Are Anamorphisms Reasonable Abstractions?

Bernhard Möller

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

Abstract. In calculational derivations of pointer algorithms the concept of a reasonable abstraction function has proved to be of central importance. A function from pointer structures to some other domain is called *reasonable* if it only depends on the reachable part of the store. For reasonable functions we can reduce questions about invariance of certain parts of objects to an analysis of (non-)reachability in the pointer structure. In this way we can prove a number of transformation laws once and for all for all types of pointer structures. In the present paper we show that all abstraction functions with an anamorphic recursive definition are reasonable, so that our laws apply for them. The approach copes also with cyclic strctures; among others, we give abstraction functions for cyclic lists and threaded trees. By tuning the degree of abstraction we can give simple specifications for routines that are to update pointer structures in situ.

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 (1997a,1997b).

A central concept there is that of a reasonable abstraction function. A function from pointer structures to some other domain is called *reasonable* if it only depends on the reachable part of the store. For reasonable functions we can reduce questions about invariance of certain parts of objects to an analysis of (non-)reachability in the pointer structure. In this way we can prove a number of transformation laws once and for all for all types of pointer structures. In the present paper we show that all abstraction functions with an anamorphic recursive definition are reasonable. This was already conjectured, but not proved, in Möller (1997a).

The approach copes also with cyclic structures; among others, we give abstraction functions for cyclic lists, doubly-linked lists and infix-threaded binary trees. By tuning the degree of abstraction we can give simple specifications for routines that are to update pointer structures in situ.

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 analyzing the reachable part of a pointer structure we shall use the *domain restriction* of R to a subset $L \subseteq M$ given by $L \bowtie R \stackrel{\text{def}}{=} R \cap (L \times N)$. 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* if $R^{\smile}; R \subseteq I_N$. 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

3.1 The Model

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 functions 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 $\mathcal{A} \to \mathcal{N}_j$ and
3. $\diamond \notin \mathit{recs}(S) \stackrel{\text{def}}{=} \bigcup_{k \in K} \mathsf{dom}S_k$, the set of records allocated in S.

A store may be viewed as a labeled directed graph: the nodes are the records 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 the uniqueness of selection in records. 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 examples of record schemes will be those for singly linked lists and labelled binary trees. For both, assume a set $\mathcal{N}$ of node values.

In the case of lists we use two selectors *head*, *tail* of types $head : \mathcal{A} \to \mathcal{N}$ and $tail : \mathcal{A} \to \mathcal{A}$. Then a list store L consists of two partial functions $L_{head} : \mathcal{A} \rightsquigarrow \mathcal{N}$ and $L_{tail} : \mathcal{A} \rightsquigarrow \mathcal{A}$, where L_{head} returns the node value and L_{tail} gives the next record in the list.

For binary trees, our selectors are $l, r : \mathcal{A} \to \mathcal{A}$ and $v : \mathcal{A} \to \mathcal{N}$. Then a binary tree store BT consists of functions $BT_l, BT_r : \mathcal{A} \rightsquigarrow \mathcal{A}$ and $BT_v : \mathcal{A} \rightsquigarrow \mathcal{N}$.

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 type $\mathcal{A} \to \mathcal{A}$. 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 $[L] \stackrel{\text{def}}{=} L_{tail}$.

Let now P 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 $\mathcal{P} \stackrel{\text{def}}{=} A^+ \times P$. For convenience we introduce the functions $ptr : \mathcal{P} \to \mathcal{A}^+$, $sto : \mathcal{P} \to P$ and $recs : \mathcal{P} \to \wp(A)$ 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. For abbreviation we set $[(s, S)] \stackrel{\text{def}}{=} [S]$.

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

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

Otherwise, $(n, S).k$ is undefined.

3.2 Reachability

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

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

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

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

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 the entries s by the restriction

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

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 $k \in K$.

We have the following properties (for the proofs see the Appendix):

Lemma 31 *$\mathit{from}(p) = \mathit{from}(q) \Rightarrow \forall\ U \subseteq \mathit{reach}(p) : [p]^*(U) = [q]^*(U)$.*

Corollary 32 *If s_i has type $\mathcal{A} \to \mathcal{A}$ then $\mathit{from}(p) = \mathit{from}(q) \Rightarrow \mathit{from}(p.s_i) = \mathit{from}(q.s_i)$.*

Corollary 33 *$\mathit{from}(p) = \mathit{from}(q) \Rightarrow \forall\ n \in \mathit{reach}(p) : \mathit{from}(n, \mathit{sto}(p)) = \mathit{from}(n, \mathit{sto}(q))$.*

4 Reasonable Abstraction Functions

4.1 Definition

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

Consider now 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

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

The pointer representation of an abstract object should be essentially determined by the entries to the structure. Therefore, generalizing from the case of an abstraction function, we say that a function $f : \mathcal{P} \rightsquigarrow M$ on pointer structures is *reasonable* if

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

Relationally this amounts to

$$\mathit{from}\,;\mathit{from}^{\smile} \;\subseteq\; \sim_f \;\;.$$

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.

4.2 Anamorphisms *Are* Reasonable Abstractions

We give now a general criterion for reasonableness of recursively defined abstraction functions. The recursion pattern considered is typical of an *unfold operation* or *anamorphism* (see Meijer et al. (1991), Bird (1996)).

Lemma 41 *Assume a function F with the recursive definition*

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

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.

Proof. We use fixpoint induction on the recursive definition of F and the continuous predicate

$$PP(h) \stackrel{\text{def}}{\Leftrightarrow} \forall\, x,p,q : \mathit{from}(p) = \mathit{from}(q) \;\Rightarrow\; h(x,p) \;=\; h(x,q)\;.$$

The functional τ associated with the recursive definition of F is

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

The induction basis $PP(\emptyset)$ is trivial. Assume now $PP(h)$. First, we observe that $\mathit{from}(p) = \mathit{from}(q)$ implies by Corollary 32

$$\begin{array}{ll} \mathit{ptr}(p) \;=\; \mathit{ptr}(q)\;, & \\ p.v_j \;=\; q.v_j \;\;\text{for all selectors } v_j \text{ of type } \mathcal{A} \to \mathcal{N}\;, & (*) \\ \mathit{from}(p.s_i) \;=\; \mathit{from}(q.s_i) \;\;\text{for all selectors } s_i \text{ of type } \mathcal{A} \to \mathcal{A}\;. & \end{array}$$

Hence

$$
\begin{array}{ll}
 & \tau(h)(x,p) \\
= & \quad \{\![\text{ definition of } \tau \ ]\!\} \\
 & \mathsf{if}\ Q(x,p) \\
 & \quad \mathsf{then}\, E(x,p) \\
 & \quad \mathsf{else}\ C(h(f_1(x,p),p.s_1),\ldots h(f_k(x,p),p.s_k),p.v_1,\ldots,p.v_m) \\
= & \quad \{\![\text{ by } (*) \text{ and } PP(h) \ ]\!\} \\
 & \mathsf{if}\ Q(x,p) \\
 & \quad \mathsf{then}\, E(x,p) \\
 & \quad \mathsf{else}\ C(h(f_1(x,p),q.s_1),\ldots h(f_k(x,p),q.s_k),q.v_1,\ldots,q.v_m) \\
= & \quad \{\![\text{ by } \mathit{from}(p) = \mathit{from}(q) \text{ and reasonableness} \\
 & \qquad \text{of } Q(x,\_), E(x,\_) \text{ and the } f_k(x,\_) \ ]\!\} \\
 & \mathsf{if}\ Q(x,p) \\
 & \quad \mathsf{then}\, E(x,p) \\
 & \quad \mathsf{else}\ C(h(f_1(x,q),q.s_1),\ldots h(f_k(x,q),q.s_k),q.v_1,\ldots,q.v_m) \\
= & \quad \{\![\text{ definition of } \tau \ ]\!\} \\
 & \tau(h)(x,q)\ .
\end{array}
$$

□

5 Examples

5.1 Acyclic Lists

Consider again the record scheme for lists with selectors $\mathit{head} : \mathcal{A} \to \mathcal{N}$ and $\mathit{tail} : \mathcal{A} \to \mathcal{A}$ for a set $\mathcal{N}$ of node values.

The set $\mathcal{L}$ of *singly linked lists* with elements of $\mathcal{N}$ as nodes is defined inductively 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 $l \in \mathcal{L}$.

A suitable abstraction function is

$$
\begin{array}{l}
\mathit{list}(p) \ = \ \mathsf{if}\ \mathit{ptr}(p) \ = \ \diamond \ \mathsf{then}\, \varepsilon \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \langle p.\mathit{head}, \mathit{list}(p.\mathit{tail}) \rangle
\end{array}
$$

It constructs the list reachable from a record in a store.

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 $\mathit{list}(n, L)$ is undefined, whereas in a non-strict setting the value of $\mathit{list}(n, L)$ 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.

According to our general criterion this abstraction function is reasonable, since its recursion is anamorphic (to match the pattern of Lemma 41 exactly, introduce an additional parameter x which is passed unchanged to the recursive call).

5.2 Cyclic Lists

We now treat the case of cyclic lists. We 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 $\mathit{clist} : P \rightsquigarrow \mathcal{L}$. For $m \in \mathcal{A}$ with $m = \diamond$ or cyclic (m, L) we set

$$\begin{aligned} \mathit{clist}(p) \;=\; &\mathsf{if}\ \mathit{ptr}(p) = \diamond\ \mathsf{then}\ \varepsilon \\ &\qquad \mathsf{else}\ \langle p.\mathit{head}, \mathit{clis}(\{\mathit{ptr}(p)\}, p.\mathit{tail})\rangle \end{aligned}$$

$$\begin{aligned} \mathit{clis}(V, p) \;=\; &\mathsf{if}\ \mathit{ptr}(p) \in V\ \mathsf{then}\ \varepsilon \\ &\qquad \mathsf{else}\ \langle p.\mathit{head}, \mathit{clis}(V \cup \{\mathit{ptr}(p)\}, p.\mathit{tail})\rangle \end{aligned}$$

Termination is now forced by the additional argument V of *clis* which remembers the set of already visited records. Again the recursion is anamorphic, so that $\mathit{clis}(V, \_)$ and *clist* are reasonable.

5.3 Binary Trees

The record scheme for labeled binary trees over set $\mathcal{N}$ of node values uses the selectors $l, r : \mathcal{A} \to \mathcal{A}$ and $v : \mathcal{A} \to \mathcal{N}$.

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 $r \in \mathcal{T}$.

Analogously to the case of acyclic lists, the abstraction function $\mathit{tree} : \mathcal{P} \to \mathcal{T}$ constructs the tree reachable from a record in a store:

$$\begin{aligned} \mathit{tree}(p) \;=\; &\mathsf{if}\ \mathit{ptr}(p) = \diamond\ \mathsf{then}\ \varepsilon \\ &\qquad \mathsf{else}\ \langle \mathit{tree}(p.l), p.v, \mathit{tree}(p.r)\rangle \end{aligned}$$

For non-singleton sequences s and $s = \emptyset$ the function $\mathit{tree}(s, B)$ is undefined. Again this recursion doesn't terminate when a cycle is reachable from n in B. Note that this abstraction function "unshares" pointer representations that implement common subtrees by shared pointers.

The proof of the reasonableness lemma does not use that E, C are *functions*. So the same holds when one uses 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.

5.4 Doubly Linked Lists

As usual, the binary tree records can also be used for implementing doubly linked lists. A suitable anamorphic and hence reasonable abstraction function is

$$\begin{aligned} \mathit{dlist}(p) \;=\; \mathsf{if}\ \mathit{ptr}(p) \;=\; \diamond\ \mathsf{then}\, \varepsilon \\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \langle p.v, \mathit{dlist}(p.r)\rangle \end{aligned}$$

The fact that doubly linked lists form a cyclic data structure does not disturb us here, since we only use the set of links to the right, which by itself is acyclic. A similar phenomenon will occur with threaded trees.

In fact, *dlist* is isomorphic to *list* modulo the renaming of *head* into v and *tail* into r.

5.5 Infix-Threaded Binary Trees

Threads are used to speed up, e.g., the infix order traversal of trees. To this end, one replaces the $\diamond$ pointer in rightmost nodes by a pointer to the successor node in infix order. To distinguish proper links to right subtrees from thread links we need to extend our record scheme for binary threads by a selector *isth* of type $\mathcal{A} \to \mathbb{B}$, where $\mathbb{B}$ is the set of booleans.

Then a suitable anamorphic and hence reasonable abstraction function is

$$\begin{aligned} \mathit{thtree}(p) \;=\; \mathsf{if}\ \mathit{ptr}(p) = \diamond \\ \qquad\quad \mathsf{then}\, \varepsilon \\ \qquad\quad \mathsf{else}\ \mathsf{if}\ p.\mathit{isth} \\ \qquad\qquad\quad \mathsf{then}\, \langle \mathit{thtree}(p.l), p.v, \varepsilon\rangle \\ \qquad\qquad\quad \mathsf{else}\ \langle \mathit{thtree}(p.l), p.v, \mathit{thtree}(p.r)\rangle \end{aligned}$$

5.6 Less Abstract Abstraction Functions

So far we have used the same set of node values at the abstract and concrete levels. However, often we want to talk about some aspects of the representation, like the order in which certain cells are linked together. This can be achieved by considering e.g. lists or trees in which the node values are *addresses* rather than primitive values.

For instance, the list of addresses that belong to a list structure is recorded by

$$\begin{aligned} \mathit{cells}(p) \;=\; \mathsf{if}\ \mathit{ptr}(p) \;=\; \diamond\ \mathsf{then}\, \varepsilon \\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \langle \mathit{ptr}(p), \mathit{cells}(p.\mathit{tail}) \rangle \end{aligned}$$

Then a specification of "in-situ-ness" of concatenation can use the conjunct

$$\mathit{cells}(\mathit{pconc}(x, y, S)) \;=\; \mathit{cells}(x, S) \bullet \mathit{cells}(y, S)\ .$$

This approach was used in Möller (1991).

Similarly, well-threadedness can be expressed by stating that the infix order of the tree addresses is the same as the threaded traversal order.

This has further applications. It is well-known that AVL-like tree rotations preserve the infix traversal. From that and the above it follows that they also preserve the correctness of threading.

6 Conclusion and Outlook

We have only given examples of reasonable abstraction functions, but not of their use in actual calculations of pointer implementations of operations specified at the abstract level. Such examples can, however, be found in Möller (1997a,1997b). In particular, these papers present a number of algebraic laws for a selective updating operation that depend on the use of a reasonable abstraction function. The approach seems adequate, as the fairly concise derivations in those papers 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.

It remains to integrate the approach with the general theory of unfold operations or anamorphisms and their duals, the catamorphisms (see Meijer et al. (1991), Bird (1996) and Bird, de Moor (1996)). Many operations at the abstract level can be specified as catamorphisms, i.e., standard traversals of recursive data structures. So presumably one can develop a general fusion property for catamorphisms and anamorphic abstraction functions. This conjecture is also supported by the similarity of the derivations of in-situ concatenation and reversal for acyclic and cyclic lists in Möller (1997b). A proper treatment using the categorical theory of data types is beyond the scope of the present paper, though.

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. R.S. Bird, O. de Moor: Algebra of programming. Prentice-Hall 1996
4. C.A.R. Hoare: Proofs of correctness of data representations. Acta Informatica **1**, 271–281 (1972)
5. 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
6. B. Möller: Formal derivation of pointer algorithms. In: M. Broy (Hrsg.): Informatik und Mathematik. Berlin: Springer1991, 419–440
7. 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
8. B. Möller: Towards pointer algebra. Science of Computer Programming **21**, 57–90 (1993)
9. 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
10. B. Möller: Linked lists calculated. Institut für Informatik, Universität Augsburg, Report Nr. 1997-7, December 1997
11. G. Schmidt, T. Ströhlein: Relations and graphs. Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Berlin: Springer 1993

Appendix: Proofs for Section 3.2

Proof of Lemma 31:
We assume $\mathit{from}(p) = \mathit{from}(q)$ and use fixpoint induction on the recursive definition of $[p]^*$ with the continuous predicate

$$PP(X) \stackrel{\text{def}}{\Leftrightarrow} \forall\, V \subseteq [p]^*(U) : X(V) \subseteq [q]^*(V)\ .$$

The induction basis $PP(\emptyset)$ is trivial. The functional associated with the recursion for $[p]^*$ is $\tau : (\mathcal{A} \leftrightarrow \mathcal{A}) \to (\mathcal{A} \leftrightarrow \mathcal{A})$ given by

$$\tau(Y) \stackrel{\text{def}}{=} I \cup [p]\,;Y\ .$$

Now for the induction step $PP(X) \Rightarrow PP(\tau(X))$ we calculate

$$\tau(X)(V)$$

$$
\begin{array}{ll}
= & \quad \{\![\text{ definition of } \tau \ ]\!\} \\
& (I \cup [p]\,;X)(V) \\
= & \quad \{\![\text{ distributivity }]\!\} \\
& V \cup ([p]\,;X)(V) \\
= & \quad \{\![\text{ image set }]\!\} \\
& V \cup X([p](V)) \\
\subseteq & \quad \{\![\text{ induction hypothesis } PP(X), \text{ since } [p](V) \subseteq [p]^*(U) \ ]\!\} \\
& V \cup [q]^*([p](V)) \\
= & \quad \{\![\text{ since } \mathit{from}(p) = \mathit{from}(q) \text{ and } [p](V) \subseteq \mathit{reach}(p) \ ]\!\} \\
& V \cup [q]^*([q](V)) \\
= & \quad \{\![\text{ distributivity }]\!\} \\
& (I \cup [q]\,;[q]^*)(V) \\
= & \quad \{\![\text{ fixpoint property of } [q]^* \ ]\!\} \\
& [q]^*(V)\ .
\end{array}
$$

This shows in particular $[p]^*(U) \subseteq [q]^*(U)$. Switching p and q gives the reverse inclusion.

Proof of Corollary 32:
Assume again $\mathit{from}(p) = \mathit{from}(q)$. Then set $n \stackrel{\text{def}}{=} \mathit{ptr}(p) = \mathit{ptr}(q)$ and $S \stackrel{\text{def}}{=} \mathit{sto}(p), T \stackrel{\text{def}}{=} \mathit{sto}(q)$. We have $S_{s_i}(n) = T_{s_i}(n)$ $(*)$ and calculate

$$
\begin{array}{ll}
& \mathit{from}(p.s_i) \\
= & \quad \{\![\text{ definition of selection }]\!\} \\
& \mathit{from}(S_{s_i}(n), S) \\
= & \quad \{\![\text{ definition of } \mathit{from} \ ]\!\} \\
& (S_{s_i}(n), \mathit{reach}(S_{s_i}(n), S) \bowtie S) \\
= & \quad \{\![\text{ by } (*) \text{ and Lemma 31, since } \mathit{reach}(S_{s_i}(n), S) \subseteq \mathit{reach}(p) \\
& \qquad \text{and } \mathit{reach}(T_{s_i}(n), T) \subseteq \mathit{reach}(q) \ ]\!\} \\
& (T_{s_i}(n), \mathit{reach}(T_{s_i}(n), T) \bowtie T) \\
= & \quad \{\![\text{ definition of } \mathit{from} \ ]\!\} \\
& \mathit{from}(T_{s_i}(n), T) \\
= & \quad \{\![\text{ definition of selection }]\!\} \\
& \mathit{from}(q.s_i)
\end{array}
$$

Proof of Corollary 33:
A straightforward induction on the minimal i such that $n \in [sto(p)]^i(ptr(p))$, using Corollary 32.