Calculating With Acyclic and Cyclic Lists

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

Abstract

We use a relational model of pointer structures to calculate a number of standard algorithms on singly linked lists, both acyclic and cyclic. This shows that our techniques are not just useful for tree-like structures, but apply to general pointer structures as well.

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. A more algebraic approach was presented in the work of Berger et al. (1991) and Möller (1991–1993) and developed into more general form by Möller (1997). However, in the latter paper only examples with tree-like structures were treated. We show that the approach covers cyclic structures as well. In fact, we demonstrate that the derivations for the acyclic case can be carried over to the cyclic case quite simply and hence be re-used. Proofs that are missing from the present paper are straightforward or can be found in Möller (1997).

2 Relational Notation

Our prominent mathematical tool are 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 $R \in M \leftrightarrow N$ and $R : M \leftrightarrow N$ synonymously. By $\mathsf{dom}\ R$ and $\mathsf{ran}\ R$ we denote domain and range of relation R. 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 analysing the reachable part of a pointer structure we shall use the *domain restriction* of R to a subset $L \subseteq M$ given by

$$L \bowtie R \stackrel{\text{def}}{=} R \cap L \times N \ .$$

Dually, the *range restriction* of $R : M \leftrightarrow N$ to a subset $L \subseteq N$ is $R \bowtie L \stackrel{\text{def}}{=} R \cap M \times L$.

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) function* iff $R^\smile ; R \subseteq I_N$, in other terms, iff

$$\forall\ x, y, z : xRy \wedge xRz \Rightarrow y = z \ .$$

We write $R : M \rightsquigarrow N$ to indicate that R is a partial function. For further notions and laws concerning relations consider e.g. Schmidt, Ströhlein (1993).

3 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.

The shape of records is usually defined by (the equivalent of) a type declaration. Recursive use of the same type either has to be explicitly shielded by a corresponding pointer type (e.g. Pascal or C/C++) or is implicitly played back to an anonymous pointer type (e.g. in functional languages or Java). Our equivalent to a type declaration is a *record type*. It consists of a non-empty set K of *selectors* each with a functionality $\mathcal{A} \to \mathcal{A}$ (corresponding to a pointer-valued field in a record of that type) or $\mathcal{A} \to \mathcal{N}_j$ for some $j \in J$ (corresponding to a scalar field). For simplicity we deal here only with the case of a single (possibly) recursive record type; it should be obvious how to generalise the treatment to systems of mutually recursive record types.

Whereas the record type describes the shape of the records, a concrete aggregation of records of a given type in memory is described by a *store*, ie. a family $S = (S_k)_{k \in K}$ of partial functions such that

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

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

A store may be viewed as a labeled directed graph: the records and node values are the vertices and the selectors are the arc labels, where S_k is the set of arcs labeled by k. We keep these sets separate to be able to model updating along a single selector adequately. The requirement that the S_k be functions reflects that selection from a record yields at most one value. By the third requirement, in a store, $\diamond$ is not related to anything and hence cannot be "dereferenced". The relational operations are extended componentwise to stores.

Our running example of a record type is one for singly linked lists. Assume a set $\mathcal{N}$ of node values and two selectors $head, tail$ of functionalities $head : \mathcal{A} \to \mathcal{N}$ and $tail : \mathcal{A} \to \mathcal{A}$. Then a list store L consists of two partial maps $L_{head} : \mathcal{A} \rightsquigarrow \mathcal{N}$ and $L_{tail} : \mathcal{A} \rightsquigarrow \mathcal{A}$, where L_{head} returns the first element of the list and L_{tail} gives the next record in the list.

Frequently we want to abstract from the node values of the records and consider just their interrelationship through the pointers. For 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 functionality $\mathcal{A} \to \mathcal{A}$. In the graph view, this operation "forgets" the arc labels and the arcs leading to the node values. For instance, the access relation for a list store L is $[L] = L_{tail}$.

A store usually contains several record structures, eg. several lists. Each substructure is accessed via an *entry point*, ie. the address of its "root" record (eg. the first cell of a singly linked list). These entry points are usually kept in program variables (on the stack), whereas the "inner" links of a pointer structure are anonymous addresses on the heap. Many algorithms deal with several substructures within one and the same store (eg. concatenation of two lists). Therefore we say that a pointer structure consists of a store together with a non-empty list of entry points. We choose lists rather than sets or bags of entry points, since in pointer algorithms both order and multiplicity of entries may be relevant, in particular when it comes to questions of sharing. For instance, the well-known algorithm that concatenates two acyclic singly-linked lists in situ (see Section 8.3) does not work when it attempts to concatenate a list to itself (it would create a circular list in this case).

Under these considerations we use $\mathcal{A}^+$, the set of all non-empty finite lists of elements of $\mathcal{A}$, as the set of *entries* to pointer structures.

Let now P denote the set of all stores for a given record type. Then a *pointer structure* is an element of $\mathcal{P} \stackrel{\text{def}}{=} \mathcal{A}^+ \times P$. For convenience we introduce the functions $ptr : \mathcal{P} \to \mathcal{A}^+$, $sto : \mathcal{P} \to P$ and $recs : \mathcal{P} \to \wp(\mathcal{A})$ defined by

$$ptr(s, S) \stackrel{\text{def}}{=} s \ , \qquad sto(s, S) \stackrel{\text{def}}{=} S \ , \qquad recs(s, S) \stackrel{\text{def}}{=} recs(S) \ .$$

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_i and store S.

4 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 function $\mathit{reach} : \mathcal{P} \to \wp(\mathcal{A})$ with

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

Here $\mathsf{set}\, s$ is the set of elements occurring in $s \in \mathcal{A}^+$. From this definition it is straightforward that

$$\mathit{reach}(n_1, \ldots, n_n, S) \ = \ \mathit{reach}(n_1, S) \cup \cdots \cup \mathit{reach}(n_n, S) \ . \tag{1}$$

Associated with *reach* is the *reachability relation* $\vdash : \mathcal{P} \leftrightarrow \wp(\mathcal{A}^+)$ given by

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

where $\mathsf{set}\, L = \bigcup_{s \in L} \mathsf{set}\, s$. 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

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

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.

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

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

i.e., the substructure that consists only of the records reachable from the entries s. The restriction is again taken componentwise, i.e., for all $k \in K$.

Suppose that $\neg\ \mathit{sharing}(n_1, n_2, S)$ holds. Then the parts $\mathit{from}(n_1, S)$ and $\mathit{from}(n_2, S)$ of the store are independent of each other, so that assignments to one part cannot influence the other one. Therefore the function *from* is of essential importance for localising side effects.

5 Overwriting Pointer Structures

To describe selective updating, we use the operation of *overwriting*. Given relations $R, S : M \leftrightarrow N$, the relation $R \mid S : M \leftrightarrow N$ (pronounced "R onto S") is given by

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

where $\overline{\mathsf{dom}\ R}$ is the complement of $\mathsf{dom}\ R$. Hence $R \mid S$ results from S by changing the image sets according to the prescription of R (if any). For example, if S is a function then $\{(x,y)\} \mid S$ "updates" S to make y the value corresponding to x.

The set $M \leftrightarrow N$ of binary relations forms a monoid under $\mid$ with $\emptyset$ as its neutral element. Moreover, the set of $M \rightsquigarrow N$ of partial functions is a submonoid of $M \leftrightarrow N$. For further properties see Möller (1993b).

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

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

The following lemma is important in localising the effects of an overwrite operation:

Lemma 5.1 $p \not\Vdash recs(S) \Rightarrow reach(S \mid p) = reach(p)$.

This means that overwriting outside the reachable part of a pointer structure does not affect reachability in that part. This will be most useful in connection with reasonable abstraction functions (cf. Section 7.4).

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

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

Let now $\mathcal{N} \stackrel{\text{def}}{=} \bigcup_{j \in K} \mathcal{N}_j$. To ease the notation and to keep with traditional programming languages, we introduce an operation

$$\_.\_ := \_ : \mathcal{P} \times K \times (\mathcal{P} \cup \mathcal{N}) \to \mathcal{P}$$

for *selective updating*:

$$\begin{array}{rcll} (n,S).k := (m,T) & \stackrel{\text{def}}{=} & (n, (n \stackrel{k}{\mapsto} m) \mid T) & \text{if } k \text{ has functionality } \mathcal{A} \to \mathcal{A}, \\ (n,S).k := x & \stackrel{\text{def}}{=} & (n, (n \stackrel{k}{\mapsto} x) \mid S) & \text{if } k \text{ has functionality } \mathcal{A} \to \mathcal{N}_j \\ & & & \text{and } x \in \mathcal{N}_j. \end{array}$$

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

$$\begin{array}{rcll} (n,S).k & \stackrel{\text{def}}{=} & (S_k(n), S) & \text{if } k \text{ has functionality } \mathcal{A} \to \mathcal{A} \text{ and } S_k(n) \text{ is defined}, \\ (n,S).k & \stackrel{\text{def}}{=} & S_k(n) & \text{if } k \text{ has functionality } \mathcal{A} \to \mathcal{N}_j. \end{array}$$

Otherwise, $(n,S).k$ is undefined. When using selections as arguments, undefinedness is assumed to propagate, according to the strictness of relational semantics. We have the following properties:

Lemma 5.2 Let $m \stackrel{\text{def}}{=} ptr(p)$, $n \stackrel{\text{def}}{=} ptr(q)$ and $r \stackrel{\text{def}}{=} (m, sto(q))$. Moreover, assume the selector functionalities $j : \mathcal{A} \to \mathcal{N}_j$ and $k, k_1, k_2 : \mathcal{A} \to \mathcal{A}$.

1. $ptr(p.k := q) = m$.
2. $(p.k := q).k = (m \stackrel{k}{\mapsto} n) \mid q$.
3. $k_1 \neq k_2 \Rightarrow (p.k_1 := q).k_2 = (m \stackrel{k_1}{\mapsto} n) \mid r.k_2$.
4. $(p.k := p.k) = p$.
5. $(p.k := q).j = r.j$.
6. $q \not\models m \Rightarrow from((p.k := q).k) = from(q)$.
7. $k_1 \neq k_2 \wedge r.k_2 \not\models n \Rightarrow from((p.k_1 := q).k_2) = from(r.k_2)$.
8. $q \not\models L \Rightarrow q \notin \mathsf{set}\, L \wedge q.k \not\models L$.
9. $\neg\, sharing(m, n, S) \Rightarrow \neg\, sharing(S_k(m), n, S)$.
10. $noreach(p) \subseteq noreach(p.k)$, i.e., p *norea* $p.k$.
11. $noreach(p.k := q) = noreach(\overline{\{m\}} \bowtie r)$.

6 Acyclic Stores and Forests

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 characterisation 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. If $T \subseteq S$ and S is acyclic or injective, then so is T, since all operations involved in the characterisations 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 have the following separation properties for acyclic stores (and hence forests) which will allow localisation of side effects:

Lemma 6.1 1. Let S be injective. Then for all $x, y \in \mathcal{A}$ we have

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

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
   $\forall\ z \in \mathcal{A} : \neg\ \mathit{sharing}(z, x, S)\ \Rightarrow\ \neg\ \mathit{sharing}(z, y, S)$.

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 $\neg\ \mathit{sharing}(x, y, S)$.

5. Let S be a forest and assume $y \in [S](x)$. Then

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

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 $\mathit{from}(n, S)$ is acyclic, injective or a forest, respectively.

7 Pointer Implementations

7.1 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 $F : \mathcal{A} \times P \rightsquigarrow \mathcal{O}$ such that F is surjective. 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 $F(n_1, \ldots, n_k, S) \stackrel{\text{def}}{=} F(n_1, S), \ldots, F(n_k, S)$.

7.2 Implementation of Operations

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

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

$$g_p\ \mathit{OPOI}\ g \stackrel{\text{def}}{\Leftrightarrow}\ g_p\ =\ F\,;g\ .$$

So the implementation g_p has to mimic the specification g faithfully. Note the implicit use of the extended abstraction function F (cf. Section 7.1) for the representation of tuples in $\mathcal{O}^n$.

For operations of functionality $\mathcal{O}^n \rightsquigarrow \mathcal{O}$ we are more liberal and allow the implementation to be non-deterministic, i.e., a relation rather than a function. This is reasonable, since different concrete objects may represent the same abstract object. A typical non-deterministic operation at the pointer level would be the allocation of new records (see Möller (1997)).

Our notion of implementation will be parameterised by additional requirements on the operation at the pointer level, such as preservation of certain aspects of the store. Such requirements are again

formulated as relations between "old" and "new" pointer structures. Hence our *parameterised implementation relation* has functionality $POI : (\mathcal{P} \leftrightarrow \mathcal{P}) \rightarrow ((\mathcal{P} \leftrightarrow \mathcal{P}) \leftrightarrow (\mathcal{O}^n \rightsquigarrow \mathcal{O}))$ and is defined by

$$f_p \; POI_{req} \; f \stackrel{\text{def}}{\Leftrightarrow} f_p ; F = F ; f \wedge f_p \subseteq req \; .$$

Here *req* is the additional requirement, examples of which will be given later.

The most liberal specification POI_{TRUE}, where $\text{TRUE} \stackrel{\text{def}}{=} \mathcal{P} \times \mathcal{P}$ is the universal relation, does not exclude indirect side-effects on parts of p that point into the reachable part $from(p)$. We also 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

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

It is the set of all records that are not reachable from the entries of p and hence should better be left alone by changes to the store of p. Note that

$$n \in noreach(p) \Leftrightarrow n \in recs(p) \wedge p \not\vdash n \; . \tag{2}$$

Now we can define two constraining relations $local : \mathcal{P} \leftrightarrow \mathcal{P}$ and $norea : \mathcal{P} \leftrightarrow \mathcal{P}$. First we set

$$p \; local \; q \stackrel{\text{def}}{\Leftrightarrow} L \bowtie sto(p) = L \bowtie sto(q) \; ,$$

where $L \stackrel{\text{def}}{=} noreach(p)$. So *local* requires that the part of the store that is unreachable in p is left untouched in q; by our definition this does not exclude adding new records to the store.

However, *local* 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 requiring *norea*, defined by

$$p \; norea \; q \stackrel{\text{def}}{\Leftrightarrow} noreach(p) \subseteq noreach(q) \; .$$

All three strengthenings POI_{local}, POI_{norea} or even $POI_{local \cap norea}$ of POI_{TRUE} still admit implementation by copying and by re-use.

Another interesting invariant is preservation of certain selections. For selector k we set

$$p \; pres_k \; q \stackrel{\text{def}}{\Leftrightarrow} sto(p)_k = (\mathsf{dom} \; sto(p)_k) \bowtie sto(q)_k \; .$$

This means that selection along k has the same effect in p and q, as long as only the "old" records in p are considered. Using domain restriction on q allows us to allocate new records in q, about which no assumptions are made by $pres_k$.

7.3 Development Strategy

Assume an abstraction function F. To calculate a pointer implementation $f_p : \mathcal{P} \leftrightarrow \mathcal{P}$ of $f : \mathcal{O}^n \rightsquigarrow \mathcal{O}$, we start with the expression $f(F(p))$, where p is an identifier of type $\mathcal{P}$, 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 f_p by setting $f_p(p) \stackrel{\text{def}}{=} E$ and are sure that $f_p \; POI_{\text{TRUE}} \; f$ 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.

7.4 Reasonable Abstraction Functions

To prepare the following discussion we need an auxiliary notion. Consider an arbitrary partial function $f : \mathcal{P} \rightsquigarrow M$ for some set M. As usual, f induces an equivalence relation $\sim_f$ on $\mathcal{P}$ by

$$p \sim_f q \stackrel{\text{def}}{\Leftrightarrow} f(p) = f(q) \; .$$

In particular, this equivalence identifies all elements of $\mathcal{P}$ for which f is undefined.

The pointer representation of an abstract object should solely be determined by the store and the entries to the structure; it should be independent of that part of the store that is not reachable from the entries. Therefore, generalising from the case of an abstraction function, we say that a function $f : \mathcal{P} \rightsquigarrow M$ on pointer structures is *reasonable* if

$$\forall \; p, q \in \mathcal{P} : \mathit{from}(p) = \mathit{from}(q) \; \Rightarrow \; p \sim_f q \; .$$

So pointer structures that agree in the part reachable from the entries represent the same abstract object (if any).

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.

Fortunately, there is a "canonical" class of abstraction functions that can be proved to be reasonable. These functions are defined by recursion following the links in the pointer structure while building up the abstract objects represented. This recursion pattern is typical of an *unfold operation* or *anamorphism* (see Meijer et al. (1991), Bird (1996)). We quote from Möller (1998):

Lemma 7.1 Assume a function F with the recursive definition

$$
\begin{array}{l}
F(p,x) = \mathsf{if}\ Q(p,x) \\
\qquad\quad \mathsf{then}\ E(p,x) \\
\qquad\quad \mathsf{else}\ C(\ F(f_1(p,x),p.s_1),\ldots,F(f_k(p,x),p.s_k), \\
\qquad\qquad\quad p.v_1,\ldots,p.v_m)
\end{array}
$$

where for all x the residual functions $Q(\_,x), E(\_,x), f_1(\_,x), \ldots f_k(\_,x)$ are reasonable. Then for all x the residual function $F(\_,x)$ is reasonable as well.

This lemma applies to all abstraction functions used in the present paper (and to many more, see again Möller (1998)). It even extends to abstraction *procedures* like printing out all nodes of a tree in infix order using indentation, such as the `C` routine

```
void print_btree ( btree b, int indentation )
  { if (b)
       { print_btree(b->right, indentation+3) ;
         printblanks(indentation) ;
         printf("%d\n",b->node) ;
         print_btree(b->left, indentation+3) ;
       }
  }
```

So our approach is directly usable for pointer programs in "real" programming languages.

For pointer implementations that use selective updating it usually is important that the updates work locally. This can be established using the following localisation properties that follow from Lemma 5.1 and Lemma 5.2:

Lemma 7.2 Assume that abstraction function F is reasonable and k has functionality $\mathcal{A} \to \mathcal{A}$.

1. $q \not\Vdash \mathit{recs}(S) \ \Rightarrow\ S \mid q \ \sim_F \ q$.
2. $q \not\Vdash \mathit{ptr}(p) \ \Rightarrow\ (p.k := q).k \ \sim_F \ q$.
3. Let $r \stackrel{\text{def}}{=} (\mathit{ptr}(p), \mathit{sto}(q))$ and assume $j \neq k$. Then

$$r.j \not\Vdash \mathit{ptr}(p) \ \Rightarrow\ (p.k := q).j \ \sim_F \ r.j\ .$$

8 Calculating with Acyclic Lists

8.1 Abstract Lists

The set $\mathcal{L}$ of *lists* with elements of $\mathcal{N}$ as node values is defined inductively, ie. as the least set $\mathcal{X}$ with

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

where ε denotes the empty list. A non-empty list, i.e., an element of $\mathcal{N} \times \mathcal{L}$, will be denoted as a pair $\langle x, l\rangle$ with head $x \in \mathcal{N}$ and tail $l \in \mathcal{L}$.

8.2 The Abstraction Function For Acyclic Structures

Let now P denote the set of all pointer structures over the record type for lists, as discussed in Section 3. The abstraction function $list : \mathcal{P} \rightsquigarrow \mathcal{L}$ constructs the list reachable from a record in a store. For $n \in \mathcal{A}$ we set

$$list(p) \stackrel{\text{def}}{=} \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \langle p.head, list(p.tail)\rangle\ .$$

In the case where a cycle is reachable from n in L, this recursion is non-terminating. In a strict underlying semantics this means that the value of $list(n, L)$ is undefined, whereas in a non-strict setting the value of $list(n, L)$ is an infinite list corresponding to an unwinding of the subgraph reachable from n in L. Since we are working in a relational setting, the strict interpretation is relevant here. So from now on we shall assume that *list* is used only for acyclic pointer structures. Below we will present an abstraction function that copes with cyclic lists. By Lemma 7.1, the abstraction function *list* is reasonable.

Moreover, we have

Lemma 8.1 An acyclic list pointer structure (m, L) with $m \in \mathcal{A}$ is a forest.

Proof: In the proof of Lemma 13 of Möller (1997) it was shown that

$$R\,;S \subseteq I \Rightarrow R^*\,;S^* \subseteq R^* \cup S^*\ .$$

Hence for a partial function R, characterised by $R\,;R^\smile \subseteq I$, we obtain from this, setting $S \stackrel{\text{def}}{=} R^\smile$, the property of downward local linearity of R^*:

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

Now assume $L_{tail}(k_1) = L_{tail}(k_2) = n$ for $k_1, k_2 \in reach(m, L)$ with $k_1 \neq k_2$. Then by $(\dagger)$ w.l.o.g. $k_1\ L_{tail}^+\ k_2$. Since L_{tail} is a partial function it follows that $k_1\ L_{tail}\ n\ L_{tail}^*\ k_2\ L_{tail}\ n$, i.e., $n\ L_{tail}^+\ n$, a contradiction to acyclicity of (m, L). ■

8.3 Concatenation

We now calculate pointer implementations of a number of sample operations. First we treat the operation $cat : \mathcal{L} \times \mathcal{L} \rightsquigarrow \mathcal{L}$ that concatenates two lists. It is recursively defined by

$$\begin{aligned} cat(\varepsilon, r) &= r\ , \\ cat(\langle x, l\rangle, r) &= \langle x, cat(l, r)\rangle\ . \end{aligned}$$

Using our scheme from Section 7 we specify a pointer implementation cat_p by requiring

$$cat_p\ POI_{\mathrm{TRUE}}\ cat$$

and want to find an explicit version of cat_p. However, we will develop this only for the case of arguments without sharing.

So assume that $p = (m, n, L)$ with list entries $m, n \in \mathcal{A}$ and store L is acyclic and that $\neg\ \mathit{sharing}(p)$ holds. Then the specification of cat_p unfolds into

$$\mathit{list}(\mathit{cat}_p(m, n, L)) = \mathit{cat}(\mathit{list}(m, L), \mathit{list}(n, L)) \ .$$

If $m = \diamond$ we have $\mathit{list}(m, L) = \varepsilon$ and hence $\mathit{list}(\mathit{cat}_p(m, n, L)) = \mathit{list}(n, L)$, so that we may choose

$$\mathit{cat}_p(\diamond, n, L) = (n, L) \ .$$

For $m \neq \diamond$ we calculate, with $p = (m, L)$ and $q = (n, L)$,

$\mathit{cat}(\mathit{list}(p), \mathit{list}(q))$

$=$ {⟦ unfold definition of *list* ⟧}

$\mathit{cat}(\langle p.\mathit{head}, \mathit{list}(p.\mathit{tail})\rangle, \mathit{list}(q))$

$=$ {⟦ unfold definition of *cat* ⟧}

$\langle p.\mathit{head}, \mathit{cat}(\mathit{list}(p.\mathit{tail}), \mathit{list}(q))\rangle$

$=$ {⟦ fold with specification of cat_p, i.e., choose an arbitrary $q' \in \mathit{cat}_p(L_{\mathit{tail}}(m), n, L)$ ⟧}

$\langle p.\mathit{head}, \mathit{list}(q')\rangle$

$=$ {⟦ setting $r \stackrel{\text{def}}{=} p.\mathit{tail} := q'$ and using Lemma 5.2.5, assuming additionally $\mathit{cat}_p \subseteq \mathit{pres}_{\mathit{head}}$ ⟧}

$\langle r.\mathit{head}, \mathit{list}(q')\rangle$

$=$ {⟦ by Lemmas 7.2.2, 6.1.2 and 8.1, assuming additionally $\mathit{cat}_p \subseteq \mathit{norea}$ ⟧}

$\langle r.\mathit{head}, \mathit{list}(r.\mathit{tail})\rangle$

$=$ {⟦ fold with definition of *list* ⟧}

$\mathit{list}(r)$.

For the correctness of the folding step we observe that the assumption $\neg\ \mathit{sharing}$ propagates to the recursive call by Lemma 5.2.9. The derivation has shown the need for introducing the additional invariants $\mathit{pres}_{\mathit{head}}$ and norea; they are established by the termination case and propagate to r by a straightforward calculation using (1).

Altogether, for the following recursion we have $\mathit{cat}_p\ \mathit{POI}_{\mathit{pres}_{\mathit{head}} \cap \mathit{norea}}\ \mathit{cat}$:

$$\mathit{cat}_p(m, n, L) = \mathsf{if}\ m = \diamond\ \mathsf{then}\ (n, L)\ \mathsf{else}\ p.\mathit{tail} := \mathit{cat}_p(L_{\mathit{tail}}(m), n, L) \ .$$

8.4 Reversal

Next we want to derive a function for reversing a list. As reversal function $rev : \mathcal{L} \rightsquigarrow \mathcal{L}$ on abstract lists we use an embedding into a tail-recursive function $rrev : \mathcal{L} \times \mathcal{L} \rightsquigarrow \mathcal{L}$:

$$\begin{array}{rcl} rev(l) & \stackrel{\text{def}}{=} & rrev(l, \varepsilon), \\ rrev(\varepsilon, t) & \stackrel{\text{def}}{=} & t\ , \\ rrev(\langle m, l\rangle, t) & \stackrel{\text{def}}{=} & rrev(l, \langle m, t\rangle)\ . \end{array}$$

The additional parameter t of $rrev$ accumulates the intermediate results. Using our scheme from Section 7 we specify a pointer implementation rev_p by requiring $rev_p\ POI_{\mathrm{TRUE}}\ rev$ and want to find an explicit version of rev_p. We do this by finding a pointer implementation of the auxiliary function $rrev$. However, we do not carry the accumulating substructure itself as a parameter, but just its head cell. Hence we specify, assuming $\neg\ sharing(m, n, L)$:

$$list(rrev_p(m, n, L)) \;=\; rrev(list(m, L), list(n, L))\ .$$

Since $list(\diamond, L) = \varepsilon$, we can use $rev_p(m, L) = rrev_p(m, \diamond, L)$ as an appropriate embedding. As before, we now perform a case analysis.
Case 1: $m = \diamond$. Then $list(m, L) = \varepsilon$, and hence $rev(list(m, L)) = \varepsilon$. Thus $list(rrev_p(m, n, L)) = list(n, L)$, so that we may choose $rrev_p(m, n, L) = (n, L)$ in this case.
Case 2: For $m \neq \diamond$ we calculate, with $p = (m, L)$ and $q = (n, L)$,

$$\begin{array}{ll} & list(rrev_p(m, n, L)) \\ = & \quad \{\![\ \text{unfold specification of } rrev_p\ ]\!\} \\ & rrev(list(p), list(q)) \\ = & \quad \{\![\ \text{unfold definition of } list\ ]\!\} \\ & rrev(\langle p.head, list(p.tail)\rangle, list(q)) \\ = & \quad \{\![\ \text{unfold definition of } rrev\ ]\!\} \\ & rrev(list(p.tail), \langle p.head, list(q)\rangle) \\ = & \quad \{\![\ \text{setting } r \stackrel{\text{def}}{=} p.tail := q \text{ and using Lemmas 5.2.5 and 7.2.3} \\ & \qquad \text{by } \neg\ sharing(m, n, L)\ ]\!\} \\ & rrev(list(p.tail), \langle r.head, list(r.tail)\rangle) \\ = & \quad \{\![\ \text{fold with definition of } list\ ]\!\} \\ & rrev(list(p.tail), list(r)) \\ = & \quad \{\![\ \text{definition of } p\ ]\!\} \\ & rrev(list(L_{tail}(m), L), list(r)) \\ = & \quad \{\![\ \text{by acyclicity of } p \text{ and Lemmas 6.1.2 and 7.2.1}\ ]\!\} \end{array}$$

$$
\begin{aligned}
& \mathit{rrev}(\mathit{list}(L_{\mathit{tail}}(m), \mathit{sto}(r)), \mathit{list}(r)) \\
= \quad & \{\!\![\text{ fold with specification of } \mathit{rrev}_p \text{ }]\!\!\} \\
& \mathit{list}(\mathit{rrev}_p(L_{\mathit{tail}}(m), r)) \ .
\end{aligned}
$$

Hence we may choose $\mathit{rrev}_p(m, n, L) = \mathit{rrev}_p(L_{\mathit{tail}}(m), r)$ in this case. Altogether, we have obtained

$$
\begin{aligned}
& \mathit{rev}_p(m, L) = \mathit{rrev}_p(m, \diamond, L) \ , \\
& \mathit{rrev}_p(m, n, L) = \mathsf{if}\ m = \diamond\, \mathsf{then}\, (n, L) \\
& \qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{let}\ k = L_{\mathit{tail}}(m) \\
& \qquad\qquad\qquad\qquad\qquad\ \ r = p.\mathit{tail} := q \\
& \qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathit{rrev}_p(k, r) \quad .
\end{aligned}
$$

The algorithm exploits that redirection of the head pointer of a list does not influence the tail of that list. In this case no strengthening of the invariant TRUE was necessary.

Again, we have to check the validity of the assertion for the recursive call. Assume that $m \neq \diamond \wedge \neg\ \mathit{sharing}(m, n, L)$ hold, and set $k \stackrel{\text{def}}{=} L_{\mathit{tail}}(m)$ and $M \stackrel{\text{def}}{=} (m \stackrel{\mathit{tail}}{\mapsto} n) \mid L$. Then

$$\mathit{reach}(m, M) = \{m\} \cup \mathit{reach}(n, M) = \{m\} \cup \mathit{reach}(n, L)$$

by Lemma 5.1 and $\neg\ \mathit{sharing}(m, n, L)$. Moreover,

$$\mathit{reach}(k, M) = \mathit{reach}(k, L) \ ,$$

again by Lemma 5.1. Now $\neg\ \mathit{sharing}(k, m, M)$ is immediate using elementary set theory, acyclicity of (m, L), Lemma 6.1.2 and Lemma 5.2.9.

8.5 Comparison of the Derivations

One may wonder why we needed to introduce additional requirements in the seemingly simpler case of the concatenation algorithm. The reason for this is that a non-tail recursion arose. The operations performed after the recursive call need some assertions about the connection between arguments and results of the recursion.

Contrarily, the reversal algorithm shows a tail-recursive pattern and all the calculation is performed on the arguments of the recursive calls. So we can check the required assertions directly on the arguments without the need of additional requirements between arguments and results.

9 The Cyclic Case

9.1 The Abstraction Function for Circular Lists

We now treat the case of circular lists. We now say that a pointer structure (m, L) represents the list which is obtained by following the links until an already visited record is reached. The

corresponding abstraction function is $clist : P \rightsquigarrow \mathcal{L}$. For $m \in \mathcal{A}$ with $m = \diamond$ or cyclic (m, L) we set

$$clist(p) \stackrel{\text{def}}{=} \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ \langle p.head, clis(p.tail, \{ptr(p)\})\rangle\ ,$$

where the auxiliary function $clis : \mathcal{P} \times \wp(\mathcal{A}) \rightsquigarrow \mathcal{L}$ is given by

$$\begin{array}{l} clis(p, V) \stackrel{\text{def}}{=} \mathsf{if}\ ptr(p) \in V \cup \{\diamond\} \\ \qquad\qquad \mathsf{then}\ \varepsilon \\ \qquad\qquad \mathsf{else}\ \langle p.head, clis(p.tail, V \cup \{ptr(p)\})\rangle\ . \end{array}$$

Termination is now forced by the additional argument V of *clis* which remembers the set of already visited records. Again we have an anamorphic recursion pattern. First we show the following reasonableness properties for *clis*:

Lemma 9.1 1. For all $V \subseteq \mathcal{A}$ the residual function $clis(\_, V)$ is reasonable.

2. $ptr(p) = ptr(q) \wedge \overline{V} \bowtie p = \overline{V} \bowtie q \Rightarrow clis(p, V) = clis(q, V)$.

Proof:

1. Immediate from Lemma 7.1.

2. We use fixpoint induction with the continuous predicate

$$\begin{array}{l} PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\ p_1, p_2, V : ptr(p_1) = ptr(p_2) \wedge \\ \qquad \overline{V} \bowtie p_1 = \overline{V} \bowtie p_2 \Rightarrow h(p_1, V) = h(p, V)\ . \end{array}$$

The induction basis $PP(\emptyset)$ is trivial. Assume now $PP(h)$. For the induction step we assume the premise of PP and set $m \stackrel{\text{def}}{=} ptr(p_1) = ptr(p_2)$. Then

$$reach(p_i)\backslash V = \{m\}\backslash V \cup reach(p_i.tail)\backslash V\ .$$

Case 1: $m \in V$. By definition $\tau(h)(p_1, V) = \varepsilon = \tau(h)(p_2, V)$.
Case 2: $m \notin V$. Then by assumption $\{m\} \bowtie p_1 = \{m\} \bowtie p_2$ and hence $p_1.head = p_2.head$ and $ptr(p_1.tail) = ptr(p_2.tail)$. Moreover, since $V \subseteq V \cup \{m\}$, we have $\overline{V \cup \{m\}} \bowtie p_1.tail = \overline{V \cup \{m\}} \bowtie p_2.tail$. So the induction hypothesis is satisfied and we have

$$\begin{array}{l} \tau(h)(p_1, V) = \langle p_1.head, clis(p_1.tail, V \cup \{m\})\rangle = \\ \langle p_2.head, clis(p_2.tail, V \cup \{m\})\rangle = \tau(h)(p_2, V)\ . \end{array}$$

■

One can even show the stronger property

$$\begin{array}{l} ptr(p) = ptr(q) \wedge (reach(p)\backslash V) \bowtie p = (reach(q)\backslash V) \bowtie q \Rightarrow \\ clis(p, V) = clis(q, V)\ . \end{array}$$

However, its premise is much harder to check than the one of 2 above, so that it is less useful. From 1 it is immediate that

Lemma 9.2 The abstraction function *clist* is reasonable.

The function *clis* also shows an important localisation property:

Lemma 9.3 $p \not\models W \Rightarrow clis(p, V \cup W) = clis(p, V)$.

Proof: We use again fixpoint induction with the continuous predicate

$$PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\, p, V, W : p \not\models W = \emptyset \Rightarrow h(p, V \cup W) = h(p, V)\ .$$

By Lemma 5.2.8 the premise of $PP(h)$ implies

$$ptr(p) \notin W \ \wedge\ p.tail \not\models W\ . \qquad (*)$$

The induction basis $PP(\emptyset)$ is trivial. Assume now $PP(h)$. With τ as in the previous proof we calculate

$$\begin{array}{ll} & \tau(h)(p, V \cup W) \\ = & \quad \{\![\text{ definition of } \tau \ ]\!\} \\ & \mathsf{if}\ ptr(p) \in V \cup W \\ & \quad \mathsf{then}\ \varepsilon \\ & \quad \mathsf{else}\ \langle p.head, clis(p.tail, V \cup W \cup \{ptr(p)\})\rangle \\ = & \quad \{\![\text{ by } (*) \text{ and } PP(h) \ ]\!\} \\ & \mathsf{if}\ ptr(p) \in V \\ & \quad \mathsf{then}\ \varepsilon \\ & \quad \mathsf{else}\ \langle p.head, clis(p.tail, V \cup \{ptr(p)\})\rangle \\ = & \quad \{\![\text{ definition of } \tau \ ]\!\} \\ & \tau(h)(p, V)\ . \end{array}$$

■

Finally we note that the representation of singleton lists is almost unique:

Lemma 9.4 $clist(q) = \langle x, \varepsilon\rangle \ \Leftrightarrow\ ptr(q) = ptr(q.tail) \wedge q.head = x$.

9.2 Concatenation

Again we require $cat_p\ POI_{\mathrm{TRUE}}\ cat$, but this time w.r.t. *clist*, and want to find an explicit version of cat_p for the case of arguments without sharing.

Assume that $p = (m, L)$ and $q = (n, L)$ both represent circular lists and that $\neg\ sharing(m, n, L)$ holds. In particular then $m \neq n$. The specification of cat_p now unfolds into

$$clist(cat_p(m, n, L)) = cat(clist(p), clist(q))\ .$$

If $m = \diamond$, we have $clist(p) = \varepsilon$ and hence $clist(cat_p(m, n, L)) = clist(n, L)$, so that we may choose

$$cat_p(\diamond, n, L) = (n, L) \ .$$

For $m \neq \diamond$ we have to consider *clis*. We introduce an auxiliary function ca_p, also with an additional parameter, that mirrors the reduction of *clist* to *clis*. It is specified by

$$clis(ca_p(m, n, L, V), V) = cat(clis(m, L, V), clis(n, L, V)) \ .$$

To obtain a directly recursive version of ca_p we again employ a case analysis.
Case 1: $m \in V$. Then $clis(m, L, V) = \varepsilon$. Hence $clis(ca_p(m, n, L, V), V) = clis(n, L, V)$, so that we may choose $ca_p(m, n, L, V) = (n, L)$ in this case.
Case 2: $m \notin V$. Set $p = (m, L)$ and $q = (n, L)$. We calculate

$$\begin{aligned}
& cat(clis(p, V), clis(q, V)) \\
= \quad & \{\![\text{ unfold definition of } clis \ ]\!\} \\
& cat(\langle p.head, clis(p.tail, V \cup \{m\})\rangle, clis(q, V)) \\
= \quad & \{\![\text{ unfold definition of } cat \ ]\!\} \\
& \langle p.head, cat(clis(p.tail, V \cup \{m\}), clis(q, V))\rangle \\
= \quad & \{\![\text{ by Lemma 9.3 }]\!\} \\
& \langle p.head, cat(clis(p.tail, V \cup \{m\}), clis(q, V \cup \{m\}))\rangle \\
= \quad & \{\![\text{ fold with specification of } ca_p \text{ setting } l \stackrel{\text{def}}{=} ptr(p.tail) \ ]\!\} \\
& \langle p.head, clis(ca_p(l, n, L, V \cup \{m\}), V \cup \{m\})\rangle \\
= \quad & \{\![\text{ fold with specification of } ca_p, \text{ i.e., choose an arbitrary} \\
& \quad q' \in ca_p(l, n, L, V \cup \{m\}) \ ]\!\} \\
& \langle p.head, clis(q')\rangle \\
= \quad & \{\![\text{ setting } r \stackrel{\text{def}}{=} p.tail := q' \text{ and using Lemma 5.2.5,} \\
& \quad \text{assuming additionally } ca_p \subseteq pres_{head} \ ]\!\} \\
& \langle r.head, clis(q', V \cup \{m\}))\rangle \\
= \quad & \{\![\text{ fold with definition of } clis, \text{ since } ptr(r) = m \\
& \quad \text{by Lemma 5.2.1, and using Lemma 9.1.2 }]\!\} \\
& clis(r, V) \ .
\end{aligned}$$

Again the assumption $\neg$ *sharing* propagates to the recursive call by Lemma 5.2.9. Altogether we have obtained

$$\begin{aligned}
cat_p(m, n, L) = \ & \mathsf{if}\ m = \diamond\ \mathsf{then}\ (n, L) \\
& \quad \mathsf{else}\ p.tail := ca_p(L_{tail}(m), n, L, \{m\})
\end{aligned}$$

$$ca_p(m,n,L,V) = \mathsf{if}\ m \in V\ \mathsf{then}\ (n,L)$$
$$\mathsf{else}\ p.tail := ca_p(L_{tail}(m),n,L,V \cup \{m\})$$

Note the close correspondence between the derivations in the acyclic and in the cyclic case.

9.3 Reversal

Assume now the specification

$$clist(rev_p(p)) = rev(clist(p))\ .$$

We have

$$\begin{array}{ll} & rev(clist(p)) \\ = & \quad \{\![\ \text{definitions of } clist \text{ and } rev\ ]\!\} \\ & \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ \varepsilon \\ & \qquad \mathsf{else}\ rrev(clis(p.tail,\{ptr(p)\}),\langle p.head,\varepsilon\rangle)\ . \\ = & \quad \{\![\ \text{definition of } clist \text{ and Lemma 9.4}\ ]\!\} \\ & \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ clist(p) \\ & \qquad \mathsf{else}\ rrev(clis(p.tail,\{ptr(p)\}),clist(p.tail := p))\ . \end{array}$$

This calls for the introduction of an auxiliary function $rrev_p$ specified by

$$clis(rrev_p(m,n,L,V),V) = rrev(clis(m,L,V),clis(n,L,V))$$

provided $\neg\ sharing(m,n,L)$. Set $p \stackrel{\text{def}}{=} (m,L)$ and $q \stackrel{\text{def}}{=} (n,L)$. To obtain a directly recursive version of $rrev_p$ we again employ a case analysis.
Case 1: $m \in V$. Then $rrev(clis(p,V),clis(q,V)) = clis(q,V)$, and we may set $rrev_p(m,n,L,V) = q$ in this case.
Case 2: $m \notin V$. We calculate

$$\begin{array}{ll} & rrev(clis(p,V),clis(q,V)) \\ = & \quad \{\![\ \text{unfold definition of } clis\ ]\!\} \\ & rrev(\langle p.head, clis(p.tail, V \cup \{m\})\rangle, clis(q,V)) \\ = & \quad \{\![\ \text{unfold definition of } rrev\ ]\!\} \\ & rrev(clis(p.tail, V \cup \{m\}), \langle p.head, clis(q,V)\rangle) \\ = & \quad \{\![\ \text{by Lemma 9.3}\ ]\!\} \\ & rrev(clis(p.tail, V \cup \{m\}), \langle p.head, clis(q, V \cup \{m\})\rangle) \\ = & \quad \{\![\ \text{setting } r \stackrel{\text{def}}{=} p.tail := q \text{ and using Lemmas 5.2.5 and 7.2.3} \end{array}$$

$$
\begin{array}{ll}
 & \quad \text{by } \neg\ \mathit{sharing}(m,n,L)\ \rrbracket \\
 & \mathit{rrev}(\mathit{clis}(p.\mathit{tail}, V \cup \{m\}), \langle r.\mathit{head}, \mathit{clis}(r.\mathit{tail}, V \cup \{m\})\rangle) \\
= & \quad \{\!\![\text{ fold with definition of } \mathit{clis} \ \rrbracket \\
 & \mathit{rrev}(\mathit{clis}(p.\mathit{tail}, V \cup \{m\}), \mathit{clis}(r, V)) \\
= & \quad \{\!\![\text{ by Lemma 9.3 using } \neg\ \mathit{sharing}(m,n,L)\ \rrbracket \\
 & \mathit{rrev}(\mathit{clis}(p.\mathit{tail}, V \cup \{m\}), \mathit{clis}(r, V \cup \{m\})) \\
= & \quad \{\!\![\text{ definition of } p \ \rrbracket \\
 & \mathit{rrev}(\mathit{clis}(L_{\mathit{tail}}(m), L, V \cup \{m\}), \mathit{clis}(r, V \cup \{m\})) \\
= & \quad \{\!\![\text{ by Lemma 9.1.2 } \rrbracket \\
 & \mathit{rrev}(\mathit{clis}(L_{\mathit{tail}}(m), \mathit{sto}(r), V \cup \{m\}), \mathit{clis}(r, V \cup \{m\})) \\
= & \quad \{\!\![\text{ fold with specification of } \mathit{rrev}_p \ \rrbracket \\
 & \mathit{clis}(\mathit{rrev}_p(L_{\mathit{tail}}(m), r, V \cup \{m\}))\ .
\end{array}
$$

Hence we may choose

$$\mathit{rrev}_p(m,n,L,V) \;=\; \mathit{rrev}_p(L_{\mathit{tail}}(m), r, V \cup \{m\})$$

in this case. Altogether, we obtain, using again Lemma 9.4 and Lemma 9.1.2,

$$
\begin{array}{l}
\mathit{rev}_p(m,L) = \mathsf{if}\ m = \Diamond\ \mathsf{then}\ (m,L) \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathit{rrev}_p(L_{\mathit{tail}}(m), m, (m \overset{\mathit{tail}}{\mapsto} m) \mid L, \{m\}) \\
\mathit{rrev}_p(m,n,L,V) = \mathsf{if}\ m \in V\ \mathsf{then}\ (n,L) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{let}\ k = L_{\mathit{tail}}(m) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad r = p.\mathit{tail} := q \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{in}\ \ \mathit{rrev}_p(k, r, V \cup \{m\}) \quad .
\end{array}
$$

Again, the derivation was very similar to the one for the acyclic case.

10 Conclusion

The relational calculus has proved to be a very useful tool for modelling and analysing pointer structures. 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 extension to properly cyclic structures has proved to be relatively simple and did not need additional concepts.

Acknowledgements I am grateful to R. Berghammer, T. Ehm and O. de Moor for a number of valuable comments. This research was partially sponsored by Esprit Working Group 8533 *NADA — New Hardware Design Methods*.

References

[1] 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

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

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

[4] 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

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

[6] 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

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

[8] B. Möller: Calculating with pointer structures. In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP TC2/WG2.1 Working Conference, Le Bischenberg, Feb. 1997. Chapman&Hall 1997, 24–48

[9] B. Möller: Are anamorphisms reasonable abstractions? Proc. Workshop on Generic Programming, Marstrand, 18 June, 1998. Chalmers University of Technology, Göteborg, 1998 (11 pp.)

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