SYSTEMATIC DERIVATION OF POINTER ALGORITHMS

Bernhard Möller

Institut für Informatik der Technischen Universität München
Arcisstr. 21, D-8000 München 2, Germany

Abstract We show that the well-known unfold/fold transformation strategy also is fruitful for the (formal) derivation of correct pointer algorithms. The key that allows this extension is the algebra of partial maps which allows convenient description and manipulation of pointer structures at the functional level.

1 Introduction

It is well-known that algorithms involving pointers are both difficult to write and to verify. The reason is that, due to the implict connections through paths within a pointer structure, the side effects of a pointer assignment are usually much harder to survey than those of an ordinary assignment. With this paper we want to show that these difficulties can be greatly reduced by making the store, which is an implicit global parameter in procedural languages, into an explicit parameter and by passing to an applicative treatment using a suitable algebra of operations on the store.

The storage state of a von Neumann machine can be viewed as a total mapping from addresses to certain values. A part of such a state that forms a logical unit may then be represented by a partial submapping of that mapping. This gives the possibility of describing the state in a modularized way as the union of the submappings for its logical subunits. In the case of pointer structures this means that the usual “spaghetti” structure of the complete state can be (at least partly) disentangled. Therefore we use the algebra of partial maps as our tool for specifying and developing pointer algorithms in a formal and yet convenient way.

We restrict ourselves here to the case of singly linked lists. However, the approach is not limited to such simple structures: In [3] we have derived an efficient and intricate garbage collection algorithm for a storage structure that allows the representation of arbitrary graphs.

Notationally, we closely follow the (ALGOL variant of) the language CIP-L (cf. [1,2]). In particular, we denote semantic equivalence of expressions by $\equiv$: We have $E_1 \equiv E_2$ iff both E_1 and E_2 are undefined (non-termination or abortion) or both are defined and have the same value. Equivalences are also denoted in the form of transformation rules, viz. as

$$\begin{array}{c} E_1 \\ \text{———}\updownarrow\text{———}[C \\ E_2 \end{array}$$

where C is a (possibly empty) list of applicability conditions, i.e., of conditions sufficient for the validity of the equivalence.

As an important aid in specifying and developing recursive routines we use assertions or restrictions about their parameters, formulated as Boolean expressions of the language. Let R be a Boolean expression possibly involving the identifier x. Then the declaration

$$\mathsf{funct}\ f \equiv (\mathsf{m}\ x : R)\,\mathsf{n} : E$$

of function f with parameter x restricted by R and with body E is by definition equivalent to

$$\mathsf{funct}\ f \equiv (\mathsf{m}\ x)\,\mathsf{n} : \mathsf{if}\ R\ \mathsf{then}\ E\ \mathsf{else\ error\ fi}\ .$$

This means that f is undefined for all arguments x that violate the restriction R; i.e., R acts as a precondition for f. If f is recursive, R has to hold also for the parameters of the recursive calls to ensure definedness; hence in this case R corresponds to invariants as known from imperative programming. Analogous constructions apply to statements and procedures.

2 The Algebra of Partial Maps

The use of algebraic operations on maps for describing the effect of a program dates back at least to [12]. The most useful operation in our setting, viz. map union, however, seems to have been neglected until recently [3, 11].

A **(partial) map** m from a set M to a set N is a subset of $M \times N$ such that $(x, y) \in m \wedge (x, z) \in m \Rightarrow y \equiv z$. Some of our notation derives from this set view of maps. E.g., by $\emptyset$ we denote the empty partial map from M to N. For finite maps we assume a boolean-valued equality test $=$. This is to be distinguished from the semantic equivalence $\equiv$ of expressions: we have

$$m = n \equiv \mathsf{true} \;\Leftrightarrow\; m \equiv n\ .$$

Let $m : P \longrightarrow Q$ be a partial map. We write $\downarrow m, \uparrow m$ for **domain** and **range** of m, resp. Moreover, we define

$$set(m) \overset{\text{def}}{\equiv} \downarrow m\ \cup \uparrow m\ .$$

For $s \subseteq P$, $[s \mapsto y]$ is the constant map $\{(x, y) \mid x \in s\}$. In using this notation we omit singleton set braces, i.e., we write $[x \mapsto y]$ instead of $[\{x\} \mapsto y]$. Note that $[x \mapsto y] \equiv \{(x, y)\}$. To cope with partialities in an algebraically convenient way, we define, for maps m, n and elements $x, y \in P$,

$$[m(x) \mapsto n(y)] \overset{\text{def}}{\equiv} \emptyset$$

if $x \notin \downarrow m$ or $y \notin \downarrow n$.

The **restriction** of a map $m : M \longrightarrow N$ to a set $s \subseteq M$ is

$$m|s \overset{\text{def}}{\equiv} m \cap (s \times N)\ .$$

Moreover,

$$m \ominus s \overset{\text{def}}{\equiv} m|\overline{s}.$$

Here again we omit singleton set braces, i.e., we write $m \ominus x$ instead of $m \ominus \{x\}$. Note that both $m|s \subseteq m$ and $m \ominus s \subseteq m$. The following decomposition property is the key to recursions over maps:

$$m \equiv m|s \cup m \ominus s\ .$$

Two maps $m, n : M \longrightarrow N$ are **compatible** if $m|(\downarrow m \cap \downarrow n) \equiv n|(\downarrow m \cap \downarrow n)$. This holds in particular if $\downarrow m \cap \downarrow n \equiv \emptyset$. For compatible m, n their union $m \cup n$ is again a map. This generalizes to families $(m_i)_{i \in I}$ of maps (I may even be infinite) if the maps m_i are pairwise compatible; we then write $\bigcup_{i \in I} m_i$ for the union map. If $I \equiv \emptyset$, we set $\bigcup_{i \in I} m_i \equiv \emptyset$ as well. It should be clear that $\emptyset$, $[.\mapsto.]$, and $\bigcup$ form a complete set of constructors for the set of partial maps, since we have

$$m \equiv \bigcup_{x \in \downarrow m} [x \mapsto m(x)]\ .$$

The operation of map union is the key tool in obtaining a modular description of pointer structures, since it allows viewing a (total) storage state as the union of those of its (partial) substates that form logical units. This aspect of modularization is reflected by a large number of distributive laws that allow propagation of operations to substates of a state. For the operations introduced so far we have:

$$\begin{array}{rclrcl} \downarrow(m\cup n) & \equiv & \downarrow m \cup \downarrow n & \qquad \uparrow(m\cup n) & \equiv & \uparrow m \cup \uparrow n \\ (m\cup n)|s & \equiv & m|s \cup n|s & \qquad (m\cup n)\ominus s & \equiv & m\ominus s \cup m\ominus t\ . \end{array}$$

Another important operation is **map overwriting** (see e.g. [6]): Given maps $m, n : M \longrightarrow N$ we define

$$m \Leftarrow n \overset{\text{def}}{\equiv} (m\ominus\downarrow n) \ \cup\ n\ .$$

Hence,

$$(m\Leftarrow n)(x) \equiv \textsf{if } x \in\ \downarrow n \textsf{ then } n(x) \textsf{ else } m(x) \textsf{ fi}.$$

In other words, $m\Leftarrow n$ results from m by changing the values according to the prescription of n (if any). For example, $m\Leftarrow[x\mapsto y]$ sets the value of x to y. This operation will be our main tool for describing selective updating. Its most important properties for our purposes are the following ones:

1. Monoid properties:
   $\emptyset\Leftarrow m \equiv m\Leftarrow\emptyset \equiv m$
   $(l\Leftarrow m)\Leftarrow n \equiv l\Leftarrow(m\Leftarrow n)$

2. Overwriting and union:
   $m\Leftarrow n \equiv n\Leftarrow m$ iff m and n are compatible.
   In this case, $m\Leftarrow n \equiv m\cup n$.

3. Domain properties:
   $\downarrow(m\Leftarrow n) \equiv \downarrow m \cup \downarrow n$
   $m\Leftarrow n \equiv n \Leftrightarrow \downarrow m \subseteq \downarrow n$

4. Overwriting and submaps:
   $m\Leftarrow n \equiv m \Leftrightarrow\ n \subseteq m$

5. Sequentialization:
   $l\Leftarrow(m\cup n) \equiv (l\Leftarrow m)\Leftarrow n$
   provided m and n are compatible.

6. Annihilation:
   $m \subseteq l \ \Rightarrow\ l \Leftarrow (m\cup n) \equiv l \Leftarrow n$
   provided m and n are compatible. This is an immediate consequence of the sequentialization and submap properties.

7. Distributivity:
   $(l\cup m)\Leftarrow n \equiv (l\Leftarrow n)\cup(m\Leftarrow n)$
   provided l and m are compatible.

8. Localization:
   $\downarrow l \cap \downarrow n \equiv \emptyset \ \Rightarrow\ (l\cup m)\Leftarrow n \equiv l\cup(m\Leftarrow n)$
   provided l and m are compatible. This property allows localizing side effects to that part of a store they really affect.

The map operations introduced enjoy a vast number of further useful algebraic laws. Some of them can be found in [3].

3 Chains

As an example of how to describe pointer structures within the algebra of maps we now study singly linked lists. We abstract from the concrete contents of the records in such a list and consider only their interrelationship through the pointers, since this is the only source of problems in pointer algorithms. Then a **state** simply is a finite partial map $m : \mathsf{cell} \longrightarrow \mathsf{cell}$ where cell is the set of storage cells; the set of states is denoted by state. A single cell x together with its contents y is modeled by the map $[x \mapsto y]$.

By a **chain** we mean a (finite) cycle-free singly linked list. Such a chain contains a number of cells in a certain order prescribed by the links in the list. This induces a sequence structure on these cells: The first element in the sequence is the head cell, followed by the others in the order of traversal. Since there is no cycle, the sequence is repetition-free.

In chains one frequently uses a special chain terminator common to all chains considered (e.g., nil in Pascal). Let therefore $\Box \in \mathsf{cell}$ be a distinguished element, called the **anchor**. The elements of $\mathsf{cell} \backslash \{\Box\}$ are called **proper cells**. In the sequel we require $\Box \notin \downarrow m$ for all states m considered. This means that $\Box$ may never be assigned a "contents"and hence never be "dereferenced"; it will always be an "empty cell", whence our notation. Moereover, this implies that there can be no $\Box$ cell properly within a chain; if present, $\Box$ terminates the respective list. A chain is called **anchored** if it ends with $\Box$, i.e., if its last proper cell contains $\Box$.

By the above considerations, anchored chains are in exact correspondence with non-empty repetition-free sequences of proper cells. Given such a sequence, we can construct an anchored chain using

$$\mathsf{funct}\ \mathit{chain} \equiv (\mathsf{cellsequ}\ s : \mathit{ischainable}(s))\,\mathsf{state} : \bigcup_{i=1}^{|s|} [s[i] \mapsto s[i+1]] \ .$$

By $|s|$ we denote the length of s and by $s[i]$ the i-th element of s; if $i > |s|$ or $i = 0$, we set $s[i] \overset{\text{def}}{\equiv} \Box$. The predicate *ischainable* is given by

```
funct $\mathit{ischainable} \equiv (\mathsf{cellsequ}\ s)\,\mathsf{bool}$ :
    if $s = \langle\rangle$ then false
                  else $\mathit{first}(s) \neq \Box \wedge (\mathit{rest}(s) = \langle\rangle$ cor
                       $(\mathit{first}(s) \notin \mathit{rest}(s) \wedge \mathit{ischainable}(\mathit{rest}(s))))$ fi ,
```

where cor is the sequential or conditional disjunction evaluated from left to right. Conversely, given a cell x and a state m we can retrieve the sequence of cells in the sublist starting from x (if any) using

```
funct $\mathit{sequ} \equiv (\mathsf{cell}\ x, \mathsf{state}\ m)\,\mathsf{cellsequ}$ :
    if $x \notin \downarrow m$ then $\langle\rangle$ else $\langle x \rangle + \mathit{sequ}(m(x), m)$ fi .
```

Here, $\langle\rangle$ denotes the empty sequence, $\langle x \rangle$ is the singleton sequence consisting just of x, and $+$ denotes concatenation. Note that this function will not terminate if the sublist within m starting from x contains a cycle. In our applications this will not occur. For more general use, however, one should base this on a non-strict functional language in which the algorithm then would return a periodically infinite sequence of cells. Then a cell y can be reached from x following the links of m (zero or more times) iff $y \in \mathit{sequ}(x, m)$, where

```
funct $. \in . \equiv (\mathsf{cell}\ y, \mathsf{cellsequ}\ s)\,\mathsf{bool}$ :
    if $s = \langle\rangle$ then false else $y = \mathit{first}(s)$ cor $y \in \mathit{rest}(s)$ fi .
```

To characterize the case where the sublist starting from x in m is an anchored chain, we use

```
funct $\mathit{isanchored} \equiv (\mathsf{cell}\ x, \mathsf{state}\ m)\,\mathsf{bool}$ :
    if $x \notin \downarrow m$ then false else $m(x) = \Box$ cor $\mathit{isanchored}(m(x), m)$ fi .
```

This function again doesn't terminate if there is a cycle, and it yields false if it runs into a "dangling reference", i.e., a cell different from $\Box$ not having any contents.

If $\mathit{isanchored}(x, s) \equiv \mathsf{true}$, the sublist starting from x actually is an anchored chain. Its last proper cell, i.e., the one containing the nil pointer, is obtained by

$$\mathsf{funct}\ \mathit{lastcell} \equiv (\mathsf{cell}\ x, \mathsf{state}\ m : \mathit{isanchored}(x, m))\ \mathsf{cell} : \mathit{last}(\mathit{sequ}(x, m))$$

We want to derive a direct recursion for this function:

$$\begin{array}{ll}
 & \mathit{lastcell}(x, m) \\
\equiv & \mathit{last}(\mathit{sequ}(x, m)) \\
\equiv & (\text{unfold } \mathit{sequ}) \\
 & \mathit{last}(\mathsf{if}\ x \notin \downarrow m\ \mathsf{then}\ \langle\rangle \\
 & \qquad\qquad\quad \mathsf{else}\ \ \langle x \rangle + \mathit{sequ}(m(x), m)\ \ \mathsf{fi}) \\
\equiv & (\text{since } x \notin \downarrow m \equiv \mathsf{false} \text{ by } \mathit{isanchored}(x, m)) \\
 & \mathit{last}(\langle x \rangle + \mathit{sequ}(m(x), m)) \\
\equiv & (\text{case introduction}) \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ \mathit{last}(\langle x \rangle + \mathit{sequ}(m(x), m)) \\
 & \qquad\qquad\quad \mathsf{else}\ \ \mathit{last}(\langle x \rangle + \mathit{sequ}(m(x), m))\ \ \mathsf{fi} \\
\equiv & (\text{evaluation of } \mathit{sequ}(m(x), m) \text{ in } \mathsf{then}\text{-branch}) \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ \mathit{last}(\langle x \rangle + \langle\rangle) \\
 & \qquad\qquad\quad \mathsf{else}\ \ \mathit{last}(\langle x \rangle + \mathit{sequ}(m(x), m))\ \ \mathsf{fi} \\
\equiv & (\text{simplification, using } \mathit{sequ}(m(x), m) \not\equiv \langle\rangle \text{ in } \mathsf{else}\text{-branch}) \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ x \\
 & \qquad\qquad\quad \mathsf{else}\ \ \mathit{last}(\mathit{sequ}(m(x), m))\ \ \mathsf{fi} \\
\equiv & (\text{fold } \mathit{lastcell}) \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ x \\
 & \qquad\qquad\quad \mathsf{else}\ \ \mathit{lastcell}(m(x), m)\ \ \mathsf{fi}\ .
\end{array}$$

A similar development shows that

Lemma 3.1
$\mathit{sequ}(x, m) \equiv \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ \langle x \rangle\ \mathsf{else}\ \langle x \rangle + \mathit{sequ}(m(x), m)\ \mathsf{fi}$
provided $\mathit{isanchored}(x, m) \equiv \mathsf{true}$.

Moreover, one easily proves by induction on the length of s that

Lemma 3.2
$\mathit{sequ}(s[1], \mathit{chain}(s)) \equiv s$ provided $\mathit{ischainable}(s) \equiv \mathsf{true}$.

Conversely, we have

Lemma 3.3
$\mathit{chain}(\mathit{sequ}(x, m)) \subseteq m$ provided $\mathit{isanchored}(x, m)$.

4 Concatenation of Chains "in Situ"

4.1 Specification and First Explicit Solution

We now want to specify and develop an algorithm for concatenating two non-overlapping anchored chains "in situ". (We do not consider the trivial case of empty chains which would only lead to tedious case distinctions.) First we give the precondition for our desired function:

$$\begin{array}{l}\textsf{funct}\ concpc \equiv (\textsf{cell}\ x, \textsf{cell}\ y, \textsf{state}\ m)\ \textsf{bool} : \\ \quad (isanchored(x,m) \wedge isanchored(y,m))\ \textsf{cand}\ set(sequ(x,m) \cap setsequ(y,m)) = \emptyset\ ,\end{array}$$

where cand is the sequential or conditional conjunction evaluated from left to right.

So we consider a state m in which the sublists starting from x and y are anchored chains the sets of proper cells of which are disjoint. We want to form a new state in which the concatenation of these two sublists is overwritten onto *the same* set of proper cells; moreover, the order of traversal within the sublists should be preserved, and all cells from the sublist of x should precede all cells in the sublist of y. This can be specified by

$$\begin{array}{l}\textsf{funct}\ conc \equiv (\textsf{cell}\ x, \textsf{cell}\ y, \textsf{state}\ m : concpc(x,y,m))\ \textsf{state} : \\ \quad m \Lleftarrow chain(sequ(x,m) + sequ(y,m))\ .\end{array}$$

So the proper cells of the subchains are collected in the right order, the resulting sequence is chained and this chain is overwritten onto m *re-using the same cells*. Hence, no copying is involved and we really are specifying concatenation "in situ".

We now want to develop an algorithm from this specification. First, we concentrate on the subexpression $chain(sequ(x,m)+sequ(y,m))$. For abbreviation, we set $s \overset{\text{def}}{\equiv} sequ(x,m)$ and $t \overset{\text{def}}{\equiv} sequ(y,m)$. Now we calculate

$$\begin{array}{ll} & chain(s+t) \\ \equiv & \bigcup\limits_{i=1}^{|s+t|} [(s+t)[i] \mapsto (s+t)[i+1]] \\ \equiv & \bigcup\limits_{i=1}^{|s|-1} [s[i] \mapsto s[i+1]] \cup [s[|s|] \mapsto t[1]] \cup \bigcup\limits_{i=|s|+1}^{|s|+|t|} [t[i-|s|] \mapsto t[i+1-|s|]] \\ \equiv & chain(s) \ominus s[|s|] \cup [s[|s|] \mapsto t[1]] \cup \bigcup\limits_{j=1}^{|t|} [t[j] \mapsto t[j+1]] \\ \equiv & (chain(s) \Lleftarrow [last(s) \mapsto first(t)]) \cup chain(t)\ . \end{array}$$

From this we obtain

$$\begin{array}{ll} & m \Lleftarrow chain(sequ(x,m) + sequ(y,m)) \\ \equiv & m \Lleftarrow ((chain(s) \Lleftarrow [last(s) \mapsto first(t)]) \cup chain(t)) \\ \equiv & (\text{commutativity of } \cup) \\ & m \Lleftarrow (chain(t) \cup (chain(s) \Lleftarrow [last(s) \mapsto first(t)])) \\ \equiv & (\text{sequentialization, associativity of } \Lleftarrow) \\ & m \Lleftarrow (chain(t) \Lleftarrow chain(s) \Lleftarrow [last(s) \mapsto first(t)]) \\ \equiv & m \Lleftarrow (chain(sequ(y,m)) \Lleftarrow chain(sequ(x,m)) \Lleftarrow [last(s) \mapsto first(t)]) \\ \equiv & (\text{Lemma 3.3, annihilation}) \\ & m \Lleftarrow [last(s) \mapsto first(t)] \\ \equiv & m \Lleftarrow [lastcell(x,m) \mapsto y]\ . \end{array}$$

Now we introduce an auxiliary function for computing this expression:

$$owlast(m,x,y) \overset{\text{def}}{\equiv} m \Lleftarrow [lastcell(x,m)) \mapsto y]\ .$$

We have

$$conc(x,y,m) \equiv owlast(m,x,y)\ .$$

Now we derive a recursion for $owlast$.

$$\begin{array}{ll} & owlast(m,x,y) \\ \equiv & (\text{unfold } owlast) \end{array}$$

$$
\begin{aligned}
& m \mathbin{\triangleleft\!\!+} [\mathit{lastcell}(x,m) \mapsto y] \\
\equiv\ & \text{(by the recursion for } \mathit{lastcell}\text{)} \\
& \textsf{if } m(x) = \Box \textsf{ then } m \mathbin{\triangleleft\!\!+} [x \mapsto y] \textsf{ else } m \mathbin{\triangleleft\!\!+} [\mathit{lastcell}(m(x),m) \mapsto y] \textsf{ fi} \\
\equiv\ & \text{(fold } \mathit{owlast}\text{)} \\
& \textsf{if } m(x) = \Box \textsf{ then } m \mathbin{\triangleleft\!\!+} [x \mapsto y] \textsf{ else } \mathit{owlast}(m, m(x), y) \textsf{ fi} .
\end{aligned}
$$

Termination of this recursion follows from $\mathit{isanchored}(x,m)$. It is quite reassuring that the fundamental unfold/fold technique for deriving recursions also applies to pointer algorithms in this setting.

4.2 Introducing Selective Updating

Since we have even obtained a tail-recursive version, we are already very close to an imperative program. To get there, we introduce a procedure specified by

```
proc $powconc \equiv$ (var state $m$, cell $x, y$ : $concpc(x,y,m)$)
    $m := owlast(m,x,y)$
```

Note that this clearly specifies m as a transient parameter, whereas x and y are passed by value. Therefore the imperative version of *powconc* needs local variables for x and y, whereas it may operate on m directly. This is described by the following schematic rule for passing from a procedure that calls a tail-recursive function to a procedure with a loop in its body:

```
proc $p \equiv$ (var m $a$, n $b$ : $P(a,b)$) : $a := f(a,b)$

where
    funct $f \equiv$ (m $a$, n $b$) m :
        if $C(a,b)$ then $T(a,b)$ else $f(K(a,b), L(a,b))$ fi
---------------------------------- $\updownarrow$ ---------------------------------[ NEW$[\![B]\!]$
proc $p \equiv$ (var m $a$, n $B$ : $P(a,B)$) :
    $\lceil$ var n $b := B$ ;
      while $\neg\, C(a,b)$ do $(a,b) := (K(a,b), L(a,b))$ od ;
      $a := T(a,b)$                                        $\rfloor$ .
```

Note that a, b, and B may stand for tuples of variables. The condition NEW$[\![B]\!]$ states that B has to be a (tuple of) fresh identifier(s). Applying this rule we obtain

```
proc $powconc \equiv$ (var state $m$, cell $X, Y$ : $concpc(X,Y,m)$) :
    $\lceil$ (var cell $x, y$) := $(X, Y)$ ;
      while $m(x) \neq \Box$ do $(m, x, y) := (m, m(x), y)$ od ;
      $m := m \mathbin{\triangleleft\!\!+} [x \mapsto y]$                              $\rfloor$
```

Our final version results from eliminating useless assignments of the form $z := z$ as well as the variable y which never is changed:

```
proc $powconc \equiv$ (var state $m$, cell $X, Y$ : $concpc(X,Y,m)$) :
    $\lceil$ var cell $x := X$ ;
      while $m(x) \neq \Box$ do $x := m(x)$ od ;
      $m := m \mathbin{\triangleleft\!\!+} [x \mapsto Y]$                    $\rfloor$ .
```

If we write the assignment

$$m := m \mathbin{\triangleleft\!\!+} [x \mapsto Y]$$

in a Pascal-like way as

$$x\uparrow := Y \ ,$$

(where m now is an implicit parameter), we see that we actually have derived a version with selective updating.

In the derivation we have not made use of any assumptions about absence of sharing. Indeed, if in m there are pointers from other data structures to (parts of) the lists headed by x and y, there will be indirect side effects on these pointers. However, since by the specification we know the value of the complete store after execution of our procedure, we can *calculate* these effects using our algebraic laws. Also, one can easily write stronger preconditions that exclude sharing if this is desired.

5 Chain Reversal

5.1 Specification and First Explicit Solution

Next we want to derive a procedure for reversing a non-empty chain "in situ". Again we first specify a purely applicative version. The reverse of a chain should contain exactly the same proper cells as the original chain, however, in reverse order of traversal. We can express this as follows:

$$\begin{array}{l} \mathsf{funct}\ \mathit{reverse} \equiv (\mathsf{cell}\ x, \mathsf{state}\ m : \mathit{isanchored}(x,m))\ \mathsf{state} : \\ \quad m \mathrel{\triangleleft\!\!-} \mathit{chain}(\mathit{rev}(\mathit{sequ}(x,m))) \end{array}$$

where *rev* is the reversal function on sequences.

Let us now derive an explicit form of $\mathit{reverse}(x,m)$. Again, we first concentrate on the subexpression $\mathit{rev}(\mathit{sequ}(x,m))$. Let $s \overset{\text{def}}{\equiv} \mathit{sequ}(x,m)$. We calculate:

$$\begin{array}{ll}
& \mathit{chain}(\mathit{rev}(s)) \\
\equiv & \bigcup\limits_{i=1}^{|\mathit{rev}(s)|} [\mathit{rev}(s)[i] \mapsto \mathit{rev}(s)[i+1]] \\
\equiv & \bigcup\limits_{i=1}^{|s|} [s[|s|+1-i] \mapsto s[|s|+1-(i+1)]] \\
\equiv & (\text{index transformation } j \equiv |s|-i) \\
& \bigcup\limits_{j=0}^{|s|-1} [s[j+1] \mapsto s[j]] \\
\equiv & [s[1] \mapsto s[0]] \cup \bigcup\limits_{j=1}^{|s|-1} [s[j+1] \mapsto s[j]] \\
\equiv & [x \mapsto \Box] \cup \bigcup\limits_{j=1}^{|s|-1} [s[j] \mapsto s[j+1]]^{-1} \\
\equiv & [x \mapsto \Box] \cup (\bigcup\limits_{j=1}^{|s|-1} [s[j] \mapsto s[j+1]])^{-1} \\
\equiv & [x \mapsto \Box] \cup (\mathit{chain}(s) \ominus s[|s|])^{-1} \\
\equiv & [x \mapsto \Box] \cup \mathit{chain}(s)^{-1} \ominus \Box \ .
\end{array}$$

Here we temporarily make use of the map $\mathit{chain}(s)^{-1}$ which is not a state; however, $\mathit{chain}(s)^{-1} \ominus \Box$ again is. Now we introduce an auxiliary function

$$\mathit{owrev}(m,x,y) \overset{\text{def}}{\equiv} m \mathrel{\triangleleft\!\!-} [x \mapsto y] \mathrel{\triangleleft\!\!-} \mathit{chain}(\mathit{sequ}(x,m))^{-1} \ominus \Box$$

with the embedding

$$\mathit{reverse}(x,m) \equiv \mathit{owrev}(m,x,\Box) \ .$$

For abbreviation we introduce $n \overset{\text{def}}{\equiv} m \ll\!\!\!- [x{\mapsto}y]$. Now we can develop a recursion equation:

$$\begin{array}{ll}
 & owrev(m,x,y) \\
\equiv & \text{(unfold } owrev) \\
 & m \ll\!\!\!- [x{\mapsto}y] \ll\!\!\!- chain(sequ(x,m))^{-1}\ominus\Box \\
\equiv & n \ll\!\!\!- chain(sequ(x,m))^{-1}\ominus\Box \\
\equiv & \text{(by Lemma 3.1 and definition of } chain) \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ n \ll\!\!\!- [x{\mapsto}\Box]^{-1}\ominus\Box \\
 & \qquad\qquad\qquad \mathsf{else}\ \ n \ll\!\!\!- ([x{\mapsto}m(x)] \cup chain(sequ(m(x),m)))^{-1}\ominus\Box\ \ \mathsf{fi} \\
\equiv & \text{(distributivity, inverse)} \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ n \ll\!\!\!- [\Box{\mapsto}x]\ominus\Box \\
 & \qquad\qquad\qquad \mathsf{else}\ \ n \ll\!\!\!- ([m(x){\mapsto}x]\ominus\Box \cup chain(sequ(m(x),m))^{-1}\ominus\Box)\ \ \mathsf{fi} \\
\equiv & (m(x){\neq}\Box \text{ in } \mathsf{else}\text{-case}) \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ n \ll\!\!\!- \emptyset \\
 & \qquad\qquad\qquad \mathsf{else}\ \ n \ll\!\!\!- ([m(x){\mapsto}x] \cup chain(sequ(m(x),m))^{-1}\ominus\Box)\ \ \mathsf{fi} \\
\equiv & \text{(neutrality, sequentialization)} \\
 & \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ n \\
 & \qquad\qquad\qquad \mathsf{else}\ \ n \ll\!\!\!- [m(x){\mapsto}x] \ll\!\!\!- chain(sequ(m(x),m))^{-1}\ominus\Box\ \ \mathsf{fi}\ .
\end{array}$$

Now we are almost in the position to fold with the definition of *owrev*. However, this would need the expression $sequ(m(x),n)$ instead of $sequ(m(x),m)$ in the else-branch. Fortunately one can show

Lemma 5.1

(1) If $z \in {\downarrow}l \equiv \mathsf{true}$ and $u \in sequ(z,l) \not\equiv \mathsf{true}$ then $u \not\equiv z$ and $u \in sequ(l(z),l) \not\equiv \mathsf{true}$.

(2) $sequ(z,l) \equiv sequ(z, l \ll\!\!\!- [u{\mapsto}v])$ provided $u \in sequ(z,l) \not\equiv \mathsf{true}$.

Proof: (1) By assumption,

$$\mathsf{true} \not\equiv u \in sequ(z,l) \equiv u \in (\langle z\rangle + sequ(l(z),z)) \equiv u \in \langle z\rangle\ \mathsf{cor}\ u \in sequ(l(z),z)\ .$$

By definition of cor we must have $u \in \langle z\rangle \not\equiv \mathsf{true}$, i.e. $u \in \langle z\rangle \equiv \mathsf{false}$. Now the claim is immediate.

(2) is proved by computational induction (see e.g. [9]) with the predicate

$$\begin{array}{lll}
P[f] & \overset{\text{def}}{\Leftrightarrow} & \forall\, l : \forall\, z : \forall\, u : \forall\, v : \\
 & & u \in sequ(z,l) \not\equiv \mathsf{true} \Rightarrow f(z,l) \equiv f(z, l \ll\!\!\!- [u{\mapsto}v])\ .
\end{array}$$

The induction base $P[\Omega]$ is trivial. For the induction step assume $P[f]$ and $u \in sequ(z,l) \not\equiv \mathsf{true}$. For the functional τ associated with the body of *sequ* we get

$$\begin{array}{ll}
 & \tau[f](z,l) \\
\equiv & \mathsf{if}\ z{\notin}{\downarrow}l\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ \langle z\rangle + f(l(z),l)\ \mathsf{fi} \\
\equiv & \text{(by (1) and the induction hypothesis } P[f]) \\
 & \mathsf{if}\ z{\notin}{\downarrow}l\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ \langle z\rangle + f(l(z), l \ll\!\!\!- [u{\mapsto}v])\ \mathsf{fi} \\
\equiv & \text{(by (1))} \\
 & \mathsf{if}\ z{\notin}{\downarrow}(l \ll\!\!\!- [u{\mapsto}v])\ \mathsf{then}\ \langle\rangle \\
 & \qquad\qquad\qquad\qquad \mathsf{else}\ \ \langle z\rangle + f((l \ll\!\!\!- [u{\mapsto}v])(z), l \ll\!\!\!- [u{\mapsto}v])\ \ \mathsf{fi} \\
\equiv & \tau[f](z, l \ll\!\!\!- [u{\mapsto}v])\ .
\end{array}$$

■

This now allows folding with the definition of *owrev* in the above expression yielding the recursion

$$owrev(m, x, y) \equiv \mathsf{if}\ m(x) = \Box\ \mathsf{then}\ n\ \mathsf{else}\ owrev(n, m(x), x)\ \mathsf{fi}\ .$$

Again we have arrived at an (obviously terminating) tail recursion.

5.2 A Version With Selective Updating

Specifying a procedure

```
proc $powrev \equiv$ (var state $m$, cell $x : isanchored(x, m)$) :
    $m := reverse(x, m)$)
```

we obtain, as in the previous section, the final version

```
proc $powrev \equiv$ (var state $m$, cell $X : isanchored(X, m)$) :
    ⌈ (var cell $x, y$) := $(X, \Box)$ ;
      while $m(x) \neq \Box$
        do $(m, x, y) := (m \nleftarrow [x \mapsto y], m(x), x)$ od;
      $m := m \nleftarrow [x \mapsto y]$                                ⌋
```

Note that sequentialization of the collective assignment would require an auxiliary variable.

This program describes a well-known algorithm for reversing a list "in situ". Whereas verification purely at the procedural level is by no means easy (see e.g. [5, 8]), in particular if all the details were to be filled in, we have derived and thereby verified the program by a fairly short and simple formal calculation using standard transformation techniques.

6 Conclusion

We have shown with two examples how to derive algorithms involving pointers and selective updating from formal specifications using standard transformation techniques. The key to the method consists in considering the store as an explicit parameter, since then one has complete information about sharing and therefore complete control about side effects. We deem this approach much clearer (and much more convenient) than the idea of hiding the store and coming up with special logics (see e.g. [10, 7, 5]) that capture the side-effects indirectly, as needs to be done in the field of verification of procedural programs.

Staying at the applicative level almost to the very end of the derivations has allowed us to take full advantage of the powerful algebra of partial maps. The operations of that algebra are even that expressive that we did not need to explain anything with the help of diagrams. This may seem due to the simplicity of the algorithms. However, also when developing the intricate garbage collection algorithm described in [3] we quite soon stopped drawing diagrams because the algebraic formulation was clearer and much more modular. Another advantage of the applicative treatment is that if additional predicates or operations on maps are needed, they are much more easily added at the applicative than at the procedural level. Finally, if pointer algorithms are developed in a systematic way at the applicative language level, there is no need for introducing additional imperative language concepts such as the highly imperspicuous pointer rotation [13].

We are convinced that our approach can be extended into a convenient method for constructing systems software with guaranteed correctness.

Acknowledgements

The idea of an algebraic treatment of pointers was stimulated by discussions within IFIP WG 2.1, notably by the algebraic way in which R. Bird and L. Meertens develop tree and list algorithms. I gratefully acknowledge many helpful conversations with my present and former colleagues from the project CIP, notably with F.L. Bauer, U. Berger, H. Partsch, P. Pepper, W. Meixner, and, particularly, H. Ehler.

References

1. F.L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T.A. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing, H. Wössner: The Munich project CIP. Volume I: The wide spectrum language CIP-L. Lecture Notes in Computer Science **183**. Berlin: Springer 1985

2. F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal program construction by transformations — Computer-aided, Intuition-guided Programming. Institut für Informatik der TU München, TUM-I8807, Juni 1988. Also in IEEE Transactions on Software Engineering **15**, 165–180 (1989)

3. U. Berger, W. Meixner, B. Möller: Calculating a garbage collector. In: M. Broy M. Wirsing (ed.): Methodik des Programmierens. Fakultät für Mathematik und Informatik der Universität Passau, MIP-8915, 1989, 1–52. Also in: M. Broy, M. Wirsing (eds.): Programming methodology — The CIP approach. To appear in Lecture Notes in Computer Science . Berlin: Springer

4. A. Bijlsma: Calculating with pointers. Science of Computer Programming **12**, 191–205 (1988)

5. R. Burstall: Some techniques for proving correctness of programs which alter data structures. In: B. Meltzer, D. Mitchie (eds.): Machine Intelligence **7**. Edinburgh University Press 1972, 23–50

6. C.B. Jones: Software development: A rigorous approach. Eglewood Cliffs: Prentice-Hall 1980

7. A. Kausche: Modale Logiken von geflechtartigen Datenstrukturen und ihre Kombination mit temporaler Programmlogik. Fakultät für Mathematik und Informatik der TU München, Dissertation, 1989

8. M. Levy: Verification of programs with data referencing. Proc. 3me Colloque sur la Programmation 1978, 413–426

9. Z. Manna: Mathematical theory of computation. New York: McGraw-Hill 1974

10. I. Mason: Verification of programs that destructively manipulate data. Science of Computer Programming **10**, 177–210 (1988)

11. P. Pepper, B. Möller: Programming with (finite) mappings. In: M. Broy (ed.): Informatik im Kreuzungspunkt von Numerischer Mathematik, Rechnerentwurf, Programmierung, Algebra und Logik. Festkolloquium für F.L. Bauer, Juni 1989. To appear in Lecture Notes in Computer Science . Berlin: Springer

12. J. Reynolds: Reasoning about arrays. Commun. ACM **22**, 290–299 (1979)

13. N. Suzuki: Analysis of pointer rotation. Conf. Record 7th POPL, 1980, 1–11. Revised version: Commun. ACM **25**, 330–335 (1982)