $\{(n,m)\}$.
□

Finally, we give a stronger criterion for acyclicity of overwritten structures:

Lemma 21 *Consider* $r \stackrel{\text{def}}{=} (n,S).k := (m,T) = (n,(n \stackrel{k}{\mapsto} m) \cup R)$ *where* $R \stackrel{\text{def}}{=} \overline{\{n\}} \bowtie T$. *Then* r *is acyclic iff* R *is acyclic and* $(m,R) \nvdash n$.

Proof. Set $S \stackrel{\text{def}}{=} (n \stackrel{k}{\mapsto} m) \cup R$.
($\Rightarrow$) By Corollary 12, R is acyclic. Suppose $n \in reach(m,R)$. Then by monotonicity also $n \in reach(m,S)$ and hence $m[S]^{*}n[S]m$, a contradiction.
($\Leftarrow$) Suppose S is cyclic. Since R is acyclic, every cycle in S must involve the only arc of $(n \stackrel{k}{\mapsto} m)$. Choose a cycle of minimal length, say $l[S]^{*}n[S]m[S]^{*}l$. By minimality then already $l[R]^{*}n$ and $m[R]^{*}l$. But then $m[R]^{*}n$, i.e., $(m,R) \vdash n$, a contradiction. □

7 POINTER IMPLEMENTATION OF BINARY TREES

7.1 Abstract Trees

The set $\mathcal{T}$ of *binary trees* with elements of $\mathcal{N}$ as nodes is defined inductively as the least set $\mathcal{X}$ with

$$\varepsilon \cup \mathcal{X} \times \mathcal{N} \times \mathcal{X} \subseteq \mathcal{X} ,$$

where ε now also denotes the empty tree and $\_ \times \_ \times \_$ is the ternary cartesian product. A non-empty tree, i.e., an element of $\mathcal{T} \times \mathcal{N} \times \mathcal{T}$, will be denoted as triple $\langle l, x, r \rangle$ with left subtree $l \in \mathcal{T}$, node $x \in \mathcal{N}$ and right subtree $r \in \mathcal{T}$.

7.2 An Abstraction Function

Let now $\mathcal{P}$ denote the set of all pointer structures over the record scheme for binary trees, as discussed in Section 3.1. The abstraction function $tree : \mathcal{P} \rightsquigarrow \mathcal{T}$ constructs the tree reachable from a record in a store. For $n \in \mathcal{A}$ we set

$$tree(n, B) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = \Diamond , \\ \langle tree(B_l(n), B), B_v(n), tree(B_r(n), B) \rangle & \text{if } n \neq \Diamond . \end{cases}$$

In the case where a cycle is reachable from n in B, this recursion is non-terminating. In a strict underlying semantics this means that the value of $tree(n, B)$ is undefined, whereas in a non-strict setting the value of $tree(n, B)$ is an infinite tree corresponding to an unwinding of the subgraph reachable from n in B. Since we are working in a relational setting, the strict interpretation is relevant here. So from now on we shall assume that *tree* is used only for acyclic pointer structures. Note, however, that *tree* also works for non-forests, i.e., dags: it copies the abstract trees that correspond to shared substructures.

The recursion pattern is typical of an *unfold operation* or *anamorphism* (see Meijer et al. (1991), Bird (1996)). A thorough investigation of this connection is left to subsequent papers.

Using the selection notation from Section 5.1, the definition of *tree* can be simplified to

$$tree(p) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } ptr(p) = \Diamond , \\ \langle tree(p.l), p.v, tree(p.r) \rangle & \text{otherwise} . \end{cases}$$

We have (see the Appendix for the proof)

Lemma 22 *The abstraction function tree is reasonable.*

7.3 Search

We now calculate pointer implementations of a number of sample operations. First we treat the operation $elem : \mathcal{N} \times \mathcal{T} \to \mathbb{B}$ that tests whether an element occurs in a binary search tree over a totally ordered set $\mathcal{N}$ of node values. It is recursively defined by

$$\begin{aligned} elem(x, \varepsilon) &= \mathsf{false}\ , \\ elem(x, \langle l, y, r \rangle) &= \begin{cases} \mathsf{true} & \text{if } x = y\ , \\ elem(x, l) & \text{if } x < y\ , \\ elem(x, r) & \text{if } x > y\ . \end{cases} \end{aligned}$$

For a derivation of this recursive version from an implicit specification via the multiset of elements contained in a tree see Dosch, Möller (1997).

According to our general scheme from Section 4, the pointer implementation *pelem* is then specified by

$$pelem(x, p) = elem(x, tree(p))\ ,$$

where we assume that p is acyclic to ensure definedness of $tree(p)$. The goal is now to calculate a direct recursion for *pelem*.

For the case that $ptr(p) = \Diamond$ we have $tree(p) = \varepsilon$ and hence

$$pelem(x, p) = \mathsf{false}\ .$$

Otherwise we calculate

$$\begin{aligned} & pelem(x, p) \\ = \quad & \{\![\text{ unfold definitions of } pelem \text{ and } tree \ ]\!\} \\ & elem(x, \langle tree(p.l), p.v, tree(p.r) \rangle) \\ = \quad & \{\![\text{ unfold definition of } elem \ ]\!\} \\ & \begin{cases} \mathsf{true} & \text{if } x = p.v \\ elem(x, tree(p.l)) & \text{if } x < p.v \\ elem(x, tree(p.r)) & \text{if } x > p.v \end{cases} \\ = \quad & \{\![\text{ fold definition of } pelem \ ]\!\} \\ & \begin{cases} \mathsf{true} & \text{if } x = p.v \\ pelem(x, p.l) & \text{if } x < p.v \\ pelem(x, p.r) & \text{if } x > p.v\ . \end{cases} \end{aligned}$$

The requirement that the argument structure be acyclic ensures that the implementation is as undefined as the specification. The recursion by itself will terminate if the searched element is found before a cycle is entered.

Of course, this derivation was easy, since no updating and hence no questions of sharing arose. We have presented it merely to show the notions at work as a preparation for the next example.

7.4 Insertion

Next we treat the operation $ins : \mathcal{N} \to (\mathcal{T} \rightsquigarrow \mathcal{T})$ that inserts an element into a binary search tree. It is recursively defined by

$$\begin{aligned} ins(x)(\varepsilon) &= \langle \varepsilon, x, \varepsilon \rangle \,, \\ ins(x)(\langle l, y, r \rangle) &= \begin{cases} \langle ins(x)(l), y, r \rangle & \text{if } x \leq y \,, \\ \langle l, y, ins(x)(r) \rangle & \text{if } x > y \,. \end{cases} \end{aligned}$$

Consider again Dosch, Möller (1997) for a derivation of this recursive version from an implicit specification.

Using our general scheme from Section 4 we specify a general pointer implementation *pins* by requiring, for all $x \in \mathcal{N}$, that $pins(x)$ $POI(ALL)$ $ins(x)$, i.e., $pins(x)$; $tree = tree$; $ins(x)$ and want to find a direct recursion for $pins(x)$.

The derivation will exhibit the need for a strengthening of ALL to allow application of our updating laws; this will lead to an additional invariant.

Again we assume that the argument p is acyclic to ensure definedness of $tree(p)$. For the case that $ptr(p) = \diamond$ we have $tree(p) = \varepsilon$ and hence $tree(pins(x)(p)) = \langle \varepsilon, x, \varepsilon \rangle$. According to the definitions of *tree* and *newrec* and Corollary 17 this can be achieved by choosing

$$pins(x)(p) = newrec(p, \langle \diamond, x, \diamond \rangle)$$

in this case.

For $ptr(p) \neq \diamond$ we only treat the subcase $x \leq p.v$, the other one being symmetric. We calculate

$$\begin{aligned} & ins(x)(tree(p)) \\ = & \quad \{\![\text{ unfold definition of } tree \,]\!\} \\ & ins(x)(\langle tree(p.l), p.v, tree(p.r) \rangle) \\ = & \quad \{\![\text{ unfold definition of } ins \,]\!\} \\ & \langle ins(x)(tree(p.l)), p.v, tree(p.r) \rangle) \\ = & \quad \{\![\text{ fold with specification of } pins(x) \,]\!\} \\ & \langle tree(pins(x)(p.l)), p.v, tree(p.r) \rangle) \,. \end{aligned}$$

The aim now is to find for certain $q' \in pins(x)(p.l)$ a pointer structure q such that this expression is equal to $tree(q)$. According to our general aim of making do with minimal change we try to choose $q \stackrel{\text{def}}{=} (p.l := q')$. Since p is assumed to be acyclic, we have by Lemma 15 that $ptr(p) \in noreach(p.l)$. If we therefore additionally require q' *loc* $p.l$, Corollary 11 tells us that $q.l \sim q'$, $q.v = p.v$ and $q.r \sim p.r$. Then we may continue

$$\begin{aligned} & \langle tree(q'), p.v, tree(p.r) \rangle) \\ = & \quad \{\![\text{ as just explained }]\!\} \\ & \langle tree(q.l), q.v, tree(q.r) \rangle \end{aligned}$$

$$
\begin{array}{ll}
= & \{\!\!\{ \text{ fold definition of } \mathit{tree} \ \}\!\!\} \\
& \mathit{tree}(q) \ ,
\end{array}
$$

so that we may choose $\mathit{pins}(x)(p) = q$ in this case. It is easily checked, using Lemma 8, that p *loc* q again, so that the additional invariant is preserved. Moreover, the above base case establishes *loc*. Altogether we obtain the recursion

$$
\begin{array}{l}
\mathit{pins}(x)(p) = \text{if } \mathit{ptr}(p) = \Diamond \\
\qquad\qquad \text{then } \mathit{newrec}(p, \langle \Diamond, x, \Diamond \rangle) \\
\qquad\qquad \text{else if } x \leq y \text{ then } p.l := \mathit{pins}(x)(p.l) \\
\qquad\qquad\qquad\qquad \text{else } p.r := \mathit{pins}(x)(p.r) \text{ fi fi}
\end{array}
$$

which satisfies $\mathit{pins}(x)$ $\mathit{POI}(\mathit{loc})$ $\mathit{ins}(x)$.

7.5 Rotation

When considering balanced search trees such as AVL trees one uses tree rotations. We consider left rotation $\mathit{lrot} : \mathcal{T} \rightsquigarrow \mathcal{T}$, specified by

$$\mathit{lrot}(\langle l, x, \langle m, y, r \rangle \rangle) \ = \ \langle \langle l, x, m \rangle, y, r \rangle \ .$$

Using our general scheme from Section 4 we specify a localized pointer implementation *plrot* by requiring *plrot* $\mathit{POI}(\mathit{ALL})$ *lrot* and want to find an explicit version of *plrot*.

Again we assume that the argument p is acyclic to ensure definedness of $\mathit{tree}(p)$. We have

$$
\begin{array}{ll}
& \mathit{lrot}(\mathit{tree}(p)) \\
= & \quad \{\!\!\{ \text{ unfold definition of } \mathit{tree} \text{ twice } \}\!\!\} \\
& \mathit{lrot}(\langle \mathit{tree}(p.l), p.v, \langle \mathit{tree}(p.r.l), p.r.v, \mathit{tree}(p.r.r) \rangle \rangle) \\
= & \quad \{\!\!\{ \text{ unfold definition of } \mathit{lrot} \ \}\!\!\} \\
& \langle \langle \mathit{tree}(p.l), p.v, \mathit{tree}(p.r.l) \rangle, p.r.v, \mathit{tree}(p.r.r) \rangle) \\
= & \quad \{\!\!\{ \text{ defining } u \stackrel{\text{def}}{=} p.r \ \}\!\!\} \\
& \langle \langle \mathit{tree}(p.l), p.v, \mathit{tree}(u.l) \rangle, u.v, \mathit{tree}(u.r) \rangle) \\
= & \quad \{\!\!\{ \text{ defining } q \stackrel{\text{def}}{=} p.r := u.l \text{ and using Corollary 18 and Lemma 21 } \}\!\!\} \\
& \langle \langle \mathit{tree}(q.l), q.v, \mathit{tree}(q.r) \rangle, u.v, \mathit{tree}(u.r) \rangle) \\
= & \quad \{\!\!\{ \text{ fold definition of } \mathit{tree} \ \}\!\!\} \\
& \langle \mathit{tree}(q), u.v, \mathit{tree}(u.r) \rangle) \\
= & \quad \{\!\!\{ \text{ defining } w \stackrel{\text{def}}{=} u.l := q \text{ and using Corollary 18 and Lemma 21 } \}\!\!\} \\
& \langle \mathit{tree}(w.l), w.v, \mathit{tree}(w.r) \rangle) \\
= & \quad \{\!\!\{ \text{ fold definition of } \mathit{tree} \ \}\!\!\} \\
& \mathit{tree}(w) \ .
\end{array}
$$

So we may choose

$$
\begin{array}{l}
\mathit{plrot}(p) = \\
\quad \mathsf{let}\ u = p.r \\
\qquad q = (p.r := u.l) \\
\quad \mathsf{in}\ u.l := q\ .
\end{array}
$$

The derivation has shown that here we do not need a strengthening of *ALL*, i.e., no additional invariant. Note that the intermediate pointer structure q is not injective and hence not a forest, but still acyclic, so that our laws apply.

8 CONCLUSION AND OUTLOOK

The chosen abstraction seems adequate, as the fairly concise derivations in the examples show. It is encouraging that to a large extent the treatment is independent of the particular data structures involved.

The approach also covers the examples of in-situ concatenation and reversal of lists dealt with in Möller (1991–1993) and makes the derivations considerably shorter. We have to omit these examples here for lack of space and note only that the predicate *sharing* now replaces the one called *disjoint* in our earlier approaches.

It remains to integrate the approach with the general theory of unfold operations or anamorphisms (see Meijer et al. (1991), Bird (1996)). The proof of Lemma 22 leads us to conjecture that every anamorphic abstraction function is reasonable.

Although our notions were defined for general pointer structures, in the examples we have concentrated on tree-like structures. However, many seemingly cyclic structures such as doubly-linked lists or even threaded trees behave as forests when selection is considered along a certain subclass of links only. Preliminary studies indicate that our results for forests can be carried over to these structures. Moreover, recent work (Möller 1997) shows that the techniques extend in a simple way to properly cyclic structures.

Acknowledgements I am grateful to Jules Desharnais and Walter Dosch for valuable remarks on drafts of this paper. This research was partially sponsored by Esprit Working Group 8533 *NADA — New Hardware Design Methods.*

9 APPENDIX: PROOFS

Proof of Lemma 2.

1. Let $T \stackrel{\text{def}}{=} \mathit{reach}(s, S) \bowtie S$. We first show $\mathit{reach}(s, S) = \mathsf{set}\, s \cup \mathsf{ran}[T]$.

$$
\begin{array}{ll}
 & \mathsf{set}\, s \cup \mathsf{ran}[T] \\
= & \quad \{\![\text{ definition of } T \text{ and (1) }]\!\} \\
 & \mathsf{set}\, s \cup [S](\mathit{reach}(s, S)) \\
= & \quad \{\![\text{ definition of } \mathit{reach}]\!\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathrm{set}\, s \cup [S]([S]^*(\mathrm{set}\, s)) \\
= & \quad \{\!\!\{ \text{ recursion for } \_^* \}\!\!\} \\
 & [S]^*(\mathrm{set}\, s) \\
= & \quad \{\!\!\{ \text{ definition of } \mathit{reach} \}\!\!\} \\
 & \mathit{reach}(s, S) \; .
\end{array}
$$

From this the claim is immediate.

2. Set $(s, T) \stackrel{\text{def}}{=} \mathit{from}(s, S)$. We first show that $\mathit{reach}(s, T) = \mathit{reach}(s, S)$. The inclusion $\subseteq$ follows from $T \subseteq S$ and monotonicity of $\_^*$. For the other inclusion we show more generally $\forall\, T \subseteq \mathit{reach}(s, S) : [S]^*(T) \subseteq [T]^*(T)$ by fixpoint induction on the continuous predicate $PP(X) \stackrel{\text{def}}{\Leftrightarrow} \forall\, T \subseteq \mathit{reach}(s, S) : X(T) \subseteq [T]^*(T)$ and the functional $\tau : X \mapsto I \cup [S]\,;X$ associated with the recursive definition of $\_^*$. The induction base $PP(\emptyset)$ is trivial. For the induction step we calculate, assuming $PP(X)$:

$$
\begin{array}{ll}
 & \tau(X)(T) \\
= & \quad \{\!\!\{ \text{ distributivity, and definition of image and composition } \}\!\!\} \\
 & T \cup X([S](T)) \\
\subseteq & \quad \{\!\!\{ \; PP(X), \text{ and } T \subseteq \mathit{reach}(s, S) \Rightarrow [S](T) \subseteq \mathit{reach}(s, S) \; \}\!\!\} \\
 & T \cup [T]^*([S](T)) \\
= & \quad \{\!\!\{ \; T \subseteq U \Rightarrow R(T) = (U \bowtie R)(T) \; \}\!\!\} \\
 & T \cup [T]^*([T](T)) \\
= & \quad \{\!\!\{ \text{ definition of image and composition and distributivity } \}\!\!\} \\
 & (I \cup [T]^* \,;[T])(T) \\
= & \quad \{\!\!\{ \text{ fixpoint property of } \_^* \}\!\!\} \\
 & [T]^*(T) \; .
\end{array}
$$

Now we have

$$
\begin{array}{ll}
 & \mathit{from}(s, T) \\
= & \quad \{\!\!\{ \text{ definition of } \mathit{from} \}\!\!\} \\
 & (s, \mathit{reach}(s, T) \bowtie T) \\
= & \quad \{\!\!\{ \text{ definition of } T \}\!\!\} \\
 & (s, \mathit{reach}(s, T) \bowtie (\mathit{reach}(s, S) \bowtie S)) \\
= & \quad \{\!\!\{ \text{ because } \mathit{reach}(s, T) = \mathit{reach}(s, S) \}\!\!\} \\
 & (s, \mathit{reach}(s, S) \bowtie (\mathit{reach}(s, S) \bowtie S)) \\
= & \quad \{\!\!\{ \text{ idempotence of restriction } \}\!\!\} \\
 & (s, \mathit{reach}(s, S) \bowtie S) \\
= & \quad \{\!\!\{ \text{ definition of } \mathit{from} \}\!\!\} \\
 & \mathit{from}(s, S) \; .
\end{array}
$$

3. See Möller (1993a), Corollary 3.

Proof of Lemma 4. Suppose that $from(s,S) = from(s,T)$. By Lemma 2.1., we have $reach(s,S) = reach(s,T)$ $(*)$.
Now we obtain

$$
\begin{array}{ll}
 & (s, reach(s,S) \bowtie S) \\
= & \quad \{\![\text{ definition of } from \ ]\!\} \\
 & from(s,S) \\
= & \quad \{\![\text{ assumption }]\!\} \\
 & from(s,T) \\
= & \quad \{\![\text{ definition of } from \ ]\!\} \\
 & (s, reach(s,T) \bowtie T) \\
= & \quad \{\![\text{ by } (*) \ ]\!\} \\
 & (s, reach(s,S) \bowtie T) \ .
\end{array}
$$

Now $F(s,S) = F(t,T)$ follows from $s = t$ and the assumption.

Proof of Lemma 13. We show more abstractly for Kleene algebras (see e.g. Möller (1993a)) that

$$a \cdot b \leq 1 \Rightarrow a^* \cdot b^* \leq a^* + b^* \ .$$

The proof is a fixpoint induction on the recursive definition

$$c^* \stackrel{\text{def}}{=} \mu x \,.\, 1 + c \cdot x \ ,$$

where μ is the least fixpoint operator, using the continuous predicate

$$PP(x) \stackrel{\text{def}}{=} a^* \cdot x \leq a^* + b^* \wedge x \leq b^* \ .$$

The induction base $PP(0)$ is trivial by strictness of $\cdot$. For the induction step we calculate

$$
\begin{array}{ll}
 & a^* \cdot (1 + b \cdot x) \\
= & \quad \{\![\text{ distributivity and neutrality of } 1 \ ]\!\} \\
 & a^* + a^* \cdot b \cdot x \\
= & \quad \{\![\text{ dual fixpoint property of } a^* \ ]\!\} \\
 & a^* + (1 + a^* \cdot a) \cdot b \cdot x \\
= & \quad \{\![\text{ distributivity and neutrality of } 1 \ ]\!\} \\
 & a^* + b \cdot x + a^* \cdot a \cdot b \cdot x \\
\leq & \quad \{\![\text{ by assumption } a \cdot b \leq 1 \text{ monotonicity and neutrality of } 1 \ ]\!\}
\end{array}
$$

$$
\begin{array}{ll}
 & a^* + b \cdot x + a^* \cdot x \\
\leq & \quad \{\!\!\{ \text{ by induction hypothesis } PP(x) \text{ and monotonicity } \}\!\!\} \\
 & a^* + b \cdot b^* + a^* + b^* \\
= & \quad \{\!\!\{ \, b \cdot b^* \leq b^* \text{ and idempotence of } + \, \}\!\!\} \\
 & a^* + b^*
\end{array}
$$

and

$$
\begin{array}{ll}
 & 1 + b \cdot x \\
\leq & \quad \{\!\!\{ \text{ by induction hypothesis } PP(x) \text{ and monotonicity } \}\!\!\} \\
 & 1 + b \cdot b^* \\
= & \quad \{\!\!\{ \text{ fixpoint property of } b^* \, \}\!\!\} \\
 & b^* \, .
\end{array}
$$

Now the claim follows from $(R^*)^\smile = (R^\smile)^*$.

Proof of Lemma 14.

1.
$$
\begin{array}{ll}
 & R^\smile \cap R^* \\
= & \quad \{\!\!\{ \text{ neutrality } \}\!\!\} \\
 & R^\smile ; I \cap R^* \\
\subseteq & \quad \{\!\!\{ \text{ Dedekind's rule, converse } \}\!\!\} \\
 & (R^\smile \cap R^* ; I^\smile) ; (I \cap R ; R^*) \\
= & \quad \{\!\!\{ \text{ definition of } \_^+ \, \}\!\!\} \\
 & (R^\smile \cap R^* ; I^\smile) ; (I \cap R^+) \\
= & \quad \{\!\!\{ \text{ assumption } \}\!\!\} \\
 & (R^\smile \cap R^* ; I^\smile) ; \emptyset \\
= & \quad \{\!\!\{ \text{ strictness } \}\!\!\} \\
 & \emptyset \, .
\end{array}
$$

2.
$$
\begin{array}{ll}
 & R^\smile ; R \cap R^+ \\
\subseteq & \quad \{\!\!\{ \text{ Dedekind's rule, converse } \}\!\!\} \\
 & (R^\smile \cap R^+ ; R^\smile) ; (R \cap R ; R^+) \\
= & \quad \{\!\!\{ \text{ definition of } \_^+ \, \}\!\!\} \\
 & (R^\smile \cap R^* ; R ; R^\smile) ; (R \cap R ; R^+) \\
\subseteq & \quad \{\!\!\{ \text{ assumption, neutrality } \}\!\!\} \\
 & (R^\smile \cap R^*) ; (R \cap R ; R^+) \\
= & \quad \{\!\!\{ \text{ by 1. } \}\!\!\} \\
 & \emptyset ; (R \cap R ; R^+) \\
= & \quad \{\!\!\{ \text{ strictness } \}\!\!\}
\end{array}
$$

$\emptyset$.

From this we obtain

$$
\begin{array}{ll}
& R^{\smile}; R \cap (R^+)^{\smile} \\
= & \quad \{\!\!\{ \text{converse} \}\!\!\} \\
& (R^{\smile}; R)^{\smile} \cap (R^+)^{\smile} \\
= & \quad \{\!\!\{ \text{distributivity} \}\!\!\} \\
& (R^{\smile}; R \cap R^+)^{\smile} \\
= & \quad \{\!\!\{ \text{by above} \}\!\!\} \\
& \emptyset^{\smile} \\
= & \quad \{\!\!\{ \text{strictness} \}\!\!\} \\
& \emptyset .
\end{array}
$$

3.

$$
\begin{array}{ll}
& \overline{I} \cap R^* \\
\subseteq & \quad \{\!\!\{ \text{definition of } \_^+ \}\!\!\} \\
& \overline{I} \cap (I \cup R^+) \\
= & \quad \{\!\!\{ \text{distributivity} \}\!\!\} \\
& (\overline{I} \cap I) \cup (\overline{I} \cap R^+) \\
= & \quad \{\!\!\{ \text{Boolean algebra, assumption} \}\!\!\} \\
& \emptyset \cup R^+ \\
= & \quad \{\!\!\{ \text{neutrality} \}\!\!\} \\
& R^+ .
\end{array}
$$

4.

$$
\begin{array}{ll}
& R^{\smile}; R \cap \overline{I} \cap R^* ; (R^*)^{\smile} \\
\subseteq & \quad \{\!\!\{ \text{by Lemma 13} \}\!\!\} \\
& R^{\smile}; R \cap \overline{I} \cap (R^* \cup (R^*)^{\smile}) \\
= & \quad \{\!\!\{ \text{distributivity} \}\!\!\} \\
& (R^{\smile}; R \cap \overline{I} \cap R^*) \cup (R^{\smile}; R \cap \overline{I} \cap (R^*)^{\smile}) \\
= & \quad \{\!\!\{ \text{by 3., converse} \}\!\!\} \\
& (R^{\smile}; R \cap R^+) \cup (R^{\smile}; R \cap (R^+)^{\smile}) \\
= & \quad \{\!\!\{ \text{by 2.} \}\!\!\} \\
& \emptyset .
\end{array}
$$

Proof of Lemma 16.

1. This is immediate from Corollary 12.
2. The assumption $\mathrm{ran} R \cap \mathrm{dom} S = \emptyset$ is equivalent to $R ; S = \emptyset$. Now easy regular algebra shows $(R \cup S)^* = S^* ; R^*$ and hence $(R \cup S)^+ = S^+ \cup S^+ ; R^+ \cup R^+$. It thus remains to show $S^+ ; R^+ \cap I = \emptyset$. For this we first note that $R ; S = \emptyset \Rightarrow S ; R \cap I = \emptyset$, since

$$
\begin{array}{ll}
& R ; S \subseteq \emptyset \\
\Leftrightarrow & \quad \{\!\!\{ \text{ Schröder equivalences } \}\!\!\} \\
& R^{\smile} ; \overline{\emptyset} \subseteq \overline{S} \\
\Rightarrow & \quad \{\!\!\{ \text{ monotonicity } \}\!\!\} \\
& R^{\smile} ; I \subseteq \overline{S} \\
\Leftrightarrow & \quad \{\!\!\{ \text{ neutrality } \}\!\!\} \\
& I ; R^{\smile} \subseteq \overline{S} \\
\Leftrightarrow & \quad \{\!\!\{ \text{ Schröder equivalences } \}\!\!\} \\
& S ; R \subseteq \overline{I} \, .
\end{array}
$$

Now the claim follows from $R^+ ; S^+ = R^* ; R ; S ; S^* = \emptyset$ by $R ; S = \emptyset$ and strictness of composition.

3. By distributivity, $(R \cup S) ; (R \cup S)^{\smile} = R ; R^{\smile} \cup R ; S^{\smile} \cup S ; R^{\smile} \cup S ; S^{\smile}$. Now the claim is immediate from $S ; R^{\smile} = (R ; S^{\smile})^{\smile}$.

Proof of Lemma 22. We show the premise of Lemma 4 by fixpoint induction on the recursive definition of *tree* and the continuous predicate

$$PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\, n, S, T : \mathit{reach}(n, S) \bowtie S = \mathit{reach}(n, S) \bowtie T \Rightarrow h(n, S) = h(n, T) \, .$$

The induction basis $PP(\Omega)$ is trivial. Assume now $PP(h)$. The functional τ associated with the recursive definition of *tree* is

$$\tau(h)(q) = \text{if } \mathit{ptr}(q) = \Diamond \text{ then } \varepsilon \text{ else } \langle h(q.l), q.v, h(q.r) \rangle \, .$$

First we observe that $n \in \mathit{reach}(n, S)$ and $\mathit{reach}(n, S) \bowtie S = \mathit{reach}(n, S) \bowtie T$ imply

$$
\begin{array}{l}
\{n\} \bowtie S = \{n\} \bowtie T \wedge \\
\mathit{reach}(l, S) \bowtie S = \mathit{reach}(l, S) \bowtie T \wedge \qquad (*) \\
\mathit{reach}(r, S) \bowtie S = \mathit{reach}(r, S) \bowtie T \, ,
\end{array}
$$

where $l = S_l(n) = T_l(n)$ and $r = S_r(n) = T_r(n)$. Now we calculate

$$
\begin{array}{ll}
& \tau[h](n, S) \\
= & \quad \{\!\!\{ \text{ definition } \}\!\!\} \\
& \text{if } n = \Diamond \text{ then } \varepsilon \\
& \qquad \text{else } \langle h((n, S).l), (n, S).v, h((n, S).r) \rangle \text{ fi} \\
= & \quad \{\!\!\{ \text{ by } (*) \text{ and } PP(h) \}\!\!\} \\
& \text{if } n = \Diamond \text{ then } \varepsilon \\
& \qquad \text{else } \langle h((n, T).l), (n, T).v, h((n, T).r) \rangle \text{ fi} \\
= & \quad \{\!\!\{ \text{ definition } \}\!\!\} \\
& \tau(h)(n, T) \, .
\end{array}
$$

REFERENCES

U. Berger, W. Meixner, B. Möller: Calculating a garbage collector. In: M. Broy, M. Wirsing (eds.): Methods of programming. Lecture Notes in Computer Science **544**. Berlin: Springer 1991, 137–192

R. Bird: Functional algorithm design. Science of Computer Programming **26**, 15–31 (1996)

R.S. Bird, O. de Moor: Algebra of programming. Prentice-Hall 1996

W. Dosch, B. Möller: Calculating a functional module for binary search trees. In: W. Kluge (ed.): Proceedings of the 8th International Workshop on Implementation of Functional Languages, Bonn, Sept. 16–18, 1996. Lecture Notes in Computer Science . Berlin: Springer (to appear). Preliminary version accessible through `http://www.math.uni-augsburg.de/~moeller`

E.C.R. Hehner: A practical theory of programming. Berlin: Springer 1993

C.A.R. Hoare: Proofs of correctness of data representations. Acta Informatica **1**, 271–281 (1972)

E.Meijer, M.Fokkinga, R. Paterson: Functional programming with bananas, lenses, envelopes and barbed wire. In: J. Hughes (ed.): Functional programming and computer architecture. Lecture Notes in Computer Science **523**. Berlin: Springer 1991, 124–144

B. Möller: Formal derivation of pointer algorithms. In: M. Broy (Hrsg.): Informatik und Mathematik. Berlin: Springer1991, 419–440

B. Möller: Development of graph and pointer algorithms. In: B. Möller, H.A. Partsch, S.A. Schuman (eds.): Formal program development. Lecture Notes in Computer Science **755**. Berlin: Springer 1993, 123–160

B. Möller: Towards pointer algebra. Science of Computer Programming **21**, 57–90 (1993)

B. Möller: Linked lists calculated. Submitted for publication (1997)

P. Pepper, B. Möller: Programming with (finite) mappings. In: M. Broy (ed.): Informatik und Mathematik. Berlin: Springer 1991, 381–405

G. Schmidt, T. Ströhlein: Relations and graphs. Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Berlin: Springer 1993

J.M. Spivey: The Z notation, 2nd edition. New York: Prentice-Hall 1994

BIOGRAPHY

Bernhard Möller studied informatics and mathematics at TU Munich and Cornell, with an M.Sc. from Cornell and doctorate and habilitation at TU Munich. Within the project CIP at TU Munich he worked on the design of the wide-spectrum language CIP-L and of the transformation system CIP-S, as well as on the methodology of deductive design. Since 1990 he has been Professor of Informatics at the University of Augsburg. He is a member of IFIP WG 2.1 *Algorithmic Languages and Calculi* and the coordinator of Esprit WG 8533 *NADA - New Hardware Design Methods.*