Transformational Construction of Correct Pointer Algorithms

Thorsten Ehm

Institut für Informatik
Universität Augsburg
Ehm@Informatik.Uni-Augsburg.DE

Abstract. This paper shows how to use the transformation of Paterson and Hewitt to improve the memory and operations used in a pointer algorithm. That transformation scheme normally is only of theoretical interest because of the inefficient performance of the transformed function. However we present a method how it can be used to decrease the amount of selective updates in memory while preserving the original runtime performance. This leads to a general transformation framework for the derivation of a class of pointer algorithms.

Keywords: Paterson/Hewitt, program transformation, pointer algorithms, destructive updates

1 Introduction

Algorithms on pointer structures are often used in lower levels of implementation. Although in modern programming languages (e.g. in Java) they are hidden from the programmer, they play a significant rôle at the implementation level due to their performance. But this advantage is bought at high expense. Pointer algorithms are very error-prone and so there is a strong demand for a formal treatment and development process for pointer algorithms. There are some approaches to achieve this goal.

Several methods [2,10,11] use the wp-calculus to show the correctness of pointer algorithms. There only properties of the algorithms are proved but the algorithms are not derived from a specification. So the developer has to provide an implementation. In these approaches proving trivialities may last several pages. Butler [7] investigates how to generate imperative procedures from applicative functions on abstract trees. To achieve this he enriches the trees by paths to eliminate recursion. A recent paper by Bornat [5] shows that it is possible, but difficult to reason in Hoare logic about programs that modify data structures defined by pointers. Reynolds [17] also uses Hoare logic and tries to improve a method described in a former paper of Burstall [6] to show the correctness of imperative programs that alter linked data structures.

In [13] Möller proposed a framework based on relation algebra to derive pointer algorithms from a functional specification. He shows that the rules presented also are capable of handling more difficult multi-linked data structures

D. Bjørner, M. Broy, and A. Zamulin (Eds.): PSI 2001, LNCS 2244, pp. 116–130, 2001.

like doubly-linked lists or trees. However the derived algorithms are still recursive. Our goal is to improve this method by showing how to derive imperative algorithms and so achieve a more complete calculus for transformational derivation of pointer algorithms. Based on the method by Möller a recent paper by Richard Bird [4] shows how one can derive the Schorr-Waite marking algorithm in a totally functional way.

This paper shows how to use the transformation of Paterson and Hewitt (P & H) to derive imperative pointer algorithms. To achieve this we take the recursive pointer algorithms derived from functional descriptions using the method of Möller. These are transformed via the P & H transformation scheme into an imperative version. Despite the inefficient general runtime performance of the scheme that results from P & H, we get well performing algorithms. As a side effect the amount of selective updates in memory is improved by eliminating ineffective updates that are only used to pass through the pointer structure. This is not a trivial task, because in general it is not decidable if an update really changes links of the pointer structure. Some systems do such optimization during runtime but not in such an early state of software development.

We will show how these aims can be achieved for a class of pointer algorithms that first pass through a pointer structure to find the position where they have to do some proper changes. A similar transformation scheme for a class of algorithms that not only alter but also delete links is in preparation. Be aware that we are not interested in algorithms that do not alter the link structure but only the contents of the nodes (like for example `map`). The advantage of the presented method over the previously mentioned approaches using wp-calculus or Hoare logic are apparent. All these methods provide correct algorithms. However the presented one treats a whole class of functions whereas the other methods have to be applied on every new algorithm. You also do not have to provide an implementation for a specification which is a time-consuming task. Not least, the transformational approach is more likely to be the easier one to automate.

The paper is structured as follows: Section 2 defines pointer structures and describes some rules needed for the derivation of pointer algorithms from a functional version. Section 3 presents as a running example the function *cat* that appends a list to another and explains the problem. Section 4 gives a short overview of the methods to get a tail-recursive version from a linear-recursive algorithm. The transformation scheme of Paterson and Hewitt is introduced in Section 5. Section 6 describes the evolution of a transformation scheme to derive a non recursive imperative pointer algorithm from a recursive one like the one presented in Section 3. Some applications of the scheme to lists and trees are shown in Section 7. Finally Section 8 concludes the presented methodology. Appendix A includes the theoretical basics and the proofs.

2 Pointer Structures and Operations

To make this abstract self-contained as far as possible we present a short introduction to pointer structures and how they are used in [13].

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 from a set $\mathcal{A}$ 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 from addresses to node values in a set $\mathcal{N}_j$ such as *Integer* or *Boolean*. Each relation represents a selector on the records like e.g. *head* and *tail* for lists with functionality $\mathcal{A} \to \mathcal{N}_j$ and $\mathcal{A} \to \mathcal{A}$, respectively.

Each abstract object implemented 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

$$ptr(n, L) = n \quad \text{and} \quad sto(n, L) = L$$

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 1. $R \mid S \stackrel{\text{def}}{=} R \cup \overline{dom(R)} \bowtie S$

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 a "mini-store" that is a family of partial maps defined by:

Definition 2. $(x \stackrel{k}{\to} y) \stackrel{\text{def}}{=} \left\{ \begin{array}{ll} \{(x, y)\} & \textit{for selector } k \\ \emptyset & \textit{otherwise} \end{array} \right\}$

It is clear that overwriting a pointer structure with links already defined in it does not change the structure:

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

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

Definition 4.
For selector k *of type* $\mathcal{A} \to \mathcal{A}$: $(n, P).k \stackrel{\text{def}}{=} (P_k(n), P)$
For selector k *of type* $\mathcal{A} \to \mathcal{N}_j$: $(n, P).k \stackrel{\text{def}}{=} P_k(n)$

To have the possibility to insert new (unused) addresses into the data structure we define the newrec operator. Let k range over all selectors used in the modeled data structure. Then the operator newrec$(L, k : x_k)$ alters the store L

to have a new record previously not in L and each selector k pointing to x_k. So for example $\mathsf{newrec}(L, head : 3, tail : \diamond)$ returns a pointer structure (m, K) with m a new address previously not used in L and store K consisting of L united with two new links $(m \overset{head}{\rightarrow} 3)$ and $(m \overset{tail}{\rightarrow} \diamond)$. If it is clear from the context which selectors are used we only enumerate the respective components. So the previous expression becomes $\mathsf{newrec}(L, \langle 3, \diamond \rangle)$.

3 A Running Example and the Problem

In this section we want to use a functional description of list concatenation. This function serves as our running example during the derivation of the transformation scheme. We will use Haskell [3] notation to denote functional algorithms:

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

We assume that the two lists are acyclic and do not share any parts. So the following pointer algorithm can be derived by transformation using the method of [13]:

$$cat_p(m, n, L) = \mathsf{if}\ m \neq \diamond\ \mathsf{then}\ (m, L).tail := cat_p(L_{tail}(m), n, L)\ \mathsf{else}\ (n, L)$$

The two pointer structures (m, L) and (n, L) are representations of the two lists. Addresses m and n model the starting points, whereas L is the memory going with them. In other words m and n form links to the beginnings of two lists in memory L.

Note that this is only one candidate of possible implementations for the functionally described specification of `cat`. Because we are interested in algorithms performing minimal **destructive** updates we did not derive a persistent variant such as the standard, partially copying interpretation in functional languages. But that would also be possible.

We now have a linear recursive function working on pointer structures. But what we want is an imperative program that does not use recursion. By investigating the execution order of cat_p we can see, that cat_p calculates a term of the following form:

$$(m, L).tail := ((L_{tail}(m), L).tail := ...(n, L))$$

If you remember the definition of the := operator, this means that updates are performed from right to left.

$$(m \overset{tail}{\rightarrow} L_{tail}(m)) \mid (\ldots \mid ((L^k_{tail}(m) \overset{tail}{\rightarrow} n) \mid L) \ldots)$$

This shows that the derived algorithm uses the update operator not only to properly alter links but also to just pass through the structure while returning from the recursion. Figure 1 shows how these updates are done in the example of *cat* ($\Rightarrow$ denotes the actually altered links).

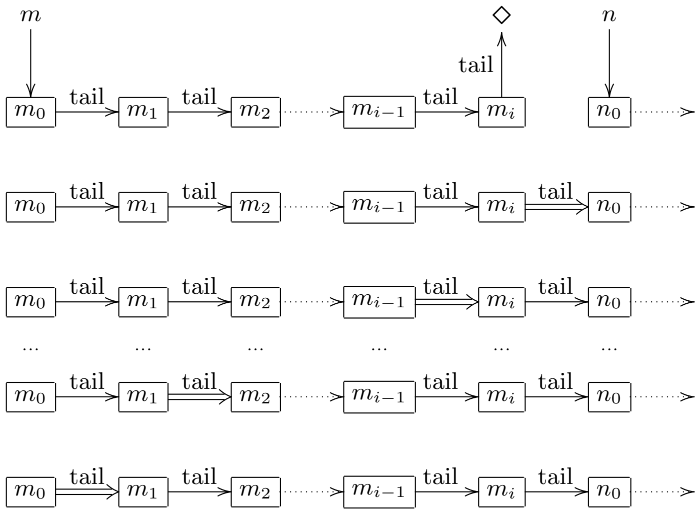

Fig. 1. Order of updates performed by *cat*

As we can see, there are several such updates that do not alter the pointer structure. For example $(m \stackrel{tail}{\to} L_{tail}(m))$ is already contained in L and does not change the pointer structure $(\ldots \mid ((L^k_{tail}(m) \stackrel{tail}{\to} n) \mid L)\ldots)$ if the previous updates do not affect this part of L. This is the case for several algorithms on pointer linked data structures, because most of them first have to scan the structure to find the position where they have to do the proper changes.

We now define the following abbreviations to get a standardized form for later transformations:

$$\begin{array}{lll} K(m,n,L) \stackrel{\text{def}}{=} (L_{tail}(m),n,L) & B(m,n,L) \stackrel{\text{def}}{=} m \neq \diamond & \phi_k(u,v) \stackrel{\text{def}}{=} v.k := u \\ H(m,n,L) \stackrel{\text{def}}{=} (n,L) & E(m,n,L) \stackrel{\text{def}}{=} (m,L) & \end{array}$$

Abbreviating (m, n, L) to x the derived pointer algorithm can then be written as

$$\begin{array}{ll} cat_p(x) = \textsf{if } B(x) & \textsf{then } \phi_{tail}(cat_p(K(x)), E(x)) \\ & \textsf{else } H(x) \end{array}$$

4 From Linear via Tail Recursion to While Programs

In transformational program design the transformation of a linearly recursive function to an imperative version always has two steps: First transform the linear recursion into tail recursion. Then apply a transformation scheme [15] like the following to get a while program:

$$f(x) = \textsf{if } B(x) \ \textsf{then } f(K(x)) \ \textsf{else } H(x)$$

$$\Updownarrow$$

$$f(x) = \textsf{var } vx := x; \ \textsf{while } B(vx) \ \textsf{do } vx := K(vx); \ H(vx);$$

But cat_p does not have tail recursive form. So we first have to find a way to transform cat_p into the right form. There are several schemes to derive a tail recursive variant from a linear recursive function [1]. One of the most popular is to change the evaluation order of parentheses in the calculated expression. To be able to do this one needs a function ψ that fulfills the equation $\phi(\phi(r,s),t) = \phi(r,\psi(s,t))$. To find such a ψ is possible only in very rare cases of ϕ. One of these is that ϕ is associative. In this case you can choose $\psi = \phi$. Another - similar - case is to change the order of operands. Here it is necessary that $\phi(\phi(r,s),t) = \phi(\phi(r,t),s)$ or more generally you need a ψ with $\phi(\psi(r,s),t) = \psi(\phi(r,t),s)$. The previously described rules assume that ϕ is good-natured enough to satisfy one of the properties mentioned. However, our function $\phi_{tail}(u,v)$ in cat_p does not show any of these properties. Another transformation uses function inversion to calculate the parameter values from the results. Here one only has to find an inverse $\overline{K}$ of K. But the function $K(m,n,L) \stackrel{\text{def}}{=} (L_{tail}(m),n,L)$ in general is not invertible. So is there no way to get a tail recursive version of cat_p ?

5 The Transformation Scheme of Paterson/Hewitt

In 1970 Paterson and Hewitt presented a transformation scheme that makes it possible to transform any linear recursive function into a tail recursive one [16]. This rule normally is only of theoretical interest because of the bad runtime performance of the resulting function. P & H applied the idea of the method mentioned in Section 4 using the inverse function $\overline{K}$ to make the step from K^{i+1} to K^i, but exhaustively recalculated K^i from the start. The evolving scheme is:

$$F(x) = \textsf{if } B(x) \ \textsf{then } \phi(F(K(x)),E(x)) \ \textsf{else } H(x)$$

$$\Updownarrow \quad \text{[P \& H]}$$

$$\begin{aligned}
F(x) = {} & G(n0, H(m0)) \ \textsf{where} \\
& (m0,n0) = num(x,0) \\
& num(y,i) = \textsf{if } B(y) \ \textsf{then } num(K(y),i+1) \ \textsf{else } (y,i) \\
& it(y,i) = \textsf{if } i \neq 0 \ \textsf{then } it(K(y),i-1) \ \textsf{else } y \\
& G(i,z) = \textsf{if } i \neq 0 \ \textsf{then } G(i-1,\phi(z,E(it(x,i-1)))) \ \textsf{else } z
\end{aligned}$$

The function num calculates the number of iterations that have to be done until the termination condition is fulfilled, as well as the final value. These values are

used by function G to change the evaluation order of the calculated term. For this, G uses the function it to iterate K to achieve the inverse $\overline{K}$ of K by doing one iteration less than has to be done for K. So G can start with the calculations done in the deepest recursion step first and then ascend from there using the inverse of K.

As we have seen, function it is only used to calculate the powers of K and we have $it(y,i) = K^i(y)$, so we can abbreviate $\phi(z, E(it(x, i-1)))$ to $\phi(z, E(K^{i-1}(x)))$ and simplify the scheme:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \phi_k(F(K(x)), E(x)) \\
\qquad\qquad\quad \mathsf{else}\ H(x) \\
\hline
\qquad\qquad\qquad\qquad \updownarrow \qquad\qquad\qquad\qquad\qquad\qquad [\ \text{Paterson/Hewitt II} \\
\hline
F(x) = G(n0, H(m0))\ \mathsf{where} \\
\qquad (m0, n0) = num(x, 0) \\
\qquad num(y,i) = \mathsf{if}\ B(y)\ \ \mathsf{then}\ num(K(y), i+1) \\
\qquad\qquad\qquad\qquad\quad \mathsf{else}\ (y,i) \\
\qquad G(i,z) = \mathsf{if}\ i \neq 0\ \ \mathsf{then}\ G(i-1, \phi_k(z, E(K^{i-1}(x)))) \\
\qquad\qquad\qquad\quad \mathsf{else}\ z
\end{array}
$$

This certainly is only a cosmetic change, because K^{i-1} has to be calculated exactly the same way as in the original transformation scheme. But this gives the basis to future simplification, because $K^{i-1}(x)$ is only used as a parameter for E and will be eliminated in further steps.

6 Deriving a General Transformation Scheme

We now present an application of the P & H transformation scheme to pointer algorithms using the function ϕ_k to pass through a pointer data structure. The whole derivation from a more theoretical point of view including all calculations and proofs can be found in the appendix.

By investigation of function $\phi_k((m,L),(n,L)) = (n, (n \xrightarrow{k} m) \mid L)$ we can see that ϕ_k updates the link starting from m via selector k and simultaneously sets n as the new starting entry of the resulting pointer structure. It is apparent that such a restricted function can not provide the simplification we aim to achieve, namely elimination of effect-less updates. So we use the technique of generalization and introduce a more flexible function $\psi_k(l,m,(n,L)) = (l, (m \xrightarrow{k} n) \mid L)$ that handles the altered address and the resulting entry independently. With this function we are in the position to eliminate the quasi-updates that do not alter the structure but are only used for passing through the pointer structure. One can say that ψ_k "eats up" the effect-less updates of ϕ_k:

Lemma 2. *If* $(t \xrightarrow{k} v) \subseteq (v \xrightarrow{k} u) \mid U$ *then for all* s

$$\psi_k(s, t, \phi_k((u,U),(v,U))) = \psi_k(s, v, (u, U))$$

Now we return to the P & H transformation scheme. There the function G applies ϕ_k so that this lemma can be used to simplify G. We apply the lemma to all

instances of G that only pass through the pointer structure. This means as long as the condition B is fulfilled we apply Lemma 2 and eliminate one application of ϕ_k. So the precondition of Lemma 2 has to hold for all those cases.

Lemma 3. *We abbreviate* $p^{(i)} \stackrel{\text{def}}{=} ptr(E(K^i(x)))$. *Then under the condition*

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

we can simplify $G(i, z)$ *to:*

$$G(i,z) = \mathsf{if}\ i \neq 0\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^{i-1}(x))), z)\ \ \mathsf{else}\ z$$

Remembering that K is the function performing the run through the pointer structure we can express the condition in human-understandable form. The pair $(p^{(i-1)}, p^{(i)})$ consists of the values under function E of two such successive elements that are met during the pass-through via K. Now, either these are equal which means the links form a cycle and the simplification is trivial. Or they are not equal and the memory already contains the link. Then an update using these values will not change anything and can be eliminated.

With $n0 = \min\{j : \neg B(K^j(x))\}$ this is a condition that in some cases can not be checked easily. But normally one proves a more general assertion. For function *cat* for example we have acylic lists and we can show that the condition holds for all successive pairs of elements in the list.

Now that G is not recursive anymore we can instantiate the application of G with its actual parameters. The test $i \neq 0$ is only calculated once. By inspection of num that calculates $n0$ (the actual argument for parameter i) we see that the inequality test can be done without $n0$:

Lemma 4. $n0 \neq 0 \Leftrightarrow B(x)$

So the scheme of Paterson and Hewitt simplifies to

$$\begin{aligned} F(x) = \mathsf{if}\ B(x)\ \ &\mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^{n0-1}(x))), H(m0)) \\ &\mathsf{else}\ H(m0) \\ \mathsf{where}\ \ &(m0, n0) = num(x, 0) \\ &num(y,i) = \mathsf{if}\ B(y)\ \ \mathsf{then}\ num(K(y), i+1) \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ (y, i) \end{aligned}$$

A straightforward induction shows that $m0 = K^{n0}(x)$. So the calculation of $m0$ can be done simultaneously with the calculation of K^{n0-1}. This is achieved by a slightly changed pair of functions num' and num'' that replace num. For this we extend the domain of K by a special element $\triangle_x$ with $K(\triangle_x) = x$ that models an imaginary predecessor of x under K:

$$\begin{aligned} num'(y) &= \mathsf{if}\ B(y)\ \ \mathsf{then}\ num''(y) \\ &\qquad\qquad\quad \mathsf{else}\ \triangle_y \\ num''(y) &= \mathsf{if}\ B(K(y))\ \ \mathsf{then}\ num''(K(y)) \\ &\qquad\qquad\qquad\quad \mathsf{else}\ y \end{aligned}$$

and obtain

Lemma 5. *$num'(x) = K^{n0-1}(x)$ and thus also $m0 = K(num'(x))$*

This is the basis for the following transformation:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(k0)), H(K(k0))) \\
\qquad\qquad\ \ \mathsf{else}\ H(K(k0)) \\
\qquad \mathsf{where}\ \ k0 = num'(x) \\
\qquad\qquad num'(y) = \mathsf{if}\ B(y)\ \ \mathsf{then}\ num''(y) \\
\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ \triangle_y \\
\qquad\qquad num''(y) = \mathsf{if}\ B(K(y))\ \ \mathsf{then}\ num''(K(y)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ y \\
\hline \\
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(k0)), H(K(k0))) \\
\qquad\qquad\ \ \mathsf{else}\ H(K(\mathsf{if}\ B(x)\ \mathsf{then}\ num''(x)\ \mathsf{else}\ \triangle_x)) \\
\qquad \mathsf{where}\ \ k0 = \mathsf{if}\ B(x)\ \ \mathsf{then}\ num''(x) \\
\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ \triangle_x \\
\qquad\qquad num''(y) = \mathsf{if}\ B(K(y))\ \ \mathsf{then}\ num''(K(y)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ y
\end{array}
\quad [\ \text{unfold } num' \text{ and } k0
$$

Although there is a term $E(k0)$ in the scheme the case that $E(\triangle_x)$ has to be evaluated can never be reached. So there is no need to define $E(\triangle_x)$. Now num'' is the only recursive function; it is even tail-recursive so that we are in the position to use the transformation scheme presented in Section 4 to achieve an imperative while program:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(k0)), H(K(k0))) \\
\qquad\qquad\ \ \mathsf{else}\ H(x) \\
\qquad \mathsf{where}\, k0 = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \ \mathsf{var}\, vx := x \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{while}\, B(K(vx))\, \mathsf{do}\, vx := K(vx) \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ vx \\
\qquad\qquad\qquad\qquad\ \ \mathsf{else}\ \triangle_x \\
\hline \\
F(x) = \mathsf{var}\, vx := x \\
\qquad\ \ \mathsf{if}\ B(x)\ \ \mathsf{then}\ \ \mathsf{while}\, B(K(vx))\, \mathsf{do}\, vx := K(vx) \\
\qquad\qquad\qquad\qquad\ \ \psi_k(ptr(E(x)), ptr(E(vx)), H(K(vx))) \\
\qquad\qquad\ \ \mathsf{else}\ H(x)
\end{array}
\quad [\ \text{Simplification}
$$

The scheme that has evolved from our calculations now is:

$$
\begin{array}{l}
F(x) = \mathsf{if}\ B(x)\ \ \mathsf{then}\ \phi(F(K(x)), E(x)) \\
\qquad\qquad\ \ \mathsf{else}\ H(x) \\
\hline \\
F(x) = \mathsf{var}\, vx := x \\
\qquad\ \ \mathsf{if}\ B(x)\ \ \mathsf{then}\ \ \mathsf{while}\, B(K(vx))\, \mathsf{do}\, vx := K(vx) \\
\qquad\qquad\qquad\qquad\ \ \psi_k(ptr(E(x)), ptr(E(vx)), H(K(vx))) \\
\qquad\qquad\ \ \mathsf{else}\ H(x)
\end{array}
\quad [\ \text{Conditions of Lemma 3}
$$

To return to our example in the previous sections we now can transform the recursive version of cat_p to an iterative program by using the derived scheme. First we check the applicability condition of our scheme abbreviating $T_i = L^i_{tail}(m)$:

$$\forall i \in \{0, \ldots, \min\{j : T_j = \diamond\}\} : n = T_i = T_{i-1} \vee (T_i \neq T_{i-1} \wedge (T_{i-1}, T_i) \in L)$$

The first disjunct is not fulfilled by the assumption that the two lists do not share any parts. But the second disjunct is true by acyclicity of p. So some simplification leads us to the imperative algorithm one has in mind:

$$\begin{array}{l}
cat_p(m,n,L) = \mathsf{var}\,(vm,vn,vL) := (m,n,L) \\
\qquad\qquad \mathsf{if}\ m \neq \diamond\ \mathsf{then}\ \mathsf{while}\ L_{tail}(vm) \neq \diamond\,\mathsf{do} \\
\qquad\qquad\qquad\qquad\qquad\qquad (vm,vn,vL) := (vL_{tail}(vm), vn, vL) \\
\qquad\qquad\qquad\qquad \psi_{tail}(m, vm, (vn, vL)) \\
\qquad\qquad\qquad\quad \mathsf{else}\ \ (n,L) \\
\hline
cat_p(m,n,L) = \mathsf{var}\, vm := m \\
\qquad\qquad \mathsf{if}\ m \neq \diamond\ \mathsf{then}\ \mathsf{while}\, L_{tail}(vm) \neq \diamond\,\mathsf{do}\, vm := L_{tail}(vm) \\
\qquad\qquad\qquad\qquad (m, (vm \overset{tail}{\rightarrow} n) \mid L) \\
\qquad\qquad\qquad\quad \mathsf{else}\ \ (n,L)
\end{array} \quad [\ vn, vL \text{ constant}$$

This formally derived program can directly be translated into a C program by mapping the pointer algebra operations into their corresponding C equivalents. Note that the memory L remains implicit:

```
list cat(list m,list n) {
  list vm = m;
  if (m) { while (vm->tail) vm=vm->tail;
           vm->tail=n;
           return m;
  }
  else return n;
}
```

7 Further Applications

In this section we want to show that the developed scheme is applicable to several algorithms passing through a pointer-linked data structure.

7.1 Insertion into a Sorted List

In [8] several algorithms on lists are derived with the calculus presented in [13]. We choose insertion into a sorted list as a first example. The function `insert` is defined like this:

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

We can bring the derived pointer algorithm $insert_p$ into the form needed by our scheme.

$$\begin{array}{l} insert_p(m,n,L) = \\ \quad \mathsf{if}\ n \neq \diamond \wedge L_{val}(m) > L_{head}(n)\ \ \mathsf{then}\ q.tail := insert_p(m, L_{tail}(n), L) \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{newrec}(L, \langle L_{val}(m), (n, L)\rangle) \end{array}$$

Now we can apply the scheme and after a simplification step achieve an imperative algorithm for insertion into a sorted list:

$$\begin{array}{l} insert_p(m,n,L) = \\ \quad \mathsf{var}\ vn := n \\ \quad \mathsf{if}\ \ (n \neq \diamond \wedge L_{val}(m) > L_{head}(n))\ \mathsf{then} \\ \qquad \mathsf{while}\ (L_{tail}(vn) \neq \diamond \wedge L_{val}(m) > L_{head}(vn))\ \mathsf{do}\ vn := L_{tail}(vn) \\ \qquad \psi_{tail}(n, vn, \mathsf{newrec}(L, \langle L_{val}(m), (vn, L)\rangle)) \\ \quad \mathsf{else}\ \mathsf{newrec}(L, \langle L_{val}(m), (n, L)\rangle) \end{array}$$

7.2 Insertion into a Tree

To show an example using a data structure different from lists we show how insertion into a tree can be derived from our scheme. It is nearly as easy as the other examples. We use the algorithm derived in [13] from the following functional specification:

```
ins x Empty        = Tree(Empty,x,Empty)
ins x Tree(l,y,r) = if x ≤ y then Tree(ins x l,y,r)
                             else Tree(l,y,ins x r)
```

The selectors used to model trees are val for node values as well as l and r for the left and right descendant. We abbreviate $u.val = L_{val}(u)$ and $p = (m, L)$ as before and get:

$$\begin{array}{l} ins_p\ x\ p = \mathsf{if}\ m = \diamond\ \ \mathsf{then}\ \mathsf{newrec}(L, \langle \diamond, x, \diamond\rangle) \\ \qquad\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{if}\ x \leq m.val\ \ \mathsf{then}\ p.l := ins_p\ x\ p.l \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ p.r := ins_p\ x\ p.r \end{array}$$

The algorithm can be transformed into the form needed by our scheme with the help of the conditional operator `_ ? _ : _` as used in several programming languages.

$$\begin{array}{l} ins_p\ x\ p = \mathsf{if}\ m \neq \diamond\ \ \mathsf{then}\ \phi_{(x \leq m.val?l:r)}(ins_p\ x\ (x \leq m.val?p.l : p.r)), p) \\ \qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{newrec}(L, \langle \diamond, x, \diamond\rangle)) \end{array}$$

Now we can use our scheme to achieve an imperative algorithm for insertion into a tree.

$$
\begin{array}{l}
ins_p\ x\ (m, L) = \\
\quad \mathsf{var}\ vm := m \\
\quad \mathsf{if}\ m \neq \diamond\ \mathsf{then}\ \mathsf{while}\ (h := x \leq vm.val?L_l(vm) : L_r(vm)) \neq \diamond\, \mathsf{do}\, vm := h \\
\qquad\qquad\qquad \psi_{(x \leq vm.val?l:r)}(m, vm, \mathsf{newrec}(L, \langle \diamond, x, \diamond \rangle)) \\
\quad \mathsf{else}\ \mathsf{newrec}(L, \langle \diamond, x, \diamond \rangle)
\end{array}
$$

Here we have used an assignment inside the condition of the while loop. Otherwise the algorithm would have to use the conditional operator _ ? _ : _ twice or introduce two new while loops. But we do not think this would make the algorithm more readable.

8 Conclusion

We have shown how the transformation of Paterson and Hewitt can be used to construct imperative algorithms on pointer-linked data structures. Although the transformation of Paterson and Hewitt normally is only of theoretical interest because of its very bad runtime behaviour, well-performing algorithms are derived. This leads to a general methodology for the derivation of pointer algorithms.

By the example algorithm for insertion into a tree it can be seen, that there is a need for more sophisticated schemes based on the presented one. It also is likely that algorithms changing more than one link such as deletion from a list can be treated the same way. For this, one has to divide the job into several parts altering only one link, applying the scheme and afterwards putting the parts together.

Further research will investigate this and other starting points to complete the methodology. Also a (semi-)automatic system checking the side-conditions and so supporting the developer of such algorithms is under development.

References

1. F.L. Bauer, H. Wössner: *Algorithmic Language and Program Development*, Springer, Berlin, 1984
2. A. Bijlsma: *Calculating with pointers*. Science of Computer Programming **12**, Elsevier 1989, 191–205
3. R. Bird: *Introduction to Functional Programming using Haskell*, 2nd edition, Prentice Hall Press, 1998
4. R. Bird: *Unfolding Pointer Algorithms*, to appear in Journal of Functional Programming, available from: `http://www.comlab.ox.ac.uk/oucl/work/richard.bird/publications`
5. R. Bornat: *Proving pointer programs in Hoare logic*. Proceedings of MPC 2000, Ponte de Lima , LNCS 1837, Springer 2000, 102–126
6. R. Burstall: *Some techniques for proving correctness of programs which alter data structures*. In B. Meltzer and D. Michie eds, Machine intelligence 7, Edinburgh University Press, 1972, 23–50
7. M. Butler: *Calculational derivation of pointer algorithms from tree operations*. Science of Computer Programming **33**, Elsevier 1999, 221–260
8. T. Ehm: *Case studies for the derivation of pointer algorithms*. to appear

9. C. A. R. Hoare: *Proofs of correctness of data representations.* Acta Informatica **1**, 1972, 271–281
10. J.M. Morris: *A general axiom of assignment.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 25–34
11. J.M. Morris: *Assignment and linked data structures.* Theoretical Foundations of Programming Methodology, NATO Advanced Study Institutes Series C Mathematical and Physical Sciences **91**, Dordrecht, Reidel, 1981, 35–51
12. B. Möller: *Towards pointer algebra.* Science of Computer Programming **21**, Elsevier, 1993, 57–90
13. B. Möller: *Calculating with pointer structures.* In: R. Bird, L. Meertens (eds.): Algorithmic languages and calculi. Proc. IFIP WG2.1 Working conference, Le Bischenberg. Chapman & Hall 1997, 24–48
14. B. Möller: *Calculating with acyclic and cyclic lists.* In A. Jaoua, G. Schmidt (eds.): Relational Methods in Computer Science. Int. Seminar on Relational Methods in Computer Science, Jan 6–10, 1997 in Hammamet. Information Sciences — An International Journal **119**, 1999, 135–154
15. H. Partsch: *Specification and transformation of programs. A formal approach to software development.* Monographs in Computer Science. Springer, 1990
16. M.S. Paterson, C.E. Hewitt: *Comparative Schematology.* Conference on Concurrent Systems and Parallel Computation Project MAC, Woods Hole, Massachuset, USA, 1970
17. J.C. Reynolds: *Intuitionistic reasoning about shared mutable data structures.* In: Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare, Palgrave, 2000

A Proofs

Here we show the proofs and calculations skipped in Section 6.
Proof of Lemma 2:

$$\begin{aligned}
& \psi_k(s,t,\phi_k((u,U),(v,U))) \\
= \ & \psi_k(s,t,(v,(v \xrightarrow{k} u) \mid U)) \\
= \ & (s,(t \xrightarrow{k} v) \mid ((v \xrightarrow{k} u) \mid U)) \\
\overset{\text{ann}}{=} \ & (s,(v \xrightarrow{k} u) \mid U) \\
= \ & \psi_k(s,v,(u,U))
\end{aligned}$$

The equality labeled with *ann* holds by annihilation if $(t \xrightarrow{k} v) \subseteq (v \xrightarrow{k} u) \mid U$. So the following conditions arise (We can restrict the calculation to k, the only selector used here):

$$\begin{aligned}
& (t \xrightarrow{k} v) \subseteq (v \xrightarrow{k} u) \mid U \\
\Leftrightarrow \ & \{(t,v)\} \subseteq \{(v,u)\} \cup \overline{dom(\{(v,u)\})} \bowtie U \\
\Leftrightarrow \ & (t,v) \in \{(v,u)\} \vee (t,v) \in \overline{\{v\}} \bowtie U \\
\Leftrightarrow \ & (t = v \wedge v = u) \vee (t \neq v \wedge (t,v) \in U) \\
\Leftrightarrow \ & (t = u = v) \vee (t \neq v \wedge (t,v) \in U) \qquad (*)
\end{aligned}$$

Proof of Lemma 3: We use Lemma 2 and induction over i:

$i = 0$**:** $G(0, z) = z$
$i \to i + 1$**:**

$$
\begin{array}{ll}
 & G(i+1, z) \\
= & \quad \{\![\text{ definition of } G \text{ and } i+1 \neq 0 \ ]\!\} \\
 & G(i, \phi_k(z, E(K^i(x)))) \\
= & \quad \{\![\text{ induction hypothesis }]\!\} \\
 & \mathsf{if}\ i \neq 0\ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^{i-1}(x))), \phi_k(z, E(K^i(x)))) \\
 & \qquad \mathsf{else}\ z \\
= & \quad \{\![\ (*) \text{ and lemma 2 }]\!\} \\
 & \mathsf{if}\ i \neq 0\ \mathsf{then}\ \psi_k(ptr(E(x)), ptr(E(K^i(x))), z) \\
 & \qquad \mathsf{else}\ z
\end{array}
$$

Proof of Lemma 4:

$$
\begin{array}{ll}
 & n0 \neq 0 \\
\Leftrightarrow & \quad \{\![\text{ definition of } n0 \ ]\!\} \\
 & snd(num(x, 0)) \neq 0 \\
\Leftrightarrow & \quad \{\![\text{ definition of } num \ ]\!\} \\
 & snd(\mathsf{if}\ B(x)\ \mathsf{then}\ num(x, 1)\ \mathsf{else}\ (x, 0)) \neq 0 \\
\Leftrightarrow & \quad \{\![\text{ propagation of } snd \text{ and } \neq 0 \ ]\!\} \\
 & (B(x) \wedge snd(num(x, 1)) \neq 0) \vee (\neg B(x) \wedge \mathtt{false}) \\
\Leftrightarrow & \quad \{\![\ snd(num(x, i)) = 0 \Leftrightarrow i = 0 \ ]\!\} \\
 & B(x)
\end{array}
$$

Proof of Lemma 5: First let $h(x, y, i) = K^{snd(num(y,i))-1}(x)$. So we can derive a recursive definition of h by:

$$
\begin{array}{ll}
 & h(x, y, i) \\
= & \quad \{\![\text{ definition of } h \ ]\!\} \\
 & K^{snd(num(y,i))-1}(x) \\
= & \quad \{\![\text{ definition of } num \ ]\!\} \\
 & K^{snd(\mathsf{if}\ B(y)\ \mathsf{then}\ num(K(y),i+1)\ \mathsf{else}\ (y,i))-1}(x) \\
= & \quad \{\![\text{ propagation of } K, \ snd \text{ and } -1 \ ]\!\} \\
 & \mathsf{if}\ B(y)\ \mathsf{then}\ K^{snd(num(K(y),i+1))-1}(x) \\
 & \qquad \mathsf{else}\ K^{snd(y,i)-1}(x)
\end{array}
$$

$$
\begin{array}{ll}
= & \{\![\text{ definition of } h \ ]\!\} \\
& \mathsf{if}\ B(y)\ \mathsf{then}\ h(x, K(y), i+1) \\
& \qquad \mathsf{else}\ K^{i-1}(x)
\end{array}
$$

By computational induction we now show

$$num''(K^i(x)) = h(x, K^{i+1}(x), i+1) \qquad (*)$$

$$
\begin{array}{ll}
& num''(K^i(x)) \\
= & \{\![\text{ definition of } num'' \ ]\!\} \\
& \mathsf{if}\ B(K(K^i(x)))\ \mathsf{then}\ num''(K(K^i(x))) \\
& \qquad \mathsf{else}\ K^i(x) \\
= & \{\![\text{ induction hypothesis }]\!\} \\
& \mathsf{if}\ B(K^{i+1}(x))\ \mathsf{then}\ h(x, K(K^{i+1}(x)), (i+1)+1) \\
& \qquad \mathsf{else}\ K^{(i+1)-1}(x) \\
= & \{\![\text{ recursive definition of } h \ ]\!\} \\
& h(x, K^{i+1}(x), i+1)
\end{array}
$$

Using the special case $h(x, K(x), 1) = num''(x)$ of this equivalence, we can show the claim:

$$
\begin{array}{ll}
& K^{n0-1}(x) \\
= & \{\![\text{ definition of } n0 \ ]\!\} \\
& K^{snd(num(x,0))-1}(x) \\
= & \{\![\text{ definition of } h \ ]\!\} \\
& h(x, x, 0) \\
= & \{\![\text{ recursive definition of } h \ ]\!\} \\
& \mathsf{if}\ B(x)\ \mathsf{then}\ h(x, K(x), 1) \\
& \qquad \mathsf{else}\ \Delta_x \\
= & \{\![\text{ by } (*) \ ]\!\} \\
& \mathsf{if}\ B(x)\ \mathsf{then}\ num''(x) \\
& \qquad \mathsf{else}\ \Delta_x \\
= & \{\![\text{ definition of } num' \ ]\!\} \\
& num'(x)
\end{array}
$$