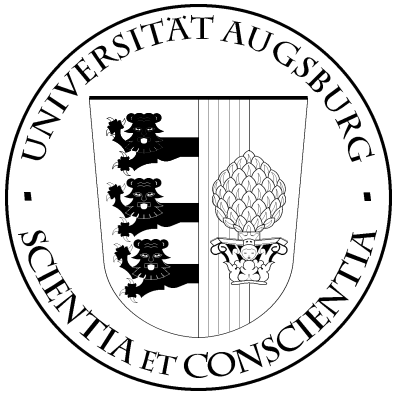
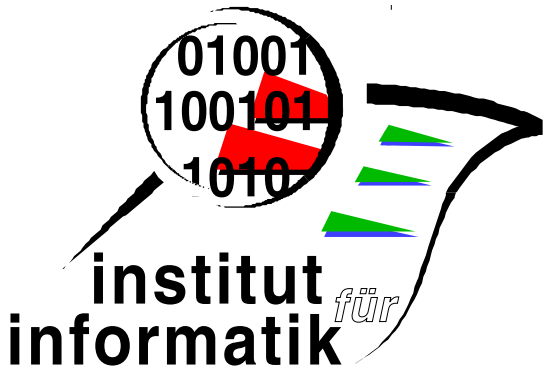
Universität Augsburg



Case studies for the derivation of pointer algorithms

Thorsten Ehm

Report 2003-9 June 2003



Institut für Informatik
D-86135 Augsburg

Case studies for the derivation of pointer algorithms

Thorsten Ehm

Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
Ehm@informatik.uni-augsburg.de

Abstract The method presented in [13] by Bernhard Möller to derive pointer algorithms has been shown well-applicable and easy-to-use in several various examples.
We present the derivation of different pointer algorithms on lists from their functional specification. The intention of this paper is to show the advantages of the method on a number of medium-sized examples on the one hand. On the other hand we point out also problems and tasks to be solved to achieve a complete framework for the derivation of pointer algorithms working on inductively defined data structures.

1 Introduction

Basically there are two different ways to come to a correct algorithm. Either you give a possible candidate together with a specification and have to prove that the algorithm fulfills all requirements of the specification. Or one starts from the specification and tries to transform it into an executable algorithm using correctness-preserving transformations only. The first method sometimes is more intuitive, because this is the way programming works since its beginning. Nevertheless there are several disadvantages. Evidently, the work to be done is doubled. First one has to provide a specification. This normally evolves from several stages of a round trip process between the customer and the software engineer. The developer then has to provide an implementation that often contains errors or does not fit the specification in all points. At the end of this process in a formal software development process the programmer has to prove that his implementation meets the specification. The differences between these two methods are comparable to the ones of parser generators versus hand-made parser code. If the grammar changes only little we have an automatic procedure to create a new program that parses the language described. This is also the intention of transformational program development. If the specification is changed, one hopes that big parts of the derivation process can be automatically or semi-automatically replayed. The other advantage is that all proofs of properties of the system can be done on the specification side. If only correctness-preserving transformation rules are used one can be sure that the properties also hold for the implementation.

In this paper we are focusing on transformational program development and the correctness of pointer algorithms. Note, that we are only interested in algorithms that really alter the pointer structure. The other ones, like e.g. `elem` to determine if a list contains a given element, can be treated the same way (see Sec. 3.2). But these algorithms are much easier. We will show how this works on the previously mentioned function `elem`. Normally these types of functions are tail-recursive and do not demand for any strange side-conditions holding on the pointer structure like for example reachability constraints.

This paper also should serve as a programme for what are the problems and what has to be done to achieve a complete framework for the derivation of pointer algorithms. Our goal is on the one hand side to provide a concise algebraic toolbox to describe and calculate with pointer structures but on the other hand not to lose sight of practical applicability.

The paper is structured as follows: Section 2 presents some changes and improvements of the original pointer algebra defined by Möller. In Section 3 several standard algorithms on singly-linked lists are derived from functional specifications. Section 4 investigates which sort of abstract patterns evolved from the derivations and how they can be transformed into an imperative form. Problems and future tasks to deal with are pointed out in Section 5. Here also some ideas for future research are noted. Section 6 closes with a short summary.

2 Detailed observations of pointer algebra

This section makes some critical remarks, minor improvements and shows more detailed derivations and proofs of the pointer algebra presented in [13] by Möller. The goal here is not to question the whole framework presented. On the contrary, it has been proved to be widely applicable and not too complex to be only of theoretical interest. So we want to improve the whole building to come to a well based calculus.

Subsections 2.1 and 2.2 give a short overview of the method. The reader that is well up in this calculus might skip these sections. For further details the reader is referred to the cited paper.

2.1 Pointer structures

In our model a pointer structure $\mathcal{P} = (s, P)$ consists of a store P and a list of entries s. The entries of a pointer structure are addresses $\mathcal{A}$ that form starting points of the modeled data structures. We assume a distinguished element $\diamond \in \mathcal{A}$ representing a terminal node (e.g. null in C or nil in Pascal). A store is a family of relations (more precisely partial maps) either between addresses or between addresses and node values $\mathcal{N}_j$ such as *Integer* or *Boolean*. Each relation represents a selector on the records like e.g. *head* and *tail* for lists with functionality $\mathcal{A} \to \mathcal{N}_j$ respectively $\mathcal{A} \to \mathcal{A}$.

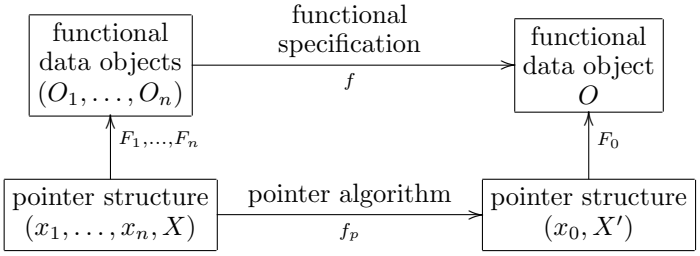
Each abstract object implemented by pointer structures is represented by a pointer structure (n, P) with a single entry $n \in \mathcal{A}$ which represents the entry point of the data structure such as for example the root node in a tree. For convenience we introduce the access functions

$$ptr(n, L) = n \qquad\qquad sto(n, L) = L$$

The relation between abstract and concrete levels is established by a partial abstraction function as described in [11]. An example of an abstraction function for singly-linked lists is:

$$\begin{aligned} list(p) = \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ [] \\ \mathsf{else}\ p.head : list(p.tail) \end{aligned}$$

The exact definition of operators used here are given in Section 2.2. Given an abstraction function F, the pointer implementation f_p of a given functional operation f is now specified by the equation $f(F(p)) = F(f_p(p))$.



To derive a pointer implementation f_p from this specification one tries to transform the expression $f(F(p))$ by equational reasoning into an expression $F(E)$ such that E does not contain the abstraction function. Then we can define f_p by setting $f_p(p) = E$. We will see several examples of this methodology more exactly in Section 3. Certainly we will not use nondeterministic or intuitively senseless functions as abstractions. So in the following we only will use *reasonable* abstraction functions.

Definition 1. *An abstraction function is called reasonable if equality of the reachable parts of two pointer structures implies equal abstractions.*

An improved and more detailed definition of *reasonable* can be found in 2.3.

2.2 Operations

We want to give only the necessary definitions of operations used in this paper. More of them and proofs can be found in [13]. The following operations on relations all are canonically lifted to families of relations. Algorithms on pointer structures stand out for altering links between elements. Such modification has to be modeled in the calculus as well. We use an update operator | (pronounced "onto") that overwrites relation S by relation R:

Definition 2. $R \mid S \stackrel{\text{def}}{\Leftrightarrow} R \cup \overline{dom(R)} \bowtie S$

Here we have used the *domain restriction* operator $\bowtie$ which is defined as $L \bowtie S = S \cap (L \times N)$ to select a particular part of $S \subseteq \mathcal{P}(M \times N)$. The update operator takes all links defined in R and adds the ones from S that no link starts from in R. To be able to change exactly one pointer in one explicit selector we define a sort of "mini-store" that is a family of partial maps defined by:

Definition 3. $(x \stackrel{k}{\rightarrow} y) \stackrel{\text{def}}{\Leftrightarrow} \left\{ \begin{array}{ll} \{(x,y)\} & \textit{for selector } k \\ \emptyset & \textit{otherwise} \end{array} \right\}$

It is clear that overwriting a pointer structure with links already defined in it does not change the structure. This leads to

Lemma 1. $S \subseteq T \Rightarrow S \mid T = T$ *(Annihilation)*

To have a more intuitive notation leaned on traditional programming languages, we introduce the following selective update notation:

Definition 4. *For selector k of type $\mathcal{A} \rightarrow \mathcal{A}$*
$(n, P).k := (m, Q) \stackrel{\text{def}}{\Leftrightarrow} (n, (n \stackrel{k}{\rightarrow} m) \mid Q)$

which overwrites Q with a single link from n to m at selector k. Selection is performed the same way:

Definition 5. k *of type* $\mathcal{A} \rightarrow \mathcal{A}$: $(n, P).k \stackrel{\text{def}}{\Leftrightarrow} (P_k(n), P)$

k *of type* $\mathcal{A} \rightarrow \mathcal{N}_j$: $(n, P).k \stackrel{\text{def}}{\Leftrightarrow} P_k(n)$

To have the possibility to insert new (unused) addresses into the data structure we define the newrec operator. The operator $\mathsf{newrec}((n, L), k_i \mapsto x_i)$ alters the pointer structure (n, L) to have a new record previously not in (n, L) and each selector k_i pointing to x_i. So for example $\mathsf{newrec}((n, L), head \mapsto 3, tail \mapsto \diamond)$ returns a pointer structure (m, K) with m a new address previously not used in (n, L) and store K consisting of L united with two new links $(m \overset{head}{\rightarrow} 3)$ and $(m \overset{tail}{\rightarrow} \diamond)$.

2.3 Refinement of reasonability

As described above, the connection between concrete and abstract level of reasoning is achieved using abstraction functions. It is necessary to demand reasonable abstraction functions that avoid magically constructed abstract objects. Demanding reasonable abstraction functions prevents for example the use of random generators or similar in the abstraction process. For some tasks to be solved in this context see Section 5.1.

The notion of reasonable abstraction functions is not wrongly defined but in our eyes not general enough. Although there was no need for such an improved version of reasonability in all the example derivations, there maybe applications in the future where one needs this form. The refined version presented here also is much more intuitive.

In the original paper an abstraction function F is called reasonable if equality of the parts of a store reachable from two pointer structures implies that F provides the same abstract objects from both pointer structures. Formally:

$$from(p) = from(q) \Rightarrow p \sim_F q$$

The stronger variant now does not demand that the whole store reachable from p or q is equal, but only the part that is reachable via selectors used in the definition of F. So we not only need a stronger version of $from$ but also variants of the operations $from$ is based on:

Definition 6.
$$from_F(m, L) = (m, reach_F(m, L) \bowtie L)$$
$$reach_F(m, L) = [L]^*_F(m)$$
$$[L]_F = \bigcup_{k \in K_F} L_k$$

Here K_F are all the selectors k used by the abstract object created by F. Note that this also includes selectors that are not directly mentioned in F. For example an abstraction function for doubly-linked lists only uses the *next* selectors. But the data structure itself also consists of a *previous* selector. So exactly these two selectors are in K_F for doubly-linked lists. As we have parameterized all needed operations by the abstraction function, it is evident which selectors are used. The new definition now is:

Definition 7 (reasonable (new)). *F is* reasonable *if*

$$from_F(p) = from_F(q) \Rightarrow p \sim_F q$$

The thus improved calculus can now be used to achieve a more local reasoning about pointer structures by only observing the really used selectors of a record.

2.4 Simplification of newrec

Another part of the calculus that can be improved is newrec This operation is used to allocate new (unused) records in a pointer structure. The newrec operation is

defined as relation between a pair consisting of the original pointer structure and a tuple of new values for all the selectors used and, on the other hand, the new pointer structure. But the entry of the original pointer structure is never used in the relational definition of newrec. So we can simplify the signature to

$$\mathsf{newrec} \in \mathcal{S} \times \mathcal{V} \Leftrightarrow \mathcal{P}$$

and adjust the definition respectively. This new version already was successfully used in [8] to derive imperative insertion algorithms into lists and trees.

2.5 An important rule

As we will see in Section 3, a rule used in almost all derivations is the proposition

$$F((p.k := q').k) = F(q') \tag{1}$$

for reasonable F and some preconditions. But reasonableness is not the only precondition that has to hold. As we want to derive correct pointer algorithms, we have to know under which conditions simplifications are valid and transformations can be used. Otherwise the transformations would not be correct. Therefore we have to investigate *exactly* which preliminaries have to hold to be able to apply the rule. We first mention some auxiliary lemmas that we will need (The reader interested in the proofs is referred to [13]).

Lemma 2. *Let F be reasonable*

1. $q \nvdash ptr(p) \Rightarrow (p.k := q).k \sim_F q$
2. $ptr(p) \in noreach(q) \Rightarrow q \nvdash ptr(p)$

Additionally we repeat the definition of the predicate *norea* that also will play a rôle in rule (1) whereas $noreach(p) \overset{\text{def}}{\Leftrightarrow} recs(p) \setminus reach(p)$.

Definition 8. $f_p \in norea \Leftrightarrow \forall (p,q) \in f_p.\ noreach(p) \subseteq noreach(q)$

Note, that p in the index of f has nothing to do with the variable p. It only shows, that f_p is a *pointer* implementation of algorithm f. In contrast to the normal order of a proof we start here with the consequence of the rule and derive and sum up which preliminaries we needed. Afterwards we state the complete lemma. We will denote a possible result of application of f_p on q by q', so $q' \in f_p(q)$ which is equivalent to $(q,q') \in f_p$. We assume $L = sto(p) = sto(q)$ and calculate:

Proof.

$F((p.k := q').k) = F(q')$

$\Leftrightarrow$ {[definition of $\sim_F$]}

$(p.k := q').k \sim_F q'$

$\Leftarrow$ {[F reasonable and Lemma 2.1]}

$q' \nvdash ptr(p)$

$\Leftarrow$ {[Lemma 2.2]}

$ptr(p) \in noreach(q')$

$\Leftarrow$ {[$f_p \in norea$]}

$ptr(p) \in noreach(q)$

Since p and q have the same store, $recs(p) = recs(q)$ holds and therefore $ptr(p) \in recs(q)$. So we only have to investigate under which conditions $ptr(p)$ is not in $reach(q)$. We distinguish two cases which require different preconditions:

Case 1: $ptr(q)$ is reachable from p (i.e. $ptr(q) \in reach(p)$). Then we can show by idempotence of $reach$ (see [9]):

$$\begin{aligned} & ptr(p) \in reach(q) \\ \Rightarrow\ & ptr(p) \in reach(p) \\ \Rightarrow\ & p \text{ cyclic} \end{aligned}$$

So if we demand acyclicity of p in this case $ptr(p) \notin reach(q)$ will hold.

Case 2: $ptr(q)$ is not reachable from p (i.e. $ptr(q) \notin reach(p)$).

$$\begin{aligned} & ptr(p) \notin reach(q) \\ \overset{(*)}{\Leftarrow}\ & ptr(p) \cap reach(q) \subseteq \{\diamond\} \\ \Leftarrow\ & reach(p) \cap reach(q) \subseteq \{\diamond\} \\ \Leftrightarrow\ & \neg sharing(ptr(p), ptr(q), L) \end{aligned}$$

This shows that in this case the precondition $\neg sharing(ptr(p), ptr(q), L)$ will establish the needed properties. Certainly the step marked by $(*)$ only holds if $ptr(p) \neq \diamond$ which should be the case. Otherwise the assignment $p.k := q'$ would be undefined, as $\diamond$ can not be dereferenced.

Now we can give the complete rule:

Lemma 3. *Let $L = sto(p) = sto(q)$, $q' \in f_p(q)$ and assume the following preconditions hold:*

1. *F is reasonable*
2. *$f_p \in norea$*
3. *$ptr(p) \neq \diamond$*
4. *$acyclic(p)$*
5. *$\neg sharing(ptr(p), ptr(q), L)$*

Then $F((p.k := q').k) = F(q')$ holds.

3 Derivations

In this section we show several derivations of algorithms on lists. They serve as the basis for further research and demonstrate which sorts of pattern arise in this context to be used for classification of algorithms. The restriction to list processing functions is not as severe as it sounds on the first sight. As we have shown in [8], different pointer structures like for example trees only use some more selectors that can be handled by case distinction.

3.1 Preliminaries

As our goal is the derivation of algorithms on lists, we need an abstraction function that establishes the connection between the pointer structure level and the object level. We will use the naturally defined function *list* from Section 2.1. Additionally we also need two little lemmas to be used in the derivations which will be given here without proofs. The interested reader may have a look into [13].

Lemma 4. *Let $p = (m, L), q = (n, L'), r = (m, L')$ and $j : \mathcal{A} \to \mathcal{N}_i, k : \mathcal{A} \to \mathcal{A}$*

1. *$(p.k := q).j = r.j$*

2. $q \nvdash recs(S) \Rightarrow S \mid q \sim_F q$

We will use the two C-like operators && and || that represent sequential conjunction and disjunction, respectively:

Definition 9. *The sequential logical operators && and || are defined by:*

$$B \,\&\&\, C = \mathsf{if}\ B\ \mathsf{then}\ C\ \mathsf{else}\ \mathsf{false}$$
$$B \,||\, C = \mathsf{if}\ B\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ C$$

3.2 Observational functions

At first we show that functions that do not alter the pointer structure are relatively easy to derive. This is because there is no need to reason about complex properties of the pointer structure. Such observational functions just return certain information about the data structure. So there also no abstraction function shows up on the pointer algorithm side of the specification. The example we use is `elem` that determines if an element is a member of a list or not. We will use Haskell [3] like notation to denote functional algorithms. A functional specification is given by:

```
elem a []       = false
elem a (x:xs) = a==x || elem a xs
```

Applying the method of [13] we use the equation

$$elem_p\ a\ p = elem\ a\ list(p)$$

to specify a pointer implementation $elem_p$ of `elem`. Here and in the sequel we will use p as an abbreviation for the pointer structure (m, L). As `elem` is defined by case distinction, a pointer algorithm can be derived from the specification by also reasoning about the following two cases.

Case $m = \diamond$**:**

$elem\ a\ list(p)$

$=$ $\quad \{\![$ unfold definition of $list$ and $elem$ $]\!\}$

$false$

So the only possible choice is $elem_p\ a\ (\diamond, L) = false$. The second case works almost the same way:

Case $m \neq \diamond$**:**

$elem\ a\ list(p)$

$=$ $\quad \{\![$ unfold definitions of $list$ and $elem$ $]\!\}$

$a == p.head \,||\, elem\ a\ list(p.tail)$

$=$ $\quad \{\![$ fold with spec. of $elem_p$ $]\!\}$

$a == p.head \,||\, elem_p\ a\ p.tail$

By putting both cases together we achieve the following pointer algorithm:

$$elem_p\ a\ (m, L) = \mathsf{if}\ m == \diamond\ \mathsf{then}\ false \\ \mathsf{else}\ a == L_{head}(m) \,||\, elem_p\ a\ p.tail$$

As this function is completely tail-recursive, we simply can apply a standard transformation scheme to get an imperative form of $elem_p$.

$$
\begin{array}{l}
elem_p\ a\ (m, L) = \mathsf{if}\ m == \diamond\ \mathsf{then}\ false \\
\qquad\qquad\qquad\qquad \mathsf{else\ if}\ a == L_{head}(m)\ \ \mathsf{then}\ true \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ elem_p\ a\ p.tail \\
\hline\hline
elem_p\ a\ (m, L) = \mathsf{var}\ vm = m \\
\qquad\qquad\qquad\quad \mathsf{while}\ vm \neq \diamond\ \&\&\ a \neq vm.head\ \mathsf{do}\ vm := vm.tail \\
\qquad\qquad\qquad\quad \mathsf{if}\ vm == \diamond\ \mathsf{then}\ false \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ true
\end{array}
$$

[See [16], Chapter 7.1 (p.329) [2]

Additionally we treated the store implicitly and wrote $vm.tail$ to achieve a more C-like syntax.

3.3 Standard derivations

We have seen in the previous example that the method resembles in the derivation the case by case definition of the functional algorithm. As we want to derive list-processing functions from a functional specification, all of them are recursively defined and therefore need a termination case. This gives us the possibility to show once and for all the transformation for functions working on a list argument using the empty list as termination case. So assume that `f c [] = []` is one part of the definition of function `f`. Here `c` could stand for several curried arguments or none and plays no essential rôle in the derivation process of the termination case. Under the condition $ptr(p) = \diamond$ we are able to calculate:

$$
\begin{array}{ll}
 & f\ c\ list(p) \\
= & \quad \{\![\text{ unfold definition of } list \ ]\!\} \\
 & f\ c\ [] \\
= & \quad \{\![\text{ unfold definition of } f \ ]\!\} \\
 & [] \\
= & \quad \{\![\text{ fold definition of } list \ ]\!\} \\
 & list(p)
\end{array}
$$

So we can choose $f_p\ c\ p = p$ to be the implementation for the case $ptr(p) = \diamond$.

Another standard termination case is to return a second list argument from the parameter list. So assume there is a line `f c [] ys = ys` in the definition of f. Then the derivation looks like:

$$
\begin{array}{ll}
 & f\ c\ list(p)\ list(q) \\
= & \quad \{\![\text{ unfold definition of } list \ ]\!\} \\
 & f\ c\ []\ list(q) \\
= & \quad \{\![\text{ unfold definition of } f \ ]\!\} \\
 & list(q)
\end{array}
$$

So choose $f_p\ c\ (\diamond, n, L) = (n, L)$. In both cases the order of the arguments does not matter. This means that we are also able to handle definition like

```
f c xs []   = xs
f c [] ys d = ys
```

with `c` and `d` representing arbitrary arguments.

[2] The rule in [16] was given wrongly. A corrected derivation can be found in Appendix A.

3.4 Insert into sorted lists

As a first algorithm really changing the pointer structure we want to derive a function, that inserts an element a into a sorted list:

```
insert a []       = [a]
insert a (x:xs) = if a ≤ x then a:(x:xs)
                           else x: insert a xs
```

Possible pointer implementations are given by the specification

$$list(insert_p\ a\ p) = insert\ a\ list(p)$$

We now can derive a pointer algorithm from the specification by reasoning over two cases.

Case $m = \diamond$**:**

$insert\ a\ list(p)$

$=$ $\{\![$ unfold definitions of $list$ and $insert$ $]\!\}$

$[a]$

$=$ $\{\![$ choose $r \in \mathsf{newrec}(p, (head \mapsto a, tail \mapsto \diamond))$ $]\!\}$

$[r.head]$

$=$ $\{\![$ $list(r.tail) = []$ and $[a] = a : []$ $]\!\}$

$r.head : list(r.tail)$

$=$ $\{\![$ fold definition of $list$ $]\!\}$

$list(r)$

So choose $insert_p\ a\ (\diamond, L) = \mathsf{newrec}((\diamond, L), (head \mapsto a, tail \mapsto \diamond))$.

Case $m \neq \diamond$**:**

$insert\ a\ list(p)$

$=$ $\{\![$ unfold definitions of $list$ and $insert$ $]\!\}$

$\mathsf{if}\ a \leq p.head\ \mathsf{then}\ a : (p.head : list(p.tail))$
$\mathsf{else}\ p.head : insert\ a\ list(p.tail)$

$=$ $\{\![$ fold with spec. of $insert_p$; Choose r $\in insert_p\ a\ (L_{tail}(m), L)$ $]\!\}$

$\mathsf{if}\ a \leq p.head\ \mathsf{then}\ a : (p.head : list(p.tail))$
$\mathsf{else}\ p.head : list(r)$

$=$ $\{\![$ set $s = q.tail := r$ and Lemma 4.1 $]\!\}$

$\mathsf{if}\ a \leq p.head\ \mathsf{then}\ a : (p.head : list(p.tail))$
$\mathsf{else}\ s.head : list(r)$

$=$ $\{\![$ Lemma 3 $]\!\}$

$\mathsf{if}\ a \leq p.head\ \mathsf{then}\ a : (p.head : list(p.tail))$
$\mathsf{else}\ s.head : list(s.tail)$

$=$ $\{\![$ fold with definition of list (2 times) $]\!\}$

$\mathsf{if}\ a \leq p.head\ \mathsf{then}\ a : list(p)$
$\mathsf{else}\ list(s)$

$=$ $\{\![$ choose $t \in \mathsf{newrec}(L, (head \mapsto a, tail \mapsto p))$ $]\!\}$

$$
\begin{array}{ll}
 & \mathsf{if}\ a \le p.head\ \ \mathsf{then}\ t.head : list(t.tail) \\
 & \qquad\qquad\qquad \mathsf{else}\ list(s) \\
= & \quad \{\!\![\ \text{fold with definition of list}\ ]\!\!\} \\
 & \mathsf{if}\ a \le p.head\ \mathsf{then}\ list(t)\ \mathsf{else}\ list(s) \\
= & \quad \{\!\![\ \text{if propagation}\ ]\!\!\} \\
 & list(\mathsf{if}\ a \le p.head\ \mathsf{then}\ t\ \mathsf{else}\ s)
\end{array}
$$

In summary we have derived the pointer algorithm:

$$
\begin{array}{l}
insert_p\ a\ (m, L) = \mathsf{if}\ \ m == \diamond \\
\qquad\qquad\qquad\qquad\quad \mathsf{then\ newrec}((m, L), (head \mapsto a, tail \mapsto (\diamond, L))) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else\ if}\ \ a \le L_{head}(m) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then\ newrec}((m, L), (head \mapsto a, tail \mapsto (m, L))) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ p.tail := insert_p\ a\ (L_{tail}(m), L)
\end{array}
$$

which can be simplified by melting the two first branches to:

$$
\begin{array}{l}
insert_p\ a\ (m, L) = \mathsf{if}\ \ m == \diamond\ ||\ a \le L_{head}(m) \\
\qquad\qquad\qquad\qquad\quad \mathsf{then\ newrec}((m, L), (head \mapsto a, tail \mapsto (m, L))) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ p.tail := insert_p\ a\ (L_{tail}(m), L)
\end{array}
$$

3.5 Delete first occurrence of an element in a list

In this section we derive a function, that deletes only the first occurrence of an element in a list:

```
del a []       = []
del a (x:xs) = if a==x then xs
                       else x: del a xs
```

Possible pointer implementations are given by the following specification:

$$list(del_p\ a\ p) = del\ a\ list(p)$$

where $p = (m, L)$ as before.

We again can derive a pointer algorithm from the specification by reasoning over two cases. The first case is covered by the sample in Section 3.3. For the second case we calculate:

Case $m \neq \diamond$**:**

$$
\begin{array}{ll}
 & del\ a\ list(p) \\
= & \quad \{\!\![\ \text{unfold definitions of } list \text{ and } del\ ]\!\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\ list(p.tail) \\
 & \qquad\qquad\qquad \mathsf{else}\ p.head : del\ a\ list(p.tail) \\
= & \quad \{\!\![\ \text{fold with spec. of } del_p\text{; Choose r} \in del_p\ a\ (L_{tail}(m), L)\ ]\!\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\ list(p.tail) \\
 & \qquad\qquad\qquad \mathsf{else}\ p.head : list(r) \\
= & \quad \{\!\![\ \text{set } s = p.tail := r \text{ and Lemma 4.1}\ ]\!\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\ list(p.tail) \\
 & \qquad\qquad\qquad \mathsf{else}\ s.head : list(r) \\
= & \quad \{\!\![\ \text{Lemma 3}\ ]\!\!\}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\, list(p.tail) \\
 & \qquad\qquad\qquad \mathsf{else}\, s.head : list(s.tail) \\
= & \quad \{\![\ \text{fold with definition of } list\ ]\!\} \\
 & \mathsf{if}\ a == p.head\, \mathsf{then}\, list(p.tail)\, \mathsf{else}\, list(s) \\
= & \quad \{\![\ \text{if propagation}\ ]\!\} \\
 & list(\mathsf{if}\ a == p.head\, \mathsf{then}\, p.tail\, \mathsf{else}\, s)
\end{array}
$$

So we have derived the pointer algorithm:

$$
\begin{array}{l}
del_p\ a\ (m, L) = \mathsf{if}\quad m == \diamond\ \ \mathsf{then}\ (\diamond, L) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else\ if}\ \ a == L_{head}(m) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\, p.tail \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\, p.tail := del_p\ a\ (L_{tail}(m), L)
\end{array}
$$

3.6 Delete all occurrences of an element in a list

An extension of the previously derived deletion algorithm is a function, that that deletes all occurrences of an element in a list:

```
delete a []       = []
delete a (x:xs) = if a==x then delete a xs
                          else x: delete a xs
```

Possible pointer implementations are given by the following specification:

$$list(delete_p\ a\ p) = delete\ a\ list(p)$$

We derive a pointer algorithm from the specification by reasoning over two cases. Again we can refer to Section 3.3 for the first case. The second is:

Case $m \neq \diamond$**:**

$$
\begin{array}{ll}
 & delete\ a\ list(p) \\
= & \quad \{\![\ \text{unfold definitions of } list \text{ and } delete\ ]\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\, delete\ a\ list(p.tail) \\
 & \qquad\qquad\qquad \mathsf{else}\, p.head : delete\ a\ list(p.tail) \\
= & \quad \{\![\ \text{fold with spec. of } delete_p\text{; Choose r} \in delete_p\ a\ (L_{tail}(m), L)\ ]\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\, list(r) \\
 & \qquad\qquad\qquad \mathsf{else}\, p.head : list(r) \\
= & \quad \{\![\ \text{set } s = p.tail := r \text{ and Lemma 4.1}\ ]\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\, list(r) \\
 & \qquad\qquad\qquad \mathsf{else}\, s.head : list(r) \\
= & \quad \{\![\ \text{Lemma 3}\ ]\!\} \\
 & \mathsf{if}\ a == p.head\ \ \mathsf{then}\, list(r) \\
 & \qquad\qquad\qquad \mathsf{else}\, s.head : list(s.tail) \\
= & \quad \{\![\ \text{fold with definition of } list\ ]\!\} \\
 & \mathsf{if}\ a == p.head\, \mathsf{then}\, list(r)\, \mathsf{else}\, list(s) \\
= & \quad \{\![\ \text{if propagation}\ ]\!\} \\
 & list(\mathsf{if}\ a == p.head\, \mathsf{then}\, r\, \mathsf{else}\, s)
\end{array}
$$

So we have derived the pointer algorithm:

$$\begin{array}{l} delete_p\ a\ (m, L) = \mathsf{if}\quad m == \diamond\ \mathsf{then}\ (\diamond, L) \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else\ if}\ \ a == L_{head}(m) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ delete_p\ a\ (L_{tail}(m), L) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ p.tail := delete_p\ a\ (L_{tail}(m), L) \end{array}$$

3.7 Mix two lists

In this section we want to derive a function, that shuffles the elements of two lists element by element:

```
mix []     ys = ys
mix (x:xs) ys = x:mix ys xs
```

Possible pointer implementations are given by the specification:

$$list(mix_p(m, n, L)) = mix\ list(p)\ list(q)$$

where $p = (m, L)$ and $q = (n, L)$.

The first case is handled in the second part of Section 3.3. The second case works the same as in the derivations before.

Case $m \neq \diamond$**:**

$mix\ list(p)\ list(q)$

$=$ $\quad \{\!\![$ unfold definitions of $list$ and mix $]\!\!\}$

$p.head : mix\ list(q)\ list(p.tail)$

$=$ $\quad \{\!\![$ fold with spec. of mix_p; Choose r $\in mix_p(n, L_{tail}(m), L)$ $]\!\!\}$

$p.head : list(r)$

$=$ $\quad \{\!\![$ set $s = p.tail := r$ and Lemma 4.1 $]\!\!\}$

$s.head : list(r)$

$=$ $\quad \{\!\![$ Lemma 3 $]\!\!\}$

$s.head : list(s.tail)$

$=$ $\quad \{\!\![$ fold with definition of $list$ $]\!\!\}$

$list(s)$

So we have derived the pointer algorithm:

$$\begin{array}{l} mix_p(m, n, L) = \mathsf{if}\quad m == \diamond\ \mathsf{then}\ (n, L) \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ p.tail := mix_p(n, L_{tail}(m), L) \end{array}$$

3.8 Merge two sorted lists

In this section we want to derive a merge function, that merges two sorted lists into an also sorted one:

```
merge [] ys          = ys
merge xs []          = xs
merge (x:xs) (y:ys) = if x ≤ y then x:merge xs (y:ys)
                                else y:merge (x:xs)
```

Possible pointer implementations are given by the following specification:

$$list(merge_p(m, n, L)) = merge\ list(p)\ list(q)$$

where $p = (m, L)$ and $q = (n, L)$.

Here we have to distinguish three cases because `merge` has two ways of termination. But both are instances of the sample derivation presented in Section 3.3. The third case is not as easy as in the derivations before because all the work has to be done twice, once in each branch.

Case $m \neq \diamond$ and $n \neq \diamond$:

$merge\ list(p)\ list(q)$

$=$ $\{\![$ unfold definitions of $list$ (2 times) and $merge$ $]\!\}$

$\mathsf{if}\ p.head \leq q.head\ \mathsf{then}\ p.head : merge\ list(p.tail)\ list(q)$
$\mathsf{else}\ q.head : merge\ list(p)\ list(q.tail)$

$=$ $\left\{\!\!\left[\begin{array}{l} \text{fold with spec. of } merge_p\text{; Choose } r \in merge_p(L_{tail}(m), n, L) \\ \text{and } s \in merge_p(m, L_{tail}(n), L) \end{array} \right]\!\!\right\}$

$\mathsf{if}\ p.head \leq q.head\ \mathsf{then}\ p.head : list(r)$
$\mathsf{else}\ q.head : list(s)$

$=$ $\{\![$ set $t = p.tail := r$ and $u = q.tail := s$ and Lemma 4.1 $]\!\}$

$\mathsf{if}\ p.head \leq q.head\ \mathsf{then}\ t.head : list(r)$
$\mathsf{else}\ u.head : list(s)$

$=$ $\{\![$ Lemma 3 $]\!\}$

$\mathsf{if}\ p.head \leq q.head\ \mathsf{then}\ t.head : list(t.tail)$
$\mathsf{else}\ u.head : list(u.tail)$

$=$ $\{\![$ fold with definition of $list$ (2 times) $]\!\}$

$\mathsf{if}\ p.head \leq q.head\ \mathsf{then}\ list(t)\ \mathsf{else}\ list(u)$

$=$ $\{\![$ if propagation $]\!\}$

$list(\mathsf{if}\ p.head \leq q.head\ \mathsf{then}\ t\ \mathsf{else}\ u)$

So we have derived the pointer algorithm:

$$\begin{array}{l} merge_p(m, n, L) = \\ \quad \mathsf{if}\ \ m == \diamond \\ \qquad \mathsf{then}\ (n, L) \\ \qquad \mathsf{else\ if}\ \ n == \diamond\ \ \mathsf{then}\ (m, L) \\ \qquad\qquad\qquad \mathsf{else\ if}\ \ L_{head}(m) \leq L_{head}(n) \\ \qquad\qquad\qquad\qquad \mathsf{then}\ p.tail := merge_p(L_{tail}(m), n, L) \\ \qquad\qquad\qquad\qquad \mathsf{else}\ q.tail := merge_p(m, L_{tail}(n), L) \end{array}$$

3.9 Filter on lists

Now we want to derive a destructive filter function:

```
filter c []       = []
filter c (x:xs) = if c x then x:filter c xs
                         else filter c xs
```

Here c is a predicate that rules which elements remain in the list. The pointer algorithms $filter_p$ are specified by the following equation:

$$list(filter_p\ c\ p) = filter\ c\ list(p)$$

The first case of the derivation follows again from the observation in Section 3.3. The second case is:

Case $m \neq \diamond$**:**

$filter\ c\ list(p)$

$=$ $\{\![$ unfold definitions of $list$ and $filter$ $]\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ p.head : filter\ c\ list(p.tail)$
$\mathsf{else}\ filter\ c\ list(p.tail)$

$=$ $\{\![$ fold with spec. of $filter_p$; Choose r $\in filter_p\ c\ (L_{tail}(m), L)$ $]\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ p.head : list(r)$
$\mathsf{else}\ list(r)$

$=$ $\{\![$ set $s = p.tail := r$ and Lemma 4.1 $]\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ s.head : list(r)$
$\mathsf{else}\ list(r)$

$=$ $\{\![$ Lemma 3 $]\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ s.head : list(s.tail)$
$\mathsf{else}\ list(r)$

$=$ $\{\![$ fold with definition of $list$ $]\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ list(s)\ \mathsf{else}\ list(r)$

$=$ $\{\![$ if propagation $]\!\}$

$list(\mathsf{if}\ c(p.head)\ \mathsf{then}\ s\ \mathsf{else}\ r)$

So the pointer algorithm derived is:

$$\begin{aligned} filter_p\ c\ (m, L) = \mathsf{if}\ \ m == \diamond\ \ &\mathsf{then}\ (\diamond, L) \\ &\mathsf{else\,if}\ \ c(L_{head}(m)) \\ &\qquad \mathsf{then}\ p.tail := filter_p\ c\ (L_{tail}(m), L) \\ &\qquad \mathsf{else}\ filter_p\ c\ (L_{tail}(m), L) \end{aligned}$$

3.10 Splitting a list

Now we want to derive a simultaneous destructive filter function, that partitions the list into two lists. One of these consisting of all elements of the original list satisfying a predicate c.

```
split c []       = ([],[])
split c (x:xs) = let (as,bs) = split c xs
                 in if c x then (x:as,bs)
                           else (as,x:bs)
```

To give a specification of $split_p$ we additionally need a (reasonable) abstraction function $pairlist$, that gets a pointer structure consisting of two entries and returns the pair of lists starting at these two entries:

$$pairlist(m, n, L) = (list(m, L), list(n, L))$$

The pointer algorithms $split_p$ can now be specified by the following equation:

$$pairlist(split_p\ c\ p) = split\ c\ list(p)$$

where p is defined the same as before. The case $m == \diamond$ is similar to the one in Section 3.3 and therefore left out here. We can choose $split_p\ c\ (\diamond, L) = (\diamond, \diamond, L)$. The second case is

Case $m \neq \diamond$**:**

$split\ c\ list(p)$

$=$ $\{\!\!\{$ unfold definitions of $list$ and $split$ $\}\!\!\}$

$\mathsf{let}\ (as, bs) = split\ c\ list(p.tail)$
$\mathsf{in\ if}\ c(p.head)\ \mathsf{then}\ (p.head : as, bs)$
$\mathsf{else}\ (as, p.head : bs)$

$=$ $\{\!\!\{$ fold with spec. of $split_p$; Choose q $\in split_p\ c\ (L_{tail}(m), L)$ $\}\!\!\}$

$\mathsf{let}\ (as, bs) = pairlist(q)$
$\mathsf{in\ if}\ c(p.head)\ \mathsf{then}\ (p.head : as, bs)$
$\mathsf{else}\ (as, p.head : bs)$

$=$ $\{\!\!\{$ let $(u, v, M) = q \Rightarrow as = list(u, M), bs = list(v, M)$ $\}\!\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ (p.head : list(u, M), list(v, M))$
$\mathsf{else}\ (list(u, M), p.head : list(v, M))$

$=$ $\{\!\!\{$ set $r = p.tail := (u, M)$ and $s = p.tail := (v, M)$ and Lemma 4.1 $\}\!\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ (r.head : list(u, M), list(v, M))$
$\mathsf{else}\ (list(u, M), s.head : list(v, M))$

$=$ $\{\!\!\{$ Lemma 3 $\}\!\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ (r.head : list(r.tail), list(v, M))$
$\mathsf{else}\ (list(u, M), s.head : list(s.tail))$

$=$ $\{\!\!\{$ fold with definition of $list$ (twice) $\}\!\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ (list(r), list(v, M))$
$\mathsf{else}\ (list(u, M), list(s))$

$=$ $\{\!\!\{$ $(v, M) \not\vdash m$ and $(u, M) \not\vdash m$ and twice Lemma 4.2 $\}\!\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ (list(r), list(v, sto(r)))$
$\mathsf{else}\ (list(u, sto(s)), list(s))$

$=$ $\{\!\!\{$ fold with definition of $pairlist$ (twice) $\}\!\!\}$

$\mathsf{if}\ c(p.head)\ \mathsf{then}\ pairlist(ptr(r), v, sto(r))$
$\mathsf{else}\ pairlist(u, ptr(s), sto(s))$

$=$ $\{\!\!\{$ if propagation $\}\!\!\}$

$pairlist(\mathsf{if}\ c(p.head)\ \mathsf{then}\ (ptr(r), v, sto(r))\ \mathsf{else}\ (u, ptr(s), sto(s)))$

So we have derived the pointer algorithm $split_p$ breaking a list into two parts whereas the elements of the first satisfying predicate c:

$$
\begin{array}{l}
split_p\ c\ (m, L) = \mathsf{if}\ \ m == \diamond\ \ \mathsf{then}\ (\diamond, \diamond, L) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathsf{let}\ (u, v, M) = split_p\ c\ (L_{tail}(m), L) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in\ if}\ \ c(L_{head}(m)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{then}\ (m, v, (m \overset{tail}{\rightarrow} u) \mid M) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ (u, m, (m \overset{tail}{\rightarrow} v) \mid M)
\end{array}
$$

4 Patterns

After deriving several standard algorithms over lists in Section 3 we now investigate what has to be done to achieve imperative versions of all these recursive ones. Therefore we study what sort of function patterns resulted from the transformations.

4.1 Singly linear-recursive functions

Most of the derived functions like e.g. `cat` (see [8]), `insert` or `del` have exactly one linear recursive call to itself. So all the other alternatives $H_0, \ldots, H_n$ do not include an application of F. The generalized pattern looks like this:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ C_0(x)\ \ \mathsf{then if}\ C_1(x)\ \ \mathsf{then}\ \ldots \mathsf{if}\ C_n(x)\ \ \mathsf{then}\ \Phi(F(K(x)), E(x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ H_n(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ H_1(x) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ H_0(x)
\end{array}
$$

The transformation pattern derived in [8] is only usable for rather simple recursive algorithms having only one terminating branch. If we increase the complexity of treated algorithms only little, for example by considering `del` that deletes the first element from a list, the scheme is not usable anymore. However we are able to bring algorithms using alternatives H_0 to H_n as in the previously described general pattern into a form the Paterson-Hewitt scheme is applicable and so derive a more sophisticated pattern. The original scheme is

$$
\begin{array}{l}
F(x) \equiv \mathsf{if}\ B(x)\ \ \mathsf{then}\ \phi(F(K(x)), E(x)) \\
\qquad\qquad\qquad\quad \mathsf{else}\ H(x) \\
\hline
F(x) \equiv \mathsf{var}\ vx = x \\
\qquad\quad \mathsf{if}\ B(x)\ \ \mathsf{then}\ \ \mathsf{while}\ B(K(vx))\ \mathsf{do}\ vx = K(vx) \\
\qquad\qquad\qquad\qquad\quad \psi_k(ptr(E(x)), ptr(E(vx)), H(K(vx))) \\
\qquad\qquad\qquad\quad \mathsf{else}\ H(x)
\end{array}
\quad [\ \text{Conditions}
$$

With abbreviations $s_i = ptr(E(K^i(x)))$ the conditions are

$$\forall i \in \{0, \ldots, n0\} : ptr(z) = s_i = s_{i-1} \vee (s_i \neq s_{i-1} \wedge (s_{i-1}, s_i) \in sto(z))$$

To bring the general pattern into the right form we need the sequential logical operators of Definition 9. We now prove a simple rule, that allows us to bring a singly recursive function with several alternatives into the form the Paterson-Hewitt transformation scheme needs. The simplest case is a sequential conjunction with two arguments:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ B(x)\ \&\&\ \ C(x)\ \ \mathsf{then}\ K(x) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ L(x) \\
\hline
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then if}\ C(x)\ \ \mathsf{then}\ K(x) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ L(x) \\
\qquad\qquad\quad \mathsf{else if false}\ \ \mathsf{then}\ K(x) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ L(x) \\
\hline
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then if}\ C(x)\ \ \mathsf{then}\ K(x) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ L(x) \\
\qquad\qquad\quad \mathsf{else}\ L(x)
\end{array}
\begin{array}{l}
\\ \\ [\ \text{normalization of condition} \\ \\ \\ \\ [\ \text{elimination of condition} \\ \\ \\
\end{array}
$$

This rule can be applied iteratively and if we choose a case distinction between all alternatives $H_0 \ldots H_n$ as $L(x)$ we get immediately:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ C_0(x)\ \ \mathsf{then}\,\mathsf{if}\ C_1(x)\ \ \mathsf{then}\ \ldots\,\mathsf{if}\ C_n(x)\ \ \mathsf{then}\,\Phi(F(K(x)), E(x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,H_n(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,H_1(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,H_0(x) \\
\hline
F(x) = \mathsf{if}\ C_0(x)\,\&\&\,C_1(x)\ldots\&\&\,C_n(x)\ \ \mathsf{then}\,\Phi(F(K(x)), E(x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ \mathsf{if}\,\neg C_0(x)\,\mathsf{then}\,H_0(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{elsif}\,\neg C_1(x)\,\mathsf{then}\,H_1(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{elsif}\,\neg C_{n-1}(x)\,\mathsf{then}\,H_{n-1}(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,H_n(x)
\end{array}
$$

Which has by abbreviations

$$
\begin{array}{l}
B(x) = C_0(x)\,\&\&\,C_1(x)\,\&\&\ldots\&\&\,C_n(x) \\
H(x) = \mathsf{if}\,\neg C_0(x)\,\mathsf{then}\,H_0(x) \\
\qquad\quad \mathsf{elsif}\,\neg C_1(x)\,\mathsf{then}\,H_1(x) \\
\qquad\quad \ldots \\
\qquad\quad \mathsf{elsif}\,\neg C_{n-1}(x)\,\mathsf{then}\,H_{n-1}(x) \\
\qquad\quad \mathsf{else}\,H_n(x)
\end{array}
$$

the form of Scheme 4.1.

Example: Transformation of `del`

We show at the example of `del` how the transformation scheme can be used to derive pointer algorithms of singly recursive functions with several alternative cases. In Section 3 we have derived a pointer algorithm that meets the specification of `del`. By simple transformations we can bring this function into the right form to be able to apply the extended Paterson-Hewitt scheme.

$$
\begin{array}{l}
del_p\ a\ p = \mathsf{if}\ m \neq \diamond\ \ \mathsf{then}\,\mathsf{if}\ m.head \neq a\ \ \mathsf{then}\,p.tail := del_p\ a\ (m.tail, L) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,p.tail \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\,(\diamond, L) \\
\hline
del_p\ a\ p = \mathsf{if}\ m \neq \diamond\ \ \&\&\ \ m.head \neq a\ \ \mathsf{then}\,p.tail := del_p\ a\ (m.tail, L) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,\mathsf{if}\ m == \diamond\ \ \mathsf{then}\,(\diamond, L) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\,(m.tail, L) \\
\hline
del_p\ a\ p = \mathsf{var}\,va = a \\
\qquad\qquad \mathsf{var}\,vm = m \\
\qquad\qquad \mathsf{var}\,vL = L \\
\qquad\qquad \mathsf{if}\ \ m \neq \diamond\ \ \&\&\ \ m.head \neq a\,\mathsf{then} \\
\qquad\qquad\quad \mathsf{while}\,vm.tail \neq \diamond\ \ \&\&\ \ vm.tail.head \neq a \\
\qquad\qquad\qquad \mathsf{do}\,(va, vm, vL) := (va, vm.tail, vL) \\
\qquad\qquad\quad \Psi_{tail}(m, vm, \mathsf{if}\ vm.tail == \diamond\,\mathsf{then}\,(\diamond, L)\,\mathsf{else}\,(vm.tail.tail, L)) \\
\qquad\qquad \mathsf{elsif}\ m == \diamond\ \ \mathsf{then}\,(\diamond, L) \\
\qquad\qquad\qquad\qquad \mathsf{else}\,(m.tail, L)
\end{array}
$$

[Previously proven scheme

[P.H.

Some simplification and elimination of unchanged variables yields the following algorithm:

$$
\begin{array}{l}
\boxed{
\begin{array}{l}
del_p\ a\ p = \mathsf{var}\, vm = m \\
\qquad\quad \mathsf{if}\ \ m \neq \diamond \wedge m.head \neq a\, \mathsf{then} \\
\qquad\qquad \mathsf{while}\, vm.tail \neq \diamond \wedge vm.tail.head \neq a\, \mathsf{do}\, vm := vm.tail \\
\qquad\qquad \mathsf{if}\, vm.tail == \diamond\ \mathsf{then}\, (m, (vm \overset{tail}{\rightarrow} \diamond) \mid L) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\, (m, (vm \overset{tail}{\rightarrow} vm.tail.tail) \mid L) \\
\qquad\quad \mathsf{elsif}\, m == \diamond\ \mathsf{then}\, (\diamond, L) \\
\qquad\qquad\qquad\quad \mathsf{else}\, (m.tail, L)
\end{array}
}
\end{array}
$$

Although this worked pretty well, the scheme can not be used to get an iterative variant of algorithms that change more than one link of the pointer structure. Here additional work has to be done to reach a higher level of abstraction.

4.2 Multiple linear-recursive functions

Functions that call themselves in distinct branches using different parameters depending on the situation like e.g. `merge` have the following form: This is a special

$$
\begin{array}{l}
F(x) = \mathsf{if}\, C_0(x)\ \ \mathsf{then}\ \ldots \mathsf{if}\, C_n(x)\ \ \mathsf{then}\, \mathsf{if}\, B(x)\ \ \mathsf{then}\, \Phi(F(K_0(x)), E_0(x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\, \Phi(F(K_1(x)), E_1(x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\, H_n(x) \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\quad \mathsf{else}\, H_0(x)
\end{array}
$$

Figure1. Most general function pattern

case of singly recursive functions if we defer the test $B(x)$ inside the evaluation of Φ using the conditional operator $\_\ ?\ \_\ :\ \_$ known from several programming languages as abbreviation. Note, that this transformation is only valid if $B(x)$ does not have any side effects.

$$\Phi(F(B(x)\ ?\ K_0(x) : K_1(x)), B(x)\ ?\ E_0(x) : E_1(x))$$

So here the same task as in Section 4.1 has to be treated. Again we need a pattern that supports the transformation of algorithms that change more than one link.

4.3 Mixed functions

Some functions like e.g. `delete` or `filter` not only call themselves in a linear recursive way but also tail-recursively in an other branch. The former mimics the scan through the structure and the latter overlooks some linked addresses. The abstract form here is:

$$
\boxed{
\begin{array}{l}
F(x) = \mathsf{if}\, C_0(x)\ \ \mathsf{then}\ \ldots \mathsf{if}\, C_n(x)\ \ \mathsf{then}\, \mathsf{if}\, B(x)\ \ \mathsf{then}\, \Phi(F(K(x)), E(x)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\, F(K(x)) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\, H_n(x) \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\quad \mathsf{else}\, H_0(x)
\end{array}
}
$$

This is the most general pattern. If we choose $B(x) = true$ we get the scheme described in Section 4.1. As the pattern from Section 4.2 is a special case of the singly recursive one, the big task is to find a transformation pattern for such mixed function schemes that change several links.

5 Open Problems and questions

In this section we deal with some problems and questions arising from the previous observations. This includes also ideas for improvement and tasks to work on.

5.1 Non-intuitive abstraction functions

The demand for reasonable abstraction functions does not avoid functions that are defined non-intuitively. So think for example about an abstraction function $list'$ for lists that skips every second address linked by the tail selector:

$$\begin{aligned} list'(p) &= \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ [] \\ &\qquad\qquad \mathsf{else}\ p.head : skip(p.tail) \\ skip(p) &= \mathsf{if}\ ptr(p) = \diamond\ \mathsf{then}\ [] \\ &\qquad\qquad \mathsf{else}\ list'(p.tail) \end{aligned}$$

The reason to derive destructive algorithms rather than copying ones is the demand for implementations that save as much memory as possible. So there should be a condition that marks for example $list$ as the abstraction function (for singly-linked lists) with better memory performance than $list'$. Clearly to be comparable the functions have to produce the same data type and use the same selectors. We will call such abstraction functions related. Then they can be ordered by observing the reachable parts needed to produce the same abstract object.

Definition 10. *Let F and G be related reasonable abstraction functions. Then we define an order by:*

$$F \sqsubseteq G \overset{\text{def}}{\Leftrightarrow} [\forall p, q.\ F(p) = G(q) \Rightarrow reach_F(p) \subseteq reach_G(q)]$$

(here we also have to use the previously defined refined version of $reach$ to only include the selectors used by the abstraction function).

We think that this is a reasonable condition that should hold for abstraction functions if we want to derive algorithms with least amount of memory used. It is not clear if there are rules that have to be fulfilled by an abstraction function to be contained in such a minimal set.

5.2 More sophisticated data structures

In [8] we have shown that algorithms on trees are not as different to list processing functions as it seems on first sight. By coding the treatment of more than one selector into a case distinction we were able to derive an algorithm for insertion into a tree. It seems that this is a generally applicable method to deal with data structures based on several selectors.

Another question is about doubly-linked data structures like lists that have a link to the successor and the predecessor or cyclic lists. Similarly there are trees that support links to the parent nodes. The question now is, if it is possible to code the doubly-linkedness into the abstraction function and so derive a corresponding algorithm from the same specification. Or is it possible to add an extra condition to the derivation process that cares for all these things? To treat cyclic lists for example maybe one can use the more sophisticated abstraction function $clist$ that uses the standard abstraction function $list$ for lists but previously applies an auxiliary function that manages the cycles:

$$clist(p) \overset{\text{def}}{\Leftrightarrow} list(breakcycle(p))$$

This would avoid huge extra effort for deriving such algorithms and make the framework more general.

5.3 Red and green algorithms

To be able to reason about the memory usage of pointer algorithms we define two different types of algorithms.

Definition 11. *Algorithms that preserve referential transparency by doing necessary copying of arguments are called green algorithms.*

Definition 12. *Algorithms that use destructive updates and so do not preserve referential transparency are called red algorithms.*

By definition it is evident, that there are no algorithms that are both red and green. If we use the number of copying actions as ordering we can compare implementations of an algorithm by their space efficiency. So for example the least element among the red implementation of an algorithm is the one that reuses as much storage as possible and only copies or allocates new cells if absolutely necessary. This mostly is the intention behind the term *destructive implementation* of an algorithm. On the other hand the greatest element of green algorithms copies all needed data and so the returned data structure is completely new.

The main question here is how to achieve an algorithm of the desired type. There should be no problem to derive a green algorithm by simple copying the whole data structure and afterwards call a derived version of the algorithm (green or red) on the copy. But is there a locally applicable rule that asserts during the derivation process that the resulting algorithm is a red or green one?

5.4 Composability of solutions

A practically useful derivation framework should be scalable. So we should be able to compose the simple components we have derived in Section 3 into more complex ones. The main problem here is aliasing. This notion describes the state of two different variables pointing to the same address in memory. Take as an example the function `dup` that duplicates a list by attaching one to the end of the other. In a functional programming language the simple property

```
dup xs = cat xs xs
```

holds by referential transparency. In our context `dup` can only be built using `cat` if the pointer implementation of `cat` is a green algorithm as defined above. So destructive updates destruct also the composability of functions. This implies that more complex destructive examples have to be transformed as a whole. Here a case study of deriving a more sophisticated algorithm that is composed of several operations could give more insight.

Nevertheless, there are cases where we can achieve composable solutions if it is possible to assure that there is no aliasing. For example in

```
let a1 = ...
    a2 = ...
    .. ...
    an = ...
in  f(a1,a2,...,an)
```

when each *let*-variable is used only once and all `ai` only depend on `ak` with `k<i`. So it is rather impossible to derive a pointer implementation for quick-sort from the standard functional specification:

```
qsort []     = []
qsort (a:as) = qsort [x | x<-as, x<=a]
               ++ [a]
               ++ qsort [x | x<-as, x>a]
```

The reason is the doubled appearance of `as` on the right side. But we can eliminate aliasing calculating the two lists in one step:

```
qsort [] = []
qsort (a:as) = let (xs,ys) = split (<=a) as
                   xs' = qsort xs
                   ys' = qsort ys
               in xs' ++ [a] ++ ys'
```

An other solution to single out specifications using aliases maybe to show termination for the abstraction function. So we can for example show

Lemma 5. *$acyclic(p) \Rightarrow list(p)$ terminates*

Proof. We take $reach_{list}$ as the termination function. Now it is easy to show that

$$reach_{list}(p.tail) \subset reach_{list}(p)$$

which implies that $list(p)$ terminates (For a simple proof in Pointer Kleene Algebra see [10]).

So we can say that `cat` terminates exactly if both of its parameters are acyclic lists and do not share any elements. This definitely is not the case in the previous definition of `dup`.

5.5 Efficiency

There are a lot of well-known performance improving techniques for functional languages. Think for example of an efficient implementation of reverse by using an aggregation variable or deforestation techniques. The question now is, if this performance improvement has a direct effect on the resulting pointer algorithm or if there are similar procedures on the imperative side. But even if this is the case we believe that the resulting techniques are much more complicated in the imperative world than in the functional one due to possible side-effects. So the change of worlds should be deferred as long as possible.

5.6 Iterative versions

As we derive all the algorithms from functional specification and there the only means of iteration is recursion, all resulting pointer algorithms are defined recursively. Certainly this is not what one wants in an imperative setting. This is because recursive calls are not very efficient in time and space. Just as there is no control over the call stack. So stack overflows from unbounded recursions are very likely.

But this is an area where we already made some progress. In [8] we described how a class of pointer algorithms can be transformed into an imperative version using the transformation of Paterson and Hewitt. There the type of algorithms scanning through a data structure to find the place where they have to change exactly one link is covered. Although functions like insertion into and concatenation of lists belong to this class it is evident that there are several other forms of algorithms. A more sophisticated transformation pattern is work in progress and will be presented soon.

6 Summary

We have presented a number of list processing algorithms and derived for each a variant working on pointer structures. Inspecting the resulting patterns we investigated what has to be done to get a framework that is able to transform all of this functions into imperative pointer algorithms. Additionally we proposed some enhancements to the pointer algebra presented by Möller and pointed out problems and things to do as a sort of working plan for the future.

A Correction of a transformation pattern

In [16] on page 329 the derivation pattern used in Section 3.2 was given wrongly. The correct scheme is:

$$
\begin{array}{l}
f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\ \mathsf{else\,if}\ C(x)\ \ \mathsf{then}\ G(x) \\
\qquad\qquad\qquad\ \ \mathsf{else}\ f(K(x)) \\
\hline
f(x) = vx := x \\
\qquad \mathsf{while}\ \neg B(x)\ \&\&\ \neg C(x)\ \mathsf{do}\ vx := K(x) \\
\qquad \mathsf{if}\ B(x)\ \mathsf{then}\ H(x)\ \mathsf{else}\ G(x)
\end{array}
$$

Proof.

$$
\begin{array}{l}
f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\ \mathsf{else\,if}\ C(x)\ \ \mathsf{then}\ G(x) \\
\qquad\qquad\qquad\ \ \mathsf{else}\ f(K(x)) \\
\hline
f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\ \mathsf{else\,if}\ \neg C(x)\ \ \mathsf{then}\ f(K(x)) \\
\qquad\qquad\qquad\ \ \mathsf{else}\ G(x) \\
\hline
f(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\ \mathsf{else\,if}\ \neg(B(x)\,||\,C(x))\ \ \mathsf{then}\ f(K(x)) \\
\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ G(x) \\
\hline
f(x) = \mathsf{if}\ \neg(B(x)\,||\,C(x))\ \ \mathsf{then}\ f(K(x)) \\
\qquad\qquad\qquad\ \ \mathsf{else\,if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ G(x) \\
\hline
f(x) = \mathsf{if}\ \neg(B(x)\,||\,C(x))\ \ \mathsf{then\,if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ G(x) \\
\qquad\qquad\qquad\ \ \mathsf{else}\ f(K(x)) \\
\hline
f(x) = \mathsf{if}\ \neg(B(x)\,||\,C(x))\ \ \mathsf{then\,if}\ B(x)\ \ \mathsf{then}\ H(x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ G(x) \\
\qquad\qquad\qquad\ \ \mathsf{else}\ f(K(x)) \\
\hline
f(x) = vx := x \\
\qquad \mathsf{while}\ \neg B(x)\ \&\&\ \neg C(x)\ \mathsf{do}\ vx := K(vx) \\
\qquad \mathsf{if}\ B(vx)\ \ \mathsf{then}\ H(vx) \\
\qquad\qquad\ \ \mathsf{else}\ G(vx)
\end{array}
\begin{array}{l}
\\ \\ \\ [\text{ Change of branches} \\
\\ \\ \\ [\ \neg(B(x)\,||\,C(x)) = \neg B(x)\ \&\&\ \neg C(x) \\
\\ \\ \\ [\ \neg(B(x)\,||\,C(x)) \wedge B(x) = \text{false} \\
\\ \\ \\ [\text{ Change of branches} \\
\\ \\ \\ [\ \neg B(x)\ \&\&\ \neg C(x) = \neg(B(x)\,||\,C(x)) \\
\\ \\ \\ [\text{ recursion to iteration} \\
\\ \\ \\
\end{array}
$$

References

1. F.L. Bauer and H. Wössner. *Algorithmic Language and Program Development*. Springer-Verlag, 1982.
2. A. Bijlsma. Calculating with pointers. *Science of Computer Programming*, 12(3):191–206, September 1989.
3. R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 1998. 2nd edition.
4. R. S. Bird. Functional pearl: Unfolding pointer algorithms. *Journal of Functional Programming*, 11(3):347–358, May 2001.
5. R. Bornat. Proving pointer programs in Hoare logic. In R. Backhouse and J. Nuno Oliveira, editors, *Mathematics of Program Construction, 5th International Conference, MPC 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer-Verlag, 2000.
6. R. M. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, Edinburgh, Scotland, 1972.
7. M. Butler. Calculational derivation of pointer algorithms from tree operations. *Science of Computer Programming*, 33(3):221–260, March 1999.
8. T. Ehm. Transformational construction of correct pointer algorithms. In D. Bjorner, M. Broy, and A.V. Zamulin, editors, *Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, July 2001.
9. T. Ehm. Kleene algebras and pointer structures. Technical report, Institut für Informatik, Universität Augsburg, 2003. To appear.
10. T. Ehm. Pointer Kleene Algebra. Submitted to RelMiCS, 2003.
11. C.A.R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.
12. B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21:57–90, 1993.
13. B. Möller. Calculating with pointer structures. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 24–48. Proc. IFIP TC2/WG2.1 Working Conference, Le Bischenberg, Feb. 1997, Chapman & Hall, 1997.
14. B. Möller. Calculating with acyclic and cyclic lists. In A. Jaoua and G. Schmidt, editors, *Relational Methods in Computer Science. Int. Seminar on Relational Methods in Computer Science, Jan 6–10, 1997 in Hammamet*, volume 119 of *Information Sciences — An International Journal*, pages 135–154, 1999.
15. J.M. Morris. A general axiom of assignment. In *Theoretical Foundations of Programming Methodology*, volume 91 of *NATO Advanced Study Institutes Series C Mathematical and Physical Sciences*, pages 25–34. Dordrecht, Reidel, 1981.
16. H. A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Monographs in Computer Science. Springer-Verlag, Berlin, 1990.
17. M. S. Paterson and C. E. Hewitt. Comparative schematology. In *Rec. Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 119–128, Woods Hole, MA, December 1970.
18. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In J. Davies, B. Roscoe, and J. Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.