SOME APPLICATIONS OF POINTER ALGEBRA

Bernhard Möller

Institut für Mathematik der Universität Augsburg
Universitätsstr.2, W-8900 Augsburg, Germany

1 Introduction

It is well-known that algorithms involving pointers are both difficult to write and to verify. One 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. Second, a careless assignment may destroy the last link to a substructure which thus is lost forever. Now, not only is it easy to make such errors; it is also very hard to find them. 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.

In the first part, we restrict ourselves to the case of singly linked lists. However, the approach is not limited to such simple structures: In [Berger et al. 89] we have derived an efficient and intricate garbage collection algorithm for a storage structure that allows the representation of arbitrary graphs. The second part presents some essential steps of this development.

2 Preliminaries

Notationally and conceptually, we closely follow the (ALGOL variant of) the language CIP-L (cf. [Bauer et al. 85, 89]). CIP-L has a mathematical semantics that associates with each expression E the set $\mathcal{B}[\![E]\!]$ of possible values; $\mathcal{B}[\![E]\!]$ is called the **breadth** of E. The special value $\perp$ models the possibility of an erroneous or nonterminating computation. We set

$$\begin{aligned} &\mathrm{DEFINED}[\![E]\!] \overset{\text{def}}{\Leftrightarrow} \perp \notin \mathcal{B}[\![E]\!] \ , \\ &\mathrm{DETERMINATE}[\![E]\!] \overset{\text{def}}{\Leftrightarrow} |\mathcal{B}[\![E]\!]| = 1 \ , \end{aligned}$$

and call E **defined** resp. **determinate** if $\mathrm{DEFINED}[\![E]\!]$ resp. $\mathrm{DETERMINATE}[\![E]\!]$ holds. For convenience, the semantical values are also considered as expressions. The details of the semantic description can be found in [Bauer et al. 85].

Based on the breadth, two fundamental relations between expressions are defined: E_1 and E_2 are called **equivalent** if $\mathcal{B}[\![E_1]\!] = \mathcal{B}[\![E_2]\!]$; we denote this by $E_1 \equiv E_2$. This "strong" or meta-equality is not to be confused with "weak" (i.e. strict) equality tests $=$ in the language itself: The formula

$$\bot \equiv \bot$$

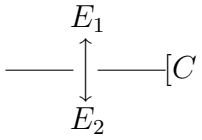
is valid. However,

$$\bot = \bot \not\equiv \mathsf{true} ;$$

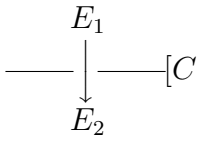
rather we have

$$\bot = \bot \equiv \bot .$$

Equivalences are also denoted in the form of transformation rules, viz. as



where C is a (possibly empty) list of applicability conditions, i.e., of conditions sufficient for the validity of the equivalence. More generally, E_2 is called a **descendant** of E_1 if $\mathcal{B}[\![E_2]\!] \subseteq \mathcal{B}[\![E_1]\!]$; we denote this by $E_1 \sqsupseteq_D E_2$ and, in the form of a transformation rule, as



where C again is a list of applicability conditions.

As an important aid in specifying and developing recursive routines we use assertions about the objects involved. They are formulated as Boolean expressions of the language. Given such an expression P, we use the notation

$$P \rhd E$$

as an abbreviation for the expression

$$\mathsf{if}\ P\ \mathsf{then}\ E\ \mathsf{else}\ \bot\ \mathsf{fi} .$$

A collection of useful algebraic properties of this construct can be found in [Möller 89]. Our principal use of it is within parameter restrictions for functions (cf. [Bauer, Wössner 82, Bauer et al. 85]): 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

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

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.

As our next piece of notation we introduce the conditional conjunction cand and the conditional disjunction cor defined by

$$\begin{array}{lcl} x \text{ cand } y & \stackrel{\text{def}}{\equiv} & \text{if } x \text{ then } y \text{ else false fi} \\ x \text{ cor } y & \stackrel{\text{def}}{\equiv} & \text{if } x \text{ then true else } y \text{ fi} . \end{array}$$

So they are asymmetric, sequentially evaluated variants of the usual boolean operations $\wedge$ and $\vee$. We mainly use them to shield partialities. E.g., the expression

$$x \neq 0 \text{ cand } 10/x = 2$$

is defined for all x, whereas

$$x \neq 0 \wedge 10/x = 2$$

is undefined for $x \equiv 0$, since then $10/x$ is undefined and $=$ and $\wedge$ are strict. Of course, in addition cand and cor are usually more efficient than $\wedge$ and $\vee$, since, as the above definition shows, they may save evaluation of the second operand.

3 The Algebra of Partial Maps

The use of algebraic operations on maps for describing the effect of a program dates back at least to [Reynolds 79]. The most useful operation in our setting, viz. map union, however, seems to have been neglected until recently [Berger et al. 89, Pepper, Möller 89].

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 strict boolean-valued equality test $=$.

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

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

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

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

Moreover,

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

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 intersection $m \cap n$ of two maps is again a map. Restriction interacts with intersection as follows:

$$\begin{array}{l} (m \cap n)|s \equiv m|s \cap n|s \\ (m \cap n) \ominus s \equiv m \ominus s \cap n \ominus s \\ m|(s \cap t) \equiv m|s \cap m|t \\ m \ominus (s \cap t) \equiv m \ominus s \cup m \ominus t . \end{array}$$

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 $\downarrow m \cap \downarrow n \equiv \emptyset$. We can give a technically simpler characterization of compatibility:

Lemma 3.1
m and n are compatible iff $m|\downarrow n \equiv n|\downarrow m$.

Proof: We observe that

$$\begin{array}{ll} & m|(\downarrow m \cap \downarrow n) \\ \equiv & m|\downarrow m \cap m|\downarrow n \\ \equiv & m \cap m|\downarrow n \\ \equiv & \quad \text{(since } m|\downarrow n \subseteq m\text{)} \\ & m|\downarrow n \ . \end{array}$$

Now the claim is immediate. ■

The union $m \cup n$ of two maps m, n is again a map iff m and n are compatible. More generally, for a family $(m_i)_{i \in I}$ of maps (I may even be infinite) the union $\bigcup_{i \in I} m_i$ is a map iff the maps m_i are pairwise compatible. 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:

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

The following decomposition property is the key to recursions over maps:

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

Another important operation is **map overwriting** (see e.g. [Jones 80]): Given maps $m, n : M \longrightarrow N$ we define

$$m \lhd n \stackrel{\text{def}}{\equiv} (m \ominus \downarrow n) \cup n \ .$$

Hence,

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

In other words, $m \lhd n$ results from m by changing the values according to the prescription of n (if any). For example, forming $m \lhd [x \mapsto y]$ makes y the value corresponding to x. This operation will be our main tool for describing selective updating. We now prove a number of useful properties of this operation.

Lemma 3.2
$\downarrow(m \lhd n) \equiv \downarrow m \cup \downarrow n$

Proof:

$$\begin{aligned} & \downarrow(m \leftarrow n) \\ \equiv\ & \downarrow(m \ominus \downarrow n \cup n) \\ \equiv\ & \downarrow(m \ominus \downarrow n) \cup \downarrow n \\ \equiv\ & \downarrow m \backslash \downarrow n \cup \downarrow n \\ \equiv\ & \downarrow m \cup \downarrow n \end{aligned}$$

■

Maps form a monoid under overwriting:

Lemma 3.3 (Monoid)
(1) $\emptyset \leftarrow m \ \equiv\ m \leftarrow \emptyset \ \equiv\ m$
(2) $(l \leftarrow m) \leftarrow n \ \equiv\ l \leftarrow (m \leftarrow n)$

Proof: (1)

$$\begin{aligned} & \emptyset \leftarrow m \\ \equiv\ & \emptyset \ominus \downarrow m \cup m \\ \equiv\ & \emptyset \cup m \\ \equiv\ & m \end{aligned}$$

$$\begin{aligned} & m \leftarrow \emptyset \\ \equiv\ & m \ominus \downarrow \emptyset \cup \emptyset \\ \equiv\ & m \ominus \emptyset \\ \equiv\ & m \end{aligned}$$

(2)

$$\begin{aligned} & (l \leftarrow m) \leftarrow n \\ \equiv\ & (l \ominus \downarrow m \cup m) \ominus \downarrow n \cup n \\ \equiv\ & (l \ominus \downarrow m) \ominus \downarrow n \cup m \ominus \downarrow n \cup n \\ \equiv\ & l \ominus (\downarrow m \cup \downarrow n) \cup m \ominus \downarrow n \cup n \\ \equiv\ & \quad \text{(by Lemma 3.2)} \\ & l \ominus \downarrow (m \leftarrow n) \cup (m \leftarrow n) \\ \equiv\ & l \leftarrow (m \leftarrow n) \end{aligned}$$

■

Lemma 3.4 (Overwriting and union)
$m \leftarrow n \ \equiv\ n \leftarrow m$ iff m and n are compatible.
In this case, $m \leftarrow n \ \equiv\ m \cup n$.

Proof: ($\Rightarrow$) Assume that $m \ominus \downarrow n \cup n \ \equiv\ n \ominus \downarrow m \cup m$. Then

$$m \ominus \downarrow n \ \equiv\ (n \ominus \downarrow m \cup m) \backslash n \ \equiv\ m \backslash n \ .$$

Now

$$\begin{aligned} & m | \downarrow n \\ \equiv\ & \quad \text{(decomposition)} \\ & m \backslash (m \ominus \downarrow n) \\ \equiv\ & m \backslash (m \backslash n) \\ \equiv\ & m \cap n \ . \end{aligned}$$

Since the assumption is symmetric in m and n, we may substitute n, m for m, n and hence obtain also $n|{\downarrow}m \equiv n \cap m$. Now m, n are compatible by commutativity of $\cap$ and Lemma 3.1.

($\Leftarrow$) By the assumption and Lemma 3.1, $m|{\downarrow}n \equiv n|{\downarrow}m$. Hence

$$\begin{array}{ll} & m \lhd n \\ \equiv & m \ominus {\downarrow}n \cup n \\ \equiv & \quad \text{(decomposition)} \\ & m \backslash (m|{\downarrow}n) \cup n \\ \equiv & m \backslash (n|{\downarrow}m) \cup n \\ \equiv & \quad \text{(since } n|{\downarrow}m \subseteq n\text{)} \\ & m \cup n \ . \end{array}$$

Now use symmetry.

■

To proceed further, we now state some general facts about binary operations. Let $\oplus$ be a binary operation. Define

$$\begin{array}{l} x \leq_\oplus y \stackrel{\text{def}}{\Leftrightarrow} x \oplus y \equiv y \\ x \ {}_\oplus\!\leq y \stackrel{\text{def}}{\Leftrightarrow} x \oplus y \equiv x \end{array}$$

Lemma 3.5

(1) If $\oplus$ is idempotent then ${}_\oplus\!\leq, \leq_\oplus$ are reflexive.

(2) If $\oplus$ is commutative then ${}_\oplus\!\leq, \leq_\oplus$ are antisymmetric and converses of each other.

(3) If $\oplus$ is associative then ${}_\oplus\!\leq, \leq_\oplus$ are transitive.

(4) If $\oplus$ has a neutral element 0 then 0 is least w.r.t. $\leq_\oplus$ and greatest w.r.t. ${}_\oplus\!\leq$.

Now, since $\lhd$ is idempotent and associative with neutral element $\emptyset$, the relations ${}_\lhd\!\leq, \leq_\lhd$ are preorders. It turns out that they have interesting equivalent characterizations:

Lemma 3.6

(1) $m \lhd n \equiv n \Leftrightarrow {\downarrow}m \subseteq {\downarrow}n$

(2) $m \lhd n \equiv m \Leftrightarrow n \subseteq m$

Proof: (1)

$$\begin{array}{ll} & m \lhd n \equiv n \\ \Rightarrow & {\downarrow}(m \lhd n) \equiv {\downarrow}n \\ \Rightarrow & {\downarrow}m \cup {\downarrow}n \equiv {\downarrow}n \\ \Leftrightarrow & {\downarrow}m \subseteq {\downarrow}n \ . \end{array}$$

On the other hand, ${\downarrow}m \subseteq {\downarrow}n$ implies $m \ominus {\downarrow}n \equiv \emptyset$ and hence $m \lhd n \equiv m \ominus {\downarrow}n \cup n \equiv n$.

(2) From $m \lhd n \equiv m \ominus {\downarrow}n \cup n \equiv m$ we conclude $n \subseteq m$. Conversely, $n \subseteq m$ implies $m|{\downarrow}n \equiv n$ and hence

$$\begin{array}{ll} & m \\ \equiv & \quad \text{(decomposition)} \end{array}$$

$$
\begin{array}{ll}
 & m \ominus \downarrow n \cup m|\downarrow n \\
\equiv & m \ominus \downarrow n \cup n \\
\equiv & m \lhd n \ .
\end{array}
$$

■

Lemma 3.7 (Sequentialization)
$l \lhd (m \cup n) \ \equiv \ (l \lhd m) \lhd n$ provided m and n are compatible.

Proof: Immediate from Lemma 3.4 and associativity of $\lhd$. ■

Lemma 3.8 (Annihilation)
$m \ \subseteq \ l \ \ \Rightarrow \ \ l \lhd (m \cup n) \ \equiv \ l \lhd n$ provided m and n are compatible.

Proof: Immediate from sequentialization and Lemma 3.6. ■

Lemma 3.9 (Distributivity)
$(l \cup m) \lhd n \ \equiv \ (l \lhd n) \cup (m \lhd n)$ provided l and m are compatible.

Proof:

$$
\begin{array}{ll}
 & (l \cup m) \lhd n \\
\equiv & (l \cup m) \ominus \downarrow n \cup n \\
\equiv & l \ominus \downarrow n \cup m \ominus \downarrow n \cup n \\
\equiv & l \ominus \downarrow n \cup n \cup m \ominus \downarrow n \cup n \\
\equiv & (l \lhd n) \cup (m \lhd n) \ .
\end{array}
$$

■

The following property allows localizing side effects to that part of a store they really affect:

Lemma 3.10 (Localization)
$\downarrow l \ \cap \ \downarrow n \ \equiv \ \emptyset$ implies $(l \cup m) \lhd n \ \equiv \ l \cup (m \lhd n)$
provided l and m are compatible.

Proof:

$$
\begin{array}{ll}
 & (l \cup m) \lhd n \\
\equiv & (l \cup m) \ominus \downarrow n \cup n \\
\equiv & l \ominus \downarrow n \cup m \ominus \downarrow n \cup n \\
\equiv & l \cup m \ominus \downarrow n \cup n \\
\equiv & l \cup (m \lhd n) \ .
\end{array}
$$

■

The map operations introduced enjoy a vast number of further useful algebraic laws. Some of them can be found in [Berger et al. 89].

4 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 treating 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. Moreover, 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 is empty or ends with $\Box$, i.e., 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 $s[i] \stackrel{\text{def}}{\equiv} \Box$. The predicate *ischainable* is given by

$$\begin{array}{l}\mathsf{funct}\ \mathit{ischainable}\ \equiv\ (\mathsf{cellsequ}\ s)\,\mathsf{bool} :\\ \quad \mathsf{if}\ s = <>\ \mathsf{then}\ \mathsf{true}\\ \qquad\qquad \mathsf{else}\ \ \mathit{first}(s) \neq \Box\ \wedge\ \mathit{first}(s) \notin \mathit{rest}(s)\ \wedge\ \mathit{ischainable}(\mathit{rest}(s))\ \ \mathsf{fi}\ ,\end{array}$$

where $<>$ denotes the empty sequence, $\mathit{first}(s) \stackrel{\text{def}}{\equiv} s[1]$, and $\mathit{rest}(s) \stackrel{\text{def}}{\equiv} s[2 : |s|]$. Note that $\mathit{first}(<>) \equiv \Box$. We have

Lemma 4.1
(1) $\mathit{chain}(<x> + s) \equiv [x \mapsto \mathit{first}(s)] \cup \mathit{chain}(s)$.
(2) $\mathit{chain}(t + <x> + s) \supseteq [x \mapsto \mathit{first}(s)]$

Here, $<x>$ is the singleton sequence consisting just of x, and $+$ denotes concatenation.

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

$$\begin{array}{l}\mathsf{funct}\ \mathit{sequ}\ \equiv\ (\mathsf{cell}\ x, \mathsf{state}\ m)\,\mathsf{cellsequ} :\\ \quad \mathsf{if}\ x \notin \downarrow m\ \mathsf{then}\ <>\ \mathsf{else}\ <x> + \mathit{sequ}(m(x), m)\ \mathsf{fi}\ .\end{array}$$

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 $y \in \mathit{sequ}(x, m) \equiv \mathsf{true}$, where

$$\begin{array}{l}\mathsf{funct}\ .\in.\ \equiv\ (\mathsf{cell}\ y, \mathsf{cellsequ}\ s)\,\mathsf{bool} :\\ \quad \mathsf{if}\ s = <>\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ y = \mathit{first}(s)\ \mathsf{cor}\ y \in \mathit{rest}(s)\ \mathsf{fi}\ .\end{array}$$

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

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

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. Note that $\mathit{isanchored}(\Box, m) \equiv \mathsf{true}$. This is reasonable, since $\mathit{sequ}(\Box, m) \equiv <>$. Moreover, we get

Lemma 4.2
(1) $\mathit{isanchored}(x, m) \wedge x \notin \downarrow m \ \equiv\ x = \Box$.
(2) $\mathit{first}(\mathit{sequ}(x, m)) \equiv x$ provided $\mathit{isanchored}(x, m) \equiv \mathsf{true}$.

We say that a pair (x, m) **represents** a chainable sequence s if $\mathit{sequ}(x, m) \equiv s$. The corresponding representation function is

$$\mathsf{funct}\ \mathit{chainrep} \ \equiv\ (\mathsf{cellsequ}\ s : \mathit{ischainable}(s))(\mathsf{cell}, \mathsf{state}):$$
$$\quad (\mathit{first}(s), \mathit{chain}(s))$$

Note that $\mathit{chainrep}(<>) \equiv (\Box, \emptyset)$. Moreover,

Lemma 4.3
(1) $\mathit{chainrep}(s)$ represents s provided $\mathit{ischainable}(s) \equiv \mathsf{true}$, i.e.,
$\quad \mathit{sequ}(\mathit{chainrep}(s)) \equiv s$ provided $\mathit{ischainable}(s) \equiv \mathsf{true}$.
(2) $\mathit{chain}(\mathit{sequ}(x, m)) \subseteq m$ provided $\mathit{isanchored}(x, m) \equiv \mathsf{true}$.

We now investigate how union and overwriting affect chains. To this end we need some more notation. For cell x and state m we define the set $m^*(x)$ of all cells reachable from x via links in m by

$$m^*(x) \stackrel{\text{def}}{\equiv} \bigcup_{i \in \mathbb{N}} m^i(x)$$

where

$$\begin{array}{lcl} m^0(x) & \stackrel{\text{def}}{\equiv} & \{x\} \\ m^{i+1} & \stackrel{\text{def}}{\equiv} & (m^i(x))\!\uparrow\! m\ , \end{array}$$

and $s\!\uparrow\! m$ denotes the image of a set s under m. Note that

$$m^*(x) \ \equiv\ \mathit{set}(\mathit{sequ}(x, m)) \cup \{\Box\}$$

provided $\mathit{isanchored}(x, m) \equiv \mathsf{true}$. Moreover,

$$m^*(x) \ \subseteq\ \{x\} \cup \uparrow m\ .$$

Lemma 4.4
$\mathit{sequ}(x, m \cup n) \equiv \mathit{sequ}(x, m)$ provided $m^*(x) \cap \downarrow n \equiv \emptyset$.

Proof: We use the well-known technique of computational induction (see e.g. [Manna 74]) for proving properties of recursively defined functions: Consider a continuous (or admissible) predicate P and a recursive definition

$$\mathsf{funct}\ g \ \equiv\ \tau[g]$$

with least fixpoint μ_τ of the associated functional τ. If we can show $P[\Omega]$ (where Ω is the totally undefined function) and $\forall\ f : P[f]\ \Rightarrow\ P[\tau[f]]$, then we may infer $P[\mu_\tau]$ as well. Here, we choose the predicate

$$P[f]\ \stackrel{\text{def}}{\Leftrightarrow}\ \forall\ x, m, n :\\ m^*(x) \cap \downarrow n\ \equiv\ \emptyset\ \Rightarrow\ f(x, m \cup n)\ \equiv\ f(x, m)\ .$$

The induction base $P[\Omega]$ is trivial. For the induction step assume $P[f]$ and $m^*(x) \cap \downarrow n\ \equiv\ \emptyset$. This implies, in particular, $x\ \notin\ \downarrow n$. The functional belonging to the definition of *sequ* is

$$\tau\ :\ f\ \mapsto\ (\mathsf{cell}\ x, \mathsf{state}\ m)\,\mathsf{cellsequ}\ :\\ \mathsf{if}\ x\ \notin\ \downarrow m\ \mathsf{then}\ <>\ \mathsf{else}\ <x> + f(m(x), m)\ \mathsf{fi}\ .$$

We calculate

$$\begin{array}{ll} & \tau[f](x, m \cup n)\\ \equiv & \mathsf{if}\ x\ \notin\ \downarrow(m \cup n)\ \mathsf{then}\ <>\\ & \qquad\qquad \mathsf{else}\ <x> + f((m \cup n)(x), m \cup n)\ \mathsf{fi}\\ \equiv & \mathsf{if}\ x\ \notin\ \downarrow m \cup \downarrow n\ \mathsf{then}\ <>\\ & \qquad\qquad \mathsf{else}\ <x> + f((m \cup n)(x), m \cup n)\ \mathsf{fi}\\ \equiv & \quad (\text{since } x\ \notin\ \downarrow n)\\ & \mathsf{if}\ x\ \notin\ \downarrow m\ \mathsf{then}\ <>\ \mathsf{else}\ <x> + f(m(x), m \cup n)\ \mathsf{fi}\\ \equiv & \quad (\text{by the induction hypothesis } P[f], \text{ since } m^*(m(x))\ \subseteq\ m^*(x))\\ & \mathsf{if}\ x\ \notin\ \downarrow m\ \mathsf{then}\ <>\ \mathsf{else}\ <x> + f(m(x), m)\ \mathsf{fi}\\ \equiv & \tau[f](x, m)\ . \end{array}$$

■

Corollary 4.5

$sequ(x, m)\ \equiv\ sequ(x, m \lhd [u \mapsto v])$ provided $u\ \notin\ m^*(x)$.

Proof: First we note that from the definition it is immediate that $.^*$ is monotonic:

$$l\ \subseteq\ n\ \Rightarrow\ \forall\ x : l^*(x)\ \subseteq\ n^*(x)\ .$$

Hence $u\ \notin\ m^*(x)$ implies $u\ \notin\ (m \ominus u)^*(x)$, so that

$$\begin{array}{ll} & sequ(x, m)\\ \equiv & \quad (\text{decomposition})\\ & sequ(x, m \ominus u \cup [u\ \mapsto\ m(u)])\\ \equiv & \quad (\text{by the above lemma})\\ & sequ(x, m \ominus u)\\ \equiv & \quad (\text{by the above lemma})\\ & sequ(x, m \ominus u \cup [u \mapsto v])\\ \equiv & sequ(x, m \lhd [u \mapsto v])\ . \end{array}$$

■

5 Concatenation of Chains "in Situ"

5.1 Specification and First Explicit Solution

We now want to specify and develop an algorithm for concatenating two non-overlapping anchored chains "in situ". First we give the precondition for our desired function:

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

Here, $\mathit{set}(s)$ gives the set of all cells occurring in sequence s. 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

$$\begin{array}{l}\mathsf{funct}\ \mathit{conc}\ \equiv\ (\mathsf{cell}\ x, \mathsf{cell}\ y, \mathsf{state}\ m : \mathit{disjoint}(x,y,m))\,\mathsf{state}: \\ \quad m \Leftarrow \mathit{chain}(\mathit{sequ}(x,m) + \mathit{sequ}(y,m))\ .\end{array}$$

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. Our first goal is a recursion without the "detour" through sequences. We try to obtain it by the familiar unfold/fold technique. We consider the following cases:

Case 1: $x \notin \downarrow m$, i.e., $x = \square$ by $\mathit{isanchored}(x,m)$ and Lemma 4.2(1). Then $\mathit{sequ}(x,m)\ \equiv <>$, and hence

$$\begin{array}{ll} & m \Leftarrow \mathit{chain}(\mathit{sequ}(x,m) + \mathit{sequ}(y,m)) \\ \equiv & m \Leftarrow \mathit{chain}(\mathit{sequ}(y,m)) \\ \equiv & \quad (\text{since } \mathit{chain}(\mathit{sequ}(y,m))\ \subseteq\ m) \\ & m\ .\end{array}$$

Case 2: $x \in \downarrow m$ cand $m(x)\ \notin \downarrow m$, i.e., $m(x) = \square$. Then $\mathit{sequ}(x,m)\ \equiv <x> + \mathit{sequ}(m(x),m)$ $\equiv\ <x>$, and hence

$$\begin{array}{ll} & m \Leftarrow \mathit{chain}(\mathit{sequ}(x,m) + \mathit{sequ}(y,m)) \\ \equiv & m \Leftarrow \mathit{chain}(<x> + \mathit{sequ}(y,m)) \\ \equiv & \quad (\text{by Lemma 4.1(1)}) \\ & m \Leftarrow ([x \mapsto y] \cup \mathit{chain}(\mathit{sequ}(y,m))) \\ \equiv & \quad (\text{annihilation, since } \mathit{chain}(\mathit{sequ}(y,m))\ \subseteq\ m \text{ by Lemma 4.3(2)}) \\ & m \Leftarrow [x \mapsto y]\ .\end{array}$$

Case 3: $x \in \downarrow m$ cand $m(x) \in \downarrow m$. Then $\mathit{sequ}(x,m)\ \equiv <x> + \mathit{sequ}(m(x),m)$, and hence

$$\begin{array}{ll} & m \Leftarrow \mathit{chain}(\mathit{sequ}(x,m) + \mathit{sequ}(y,m)) \\ \equiv & m \Leftarrow \mathit{chain}(<x> + \mathit{sequ}(m(x),m) + \mathit{sequ}(y,m))\end{array}$$

$\equiv$ (by Lemma 4.1(1))

$m \mathbin{\triangleleft} ([x \mapsto m(x)] \cup chain(sequ(m(x), m) + sequ(y, m)))$

$\equiv$ (annihilation, since $[x \mapsto m(x)] \subseteq m$)

$m \mathbin{\triangleleft} chain(sequ(m(x), m) + sequ(y, m))$

$\equiv$ (fold $conc$)

$conc(m(x), y, m)$.

Altogether we have

$$\begin{array}{l}\mathsf{funct}\ conc \equiv (\mathsf{cell}\ x, \mathsf{cell}\ y, \mathsf{state}\ m : disjoint(x, y, m))\ \mathsf{state}: \\ \quad \mathsf{if}\ x = \square\ \mathsf{then}\ m \\ \qquad \mathsf{else}\ \ \mathsf{if}\ m(x) = \square\ \mathsf{then}\ m \mathbin{\triangleleft} [x \mapsto y] \\ \qquad\qquad\qquad \mathsf{else}\ \ conc(m(x), y, m)\ \ \mathsf{fi}\ \ \mathsf{fi}\ .\end{array}$$

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

5.2 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

$$\begin{array}{l}\mathsf{proc}\ powconc \equiv (\mathsf{var\ state}\ m,\ \mathsf{cell}\ x, y : disjoint(x, y, m)) \\ \quad m := conc(m, x, y)\ .\end{array}$$

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:

$$\begin{array}{l}\mathsf{proc}\ p \equiv (\mathsf{var\ m}\ a,\ \mathsf{n}\ b : P) : \quad a := f(a, b) \\ \mathsf{where} \\ \mathsf{funct}\ f \equiv (\mathsf{m}\ a, \mathsf{n}\ b)\ \mathsf{m}: \\ \quad \mathsf{if}\ C\ \mathsf{then}\ T\ \mathsf{else}\ f(K, L)\ \mathsf{fi} \\ \hline \\ \mathsf{proc}\ p \equiv (\mathsf{var\ m}\ a,\ \mathsf{n}\ B : P[\![B\ \mathsf{for}\ b]\!]) : \\ \quad \lceil\ \mathsf{var\ n}\ b := B\ ; \\ \quad\ \ \mathsf{while}\neg\ C\ \mathsf{do}\ (a, b) := (K, L)\ \mathsf{od}\ ; \\ \quad\ \ a := T \qquad\qquad\qquad\qquad \rfloor\ .\end{array} \quad \Big\uparrow \Big\downarrow \quad [\ \mathrm{NEW}[\![B]\!]$$

Note that a, b, and B may stand for tuples of identifiers. P, C, T, K, and L stand for expressions of appropriate types, possibly involving a and b. The condition NEW$[\![B]\!]$ states that B has to be a (tuple of) fresh identifier(s). The rule does not match our current form of *conc*; however, this is easily transformed into

$$\begin{array}{l}\mathsf{funct}\ conc\ \equiv\ (\mathsf{cell}\ x, \mathsf{cell}\ y, \mathsf{state}\ m : disjoint(x,y,m))\,\mathsf{state}: \\ \quad \mathsf{if}\ x = \square\ \mathsf{cor}\ m(x) = \square \\ \quad\ \mathsf{then\ if}\ x = \square\ \mathsf{then}\ m\ \mathsf{else}\ m{\Leftarrow}[x \mapsto y]\ \mathsf{fi} \\ \quad\ \mathsf{else}\ \ conc(m(x), y, m) \qquad\qquad\qquad\qquad \mathsf{fi}\ .\end{array}$$

Now the rule applies leading to

$$\begin{array}{l}\mathsf{proc}\ powconc\ \equiv\ (\mathsf{var\ state}\ m, \mathsf{cell}\ X, Y : disjoint(X,Y,m)): \\ \quad \lceil\ (\mathsf{var\ cell}\ x, y) := (X, Y)\ ; \\ \quad\ \mathsf{while} x\ \neq\ \square\ \mathsf{cand}\ m(x)\ \neq\ \square\ \mathsf{do}\ (m, x, y) := (m, m(x), y)\ \mathsf{od}\ ; \\ \quad\ m := \mathsf{if}\ x = \square\ \mathsf{then}\ m\ \mathsf{else}\ m \Leftarrow [x \mapsto y]\ \mathsf{fi} \qquad\qquad\qquad\qquad \rfloor\ .\end{array}$$

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

$$\begin{array}{l}\mathsf{proc}\ powconc\ \equiv\ (\mathsf{var\ state}\ m, \mathsf{cell}\ X, Y : disjoint(X,Y,m)): \\ \quad \lceil\ \mathsf{var\ cell}\ x := X\ ; \\ \quad\ \mathsf{while} x\ \neq\ \square\ \mathsf{cand}\ m(x)\ \neq\ \square\ \mathsf{do}\ x := m(x)\ \mathsf{od}\ ; \\ \quad\ \mathsf{if}\ x = \square\ \mathsf{then\ skip} \\ \qquad\qquad\ \mathsf{else}\ \ m := m \Leftarrow [x \mapsto Y]\ \ \mathsf{fi} \qquad\qquad\qquad \rfloor\ .\end{array}$$

If we write the assignment

$$m := m \Leftarrow [x \mapsto Y]$$

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. By standard transformation techniques one can derive a version with a simpler loop test that avoids double inspection of cells.

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.

6 Chain Reversal

6.1 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:

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

where *rev* is the reversal function on sequences:

$$\textsf{funct}\ rev\ \equiv\ (\textsf{cellsequ}\ s)\textsf{cellsequ}:\ \textsf{if}\ s = <>\ \textsf{then}\ <>\ \textsf{else}\ rev(rest(s)) + <first(s)>\ \textsf{fi}\ .$$

Let us now derive a recursion for *reverse*. The basic idea for the development is to adapt the standard technique for making *rev* tail-recursive. There one defines a generalized function *rrev* with an additional parameter that accumulates the intermediate results:

$$rrev(s,t)\ \stackrel{\text{def}}{\equiv}\ rev(s) + t\ .$$

rev is embedded into *rrev* by

$$rev(s)\ \equiv\ rrev(s, <>)\ .$$

A straightforward unfold/fold derivation using associativity of + leads to the tail-recursion

$$\begin{array}{ll} & rrev(s,t) \\ \equiv & \textsf{if}\ s = <>\ \textsf{then}\ t \\ & \qquad \textsf{else}\ \ rrev(rest(s), <first(s)> + t)\ \ \textsf{fi}\ . \end{array}$$

In the case of *reverse* we now proceed similarly; however, we do not carry the accumulating submap itself as a parameter, but just its head cell. Hence we define

$$\textsf{funct}\ rreverse\ \equiv\ (\textsf{cell}\ x, y, \textsf{state}\ m : disjoint(x,y,m))\,\textsf{state}:\ m \Leftarrow chain(rev(sequ(x,m)) + sequ(y,m))\ .$$

An appropriate embedding is

$$reverse(x,m)\ \equiv\ rreverse(x, \Box, m)\ ,$$

since $sequ(\Box, m)\ \equiv <>$.

As before, we now perform a case analysis.

Case 1: $x = \Box$. Then $sequ(x,m)\ \equiv <>$, and hence $rev(sequ(x,m))\ \equiv <>$. Thus

$$\begin{array}{ll} & m \Leftarrow chain(rev(sequ(x,m)) + sequ(y,m)) \\ \equiv & m \Leftarrow chain(sequ(y,m)) \\ \equiv & \qquad \text{(since } chain(sequ(y,m))\ \subseteq\ m \text{ by Lemma 4.3(2))} \\ & m\ . \end{array}$$

Case 2: $x\ \neq\ \Box$. Then $sequ(x,m)\ \equiv <x> + sequ(m(x), m)$, and hence $rev(sequ(x,m))\ \equiv rev(sequ(m(x),m)) + <x>$. Thus

$$\begin{array}{ll} & chain(rev(sequ(x,m)) + sequ(y,m)) \\ \equiv & chain(rev(sequ(m(x),m)) + <x> + sequ(y,m)) \\ \equiv & \qquad \text{(by Corollary 4.5, since } disjoint(x,y,m) \text{ implies } x\ \notin\ sequ(y,m)) \\ & chain(rev(sequ(m(x),m)) + <x> + sequ(y, m \Leftarrow [x \mapsto y])) \\ \equiv & \qquad \text{(definition of } sequ) \\ & chain(rev(sequ(m(x),m)) + sequ(x, m \Leftarrow [x \mapsto y])) \\ \equiv & \qquad \text{(by Corollary 4.5, since by cyclefreeness } x\ \notin\ sequ(m(x),m)) \\ & chain(rev(sequ(m(x), m \Leftarrow [x \mapsto y])) + sequ(x, m \Leftarrow [x \mapsto y]))\ . \end{array}$$

Now we obtain

$$
\begin{array}{ll}
 & m \Leftarrow chain(rev(sequ(x,m)) + sequ(y,m)) \\
\equiv & m \Leftarrow chain(rev(sequ(m(x),m)) + <x> + sequ(y,m)) \\
\equiv & \quad \text{(by Lemma 4.1(2) and Lemma 4.2(2))} \\
 & m \Leftarrow ([x \mapsto y] \cup chain(rev(sequ(m(x),m)) + <x> + sequ(y,m))) \\
\equiv & \quad \text{(sequentialisation)} \\
 & m \Leftarrow [x \mapsto y] \Leftarrow chain(rev(sequ(m(x),m)) + <x> + sequ(y,m)) \\
\equiv & \quad \text{(by the above subderivation)} \\
 & m \Leftarrow [x \mapsto y] \Leftarrow chain(rev(sequ(m(x), m{\Leftarrow}[x \mapsto y])) + sequ(x, m{\Leftarrow}[x \mapsto y])) \\
\equiv & \quad \text{(fold } rreverse\text{)} \\
 & rreverse(m(x), x, m{\Leftarrow}[x \mapsto y]) \ .
\end{array}
$$

Altogether, we have

$$
\begin{array}{l}
\mathsf{funct}\ rreverse \ \equiv \ (\mathsf{cell}\ x, y, \mathsf{state}\ m : disjoint(x,y,m))\ \mathsf{state} : \\
\quad \mathsf{if}\ x = \Box\ \mathsf{then}\ m \\
\qquad\quad \mathsf{else}\ \ rreverse(m(x), x, m{\Leftarrow}[x \mapsto y])\ \ \mathsf{fi}\ .
\end{array}
$$

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

6.2 A Version With Selective Updating

Specifying a procedure

$$
\begin{array}{l}
\mathsf{proc}\ powrev \ \equiv \ (\mathsf{var\ state}\ m, \mathsf{cell}\ x : isanchored(x,m)) : \\
\quad m := reverse(x,m)\ ,
\end{array}
$$

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

$$
\begin{array}{l}
\mathsf{proc}\ powrev \ \equiv \ (\mathsf{var\ state}\ m, \mathsf{cell}\ X : isanchored(X,m)) : \\
\quad \lceil\ (\mathsf{var\ cell}\ x, y) := (X, \Box)\ ; \\
\quad\ \ \mathsf{while}\ x \neq \Box \\
\quad\quad \mathsf{do}\ \ (x,y,m) := (m(x), x, m \Leftarrow [x \mapsto y])\ \mathsf{od}\ \rfloor\ .
\end{array}
$$

Note that sequentialization of the collective assignment would require an auxiliary variable. This is a spot of frequent error in attempts to write down this algorithm straightforwardly without deriving it. The systematic derivation allows us to avoid such errors by using the standard knowledge about the treatment of collective assignments.

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. [Burstall 72, Levy 78]), 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.

7 A Garbage Collection Problem

We now want to present the specification and parts of the derivation of a garbage collection algorithm; the full details are given in [Berger et al. 89]. For earlier attempts at similar problems

cf. [Broy, Pepper 82], [Dewar et al. 82], [van Diepen, de Roever 86]. Differing from all of these, [Berger et al. 89] develops the algorithm to a level which can actually be transcribed directly into machine code allowing the use of overwriting, address arithmetic, and the like. Again, the algebra of partial maps is the most important tool.

The situation in which garbage collection becomes necessary is the following: The store, which accomodates a large number of records referencing one another through pointers, is exhausted, i.e., there is (almost) no more free storage left for the allocation of new records. Usually there is a distinguished set of **entry pointers** to the pointer structure which is given by the values of the currently active variables of the program that operates on the store. Only those records reachable through chains of references from the entry pointers actually need to be saved; all other records are inaccessible and thus the corresponding storage can be reclaimed.

What does garbage collection mean in a more abstract sense? To explain this, we liberate ourselves from the concrete contents of the records in the store and consider only their interrelationship through the pointers. This leads to a graph-like structure G in which the nodes correspond to the records, and the arcs correspond to the pointers. If, by some process, we can distinguish a proper subgraph G' of G such that G' contains all the nodes accessible from the entry nodes, then G can be said to contain garbage about which we could as well forget. In this case, garbage collection means to compute G' from G and to operate on G' successively.

Getting more concrete again, we work with representations of such graphs. The task then consists in computing a representation of G' from one of G. In this paper we treat representations, called **states**, of graphs in a linear storage. For a state, the restriction to a substate usually leads to gaps in the storage; i.e., there are cells the contents of which have no meaning for the represented graph. Now, one possibility of garbage collection consists in detecting these gaps and compactifying the meaningful part by copying it to an initial interval of the storage; then a contiguous rest of the storage becomes free for further use.

We give a formal specification of this problem at the level of graphs together with a notion of their representation in a linear storage. Then we treat in detail the copying algorithm involved which is the center of the whole algorithm derived in [Berger et al. 89].

7.1 Storage Graphs

Storage graphs are intended to model the accessibility relations between records as given by the pointers in the records. Since the fields of a record are ordered and may contain repetitions of pointers, the usual notion of a directed graph where each node is connected to a *set* of successor nodes is not adequate for our purposes. Rather we consider *sequences* of successor nodes. Also, since we shall have to deal with arbitrary parts of such storage graphs, we generalize in another direction by allowing the successor map to "leave" the part under consideration.

Let node be a set of "nodes". Given a set M, we denote by M^* the set of all finite sequences of M-elements. Then **pseudo-graph** is a partial map $G : \mathsf{node} \longrightarrow \mathsf{node}^*$ such that $\downarrow G$ is finite. For a node $x \in \downarrow G$ the nodes in $G(x)$ are called the **immediate successors** of x. We set

$$nodes(G) \stackrel{\text{def}}{=} \downarrow G \cup \bigcup_{x \in \downarrow G} set(G(x)) \ .$$

A **storage graph** then is a pseudo-graph G such that $nodes(G) \subseteq \downarrow G$. The more general notion of pseudo-graphs also allows "dangling references" which will occur e.g. during the

copying phase of our garbage collection algorithm when only part of the accessible cells have been copied to their new locations.

7.2 Allocations

We also want to talk about the representation of such pseudo-graphs in a linear memory. Hence we now assume that cell is denumerable and linearly ordered by some ordering $\leq$ in which $\Box$ is the least element. Without loss of generality we assume cell to be the set $\mathbb{N} \cup \{\infty\}$ of natural numbers under the usual ordering, enlarged by a greatest element ∞; then $\Box \equiv 0$.

Let now G be a pseudo-graph. We want to represent G by a state. The idea is to store each node of G together with its successors in a block of contiguous cells; the node itself is marked by cell contents $\Box$. In practice, this leading cell frequently is used for storing information about a record, such as its length, type information, and the like. However, as stated in the introduction, we abstract from such details; $\Box$ seems an adequate substitute here.

An **allocation** of G is an injective partial map $g : \mathsf{node} \longrightarrow \mathsf{cell}$ such that $nodes(G) \subseteq \downarrow g$ and $\Box \notin \uparrow g$. It is supposed to assign to each node in $\downarrow G$ the starting cell of its block; the additional condition ensures that $\Box$ can in fact be used to characterize block beginnings. Given an allocation g of G we define for $x \in \downarrow G$

$$block(x, g) \stackrel{\text{def}}{\equiv} [g(x) \mapsto \Box] \cup \bigcup_{i \in [1:|G(x)|]} [(g(x) + i) \mapsto g(G(x)[i])] .$$

Define for a subset $s \subseteq \mathsf{cell}$

$$\begin{aligned} s^{\vee} &\stackrel{\text{def}}{\equiv} \{y \mid \exists\, x \in s : x \leq y\} \\ s^{\wedge} &\stackrel{\text{def}}{\equiv} \{y \mid \exists\, x \in s : y \leq x\} . \end{aligned}$$

We call s an **interval** if $s \equiv s^{\vee} \cap s^{\wedge}$. For abbreviation we write $\overset{\vee}{x}$, $\hat{x}$ instead of $\{x\}^{\vee}$, $\{x\}^{\wedge}$.

Corollary 7.1
$\downarrow block(x, g)$ is the interval $[g(x) : g(x) + |G(x)|]$.

An allocation g of G is **overlap-free** if

$$x \not\equiv y \Rightarrow \downarrow block(x, g) \cap \downarrow block(y, g) \equiv \emptyset .$$

For an overlap-free allocation we can extend the function *block* to sets $s \subseteq \downarrow G$ by setting

$$block(s, g) \stackrel{\text{def}}{\equiv} \bigcup_{x \in s} block(x, g) .$$

Then the following state is a **representation** of G:

$$blockrep(G, g) \stackrel{\text{def}}{\equiv} block(\downarrow G, g) .$$

We call a state m a **pseudo-graph state** if $m \equiv blockrep(G, g)$ for some pseudograph G and some overlap-free allocation g of G. Then

$$keys(m) \stackrel{\text{def}}{\equiv} \Box \downarrow m ,$$

the inverse image of $\Box$ under m, denotes the set of **keys** of m, i.e., the set of cells that are the beginnings of blocks. Moreover we define

$$followers(m, y) \equiv \mathsf{if}\ y+1 \in keys(m)\ \mathsf{then}\ <>\ \mathsf{else}\ <y+1> + followers(m, y+1)\ \mathsf{fi}.$$

This is the sequence of cells following y in the block to which y belongs.

The following lemma shows how a graph can be reconstructed from its block representation:

Lemma 7.2
Let g be an overlap-free allocation of G and let $m \equiv blockrep(G, g)$. Then

$$\begin{array}{lcl} \downarrow G & \equiv & keys(m)\downarrow g \qquad \text{and} \\ G & \equiv & \bigcup_{x \in keys(m)} [x\downarrow g \mapsto g^{-1} * (m * followers(m, x))] \end{array}$$

where $keys(m)\downarrow g$ is the inverse image of $keys(m)$ under g and for a function $f : M \longrightarrow N$ we denote by $f*$ its unique homomorphic extension mapping M^* to N^*, i.e.,

$$f * (<x_1, \ldots, x_n>) \equiv <f(x_1), \ldots, f(x_n)> \ .$$

Assume now that $nodes(G)$ is linearly ordered by some order $\leq$. Then an allocation g is called **order-preserving** if

$$\forall\ x, y \in nodes(G) : x \leq y \Rightarrow g(x) \leq g(y)\ .$$

Lemma 7.3
Let g be an order-preserving allocation. Then for $x, y \in nodes(G)$ we have

1. $x < y \Rightarrow g(x) < g(y)$.
2. $x \leq y$ iff $g(x) \leq g(y)$, i.e., $nodes(G)$ and $\uparrow g$ are order-isomorphic.

Proof:
1. is immediate from the injectivity of g.
2. We only need to show ($\Leftarrow$). Assume $g(x) \leq g(y)$ but $x \not\leq y$. By linearity of $\leq$ then $y < x$ and hence also $g(y) < g(x)$ by 1. Contradiction!

■

This lemma holds for arbitrary order-preserving injections between linear orders. For the special case of graph linearization we get

Lemma 7.4
Let g be an overlap-free and order-preserving allocation. Then

$$x < y \Rightarrow \downarrow block(x, g) < \downarrow block(y, g)\ ,$$

where for subsets s, t of an ordered set

$$s < t \stackrel{\text{def}}{\Leftrightarrow} \forall\ x \in s : \forall\ y \in t : x < y\ .$$

Proof: Since g is order-preserving, $g(x) < g(y)$. Let now $u \in \downarrow block(x, g)$ and $v \in \downarrow block(y, g)$ and assume $v \leq u$. Since $g(y) \leq v$, we have then $g(x) \leq g(y) \leq u$ and hence $g(y) \in \downarrow block(x, g)$, since $\downarrow block(x, g)$ is an interval. But then $\downarrow block(x, g) \cap \downarrow block(y, g) \not\equiv \emptyset$, a contradiction. ■

We now want to characterize contiguous block representations. Call an overlap-free allocation g of G **gap-free** if $\downarrow\mathit{blockrep}(G,g)$ is an interval. g is called **perfect** if it is overlap-free, order-preserving, and gap-free.

For a finite linearly ordered set M with greatest element ∞ we define for $s \subseteq M$ and $x \in M\backslash\{\infty\}$

$$\mathit{succ}_s(x) \stackrel{\text{def}}{\equiv} \mathit{min}((s\backslash \hat{x}) \cup \{\infty\}) \ .$$

Lemma 7.5

Let $g : \mathsf{node} \longrightarrow \mathsf{cell}$ be a perfect allocation of G and $x \in \downarrow G$ such that x is not the maximum of $\downarrow G$. Then

$$g(\mathit{succ}_{\downarrow G}(x)) \ \equiv \ g(x) + |G(x)| + 1 \ .$$

Proof: Since g is order-preserving and injective, we have $g(x) < g(\mathit{succ}_{\downarrow G}(x))$. The previous lemma now implies $\downarrow\mathit{block}(x,g) < \downarrow\mathit{block}(\mathit{succ}_{\downarrow G}(x),g)$ and, in particular, $\downarrow\mathit{block}(x,g) < \{\mathit{succ}_{\downarrow G}(x)\}$. Let

$$z \stackrel{\text{def}}{\equiv} \mathit{max}(\downarrow\mathit{block}(x,g)) \ \equiv \ g(x) + |G(x)| \ .$$

Assume $z < u < g(\mathit{succ}_{\downarrow G}(x))$ for some u. Since $\downarrow\mathit{blockrep}(G,g))$ is an interval, we get $u \in \downarrow\mathit{blockrep}(G,g)$, say $u \in \downarrow\mathit{block}(w,g)$ for some $w \in \downarrow G$. Then $g(w) \leq u < g(\mathit{succ}_{\downarrow G}(x))$ and hence $w < \mathit{succ}_{\downarrow G}(x)$, since g is order-preserving and injective. This is equivalent to $w \leq x$. But then $\downarrow\mathit{block}(w,g) \leq \downarrow\mathit{block}(x,g)$ which, by Lemma 7.4, contradicts $g(x) \leq z < u \in \downarrow\mathit{block}(w,g)$. Therefore

$$g(\mathit{succ}_{\downarrow G}(x)) \ \equiv \ z+1 \ \equiv \ g(x) + |G(x)| + 1 \ .$$

■

Set, for $x \in \downarrow G$,

$$\|x\| \stackrel{\text{def}}{\equiv} |G(x)| + 1 \ .$$

Corollary 7.6

Let $g : \mathsf{node} \longrightarrow \mathsf{cell}$ be a perfect allocation of G and $x \in \downarrow G$ such that $|\ \check{x} \cap \downarrow G| > i$. Then

$$g(\mathit{succ}^i_{\downarrow G}(x)) \ \equiv \ g(x) + \sum_{j<i} \|\mathit{succ}^j_{\downarrow G}(x)\| \ .$$

Proof: Induction on i using the above lemma. ■

A pseudo-graph state m is called **compressed** if

$$\downarrow m \ \equiv \ (\downarrow m)^{\wedge}\backslash\{\Box\} \ ,$$

i.e., if its domain is an initial interval of $\mathsf{cell}\backslash\{\Box\}$, which implies that there are no gaps in $\downarrow m$. By the above corollary, given a pseudo-graph G there is exactly one perfect allocation g of G such that $\mathit{blockrep}(G,g)$ is compressed; g is called the **compressing allocation** of G.

7.3 Formal Specification of the Garbage Collection Problem

When garbage collection becomes necessary, there is a set of immediate entries into the store. All blocks reachable from these entries need to be saved whereas everything else is garbage to be removed. We first treat the reachability problem at the level of storage graphs.

Let G be a pseudo-graph and let $x, y \in \mathsf{node}$. A sequence $p \in \mathsf{node}^*$ is called a **path in** G **from** x **to** y iff the predicate $ispath_G(p, x, y)$ holds, where

$$\begin{aligned} ispath_G(p,x,y) \;\stackrel{\text{def}}{\Leftrightarrow}\; & |p| > 0 \;\wedge\; set(p)\backslash\{last(p)\} \;\subseteq\; {\downarrow}G \;\wedge \\ & x = first(p) \;\wedge\; y = last(p) \;\wedge \\ & \forall\, i \in [1 : |p| - 1] : p[i+1] \in G(p[i]) \;. \end{aligned}$$

We define

$$\mathsf{nodeset} \;\stackrel{\text{def}}{\equiv}\; \{s \,|\, s \;\subseteq\; \mathsf{node} \;\wedge\; |s| < \infty\} \;.$$

Given a pseudo-graph G, a node $x \in nodes(G)$ is **reachable** from some set $s \in \mathsf{nodeset}$ iff the predicate $isreachable_G(x, s)$ holds where

$$isreachable_G(x,s) \;\stackrel{\text{def}}{\Leftrightarrow}\; \exists\, z \in s, p \in \mathsf{node}^* : ispath_G(p,z,x) \;.$$

The function $rnset_G : \mathsf{nodeset} \longrightarrow \mathsf{nodeset}$, defined by

$$rnset_G(s) \;\stackrel{\text{def}}{\equiv}\; \{\; x \in \mathsf{node} \mid isreachable_G(x,s) \;\}$$

computes the set of nodes reachable from a given set.

Now, for a storage graph G and a set $s \;\subseteq\; {\downarrow}G$, the **subgraph of** G **reachable from** s is $G_s \;\stackrel{\text{def}}{\equiv}\; G|rnset_G(s)$. It is easily verified that the pseudo-graph G_s indeed is a storage graph.

Consider now a storage graph G with a perfect allocation g and a set $s \;\subseteq\; {\downarrow}G$. Moreover, set $n \;\stackrel{\text{def}}{\equiv}\; blockrep(G, g)$. Then the **garbage collection problem** consists in computing the reachable subgraph G_s together with the compressing allocation g_s of G_s as well as the corresponding state $n_s \;\stackrel{\text{def}}{\equiv}\; blockrep(G_s, g_s)$. In fact, ultimately we are interested in an algorithm that computes n_s directly from n.

7.4 A First Analysis of the Problem

Assume G, g, n and G_s, g_s, n_s as in Section 7.3. Define

$$n_1 \;\stackrel{\text{def}}{\equiv}\; blockrep(G_s, g) \;.$$

Thus, n_1 is the accessible but not yet compressed part of the storage. To compute n_s from n_1, define a collapsing map $k : {\downarrow}n_1 \longrightarrow \mathsf{cell}$ by

$$k(g(x) + i) \;\stackrel{\text{def}}{\equiv}\; g_s(x) + i$$

for $x \in {\downarrow}G_s$ and $0 \leq i \leq |G_s(x)|$.

Lemma 7.7

(1) k is well-defined. Moreover, k is an order-embedding and ${\uparrow}k \;\equiv\; {\downarrow}n_s$ (which is an initial interval of $\mathsf{cell}\backslash\{\Box\}$).

(2) $n_s \circ k \;\equiv\; k \circ n_1$.

Proof: (1) is obvious.

$$\begin{array}{lll} (2) & & n_s(k(g(x)+i)) \\ & \equiv & n_s(g_s(x)+i) \\ & \equiv & g_s(G(x)[i]) \\ & \equiv & k(g(G(x)[i])) \\ & \equiv & k(n_1(g(x)+i)) \ . \end{array}$$

■

The preceding considerations suggest a decomposition of the problem into the following parts:

1. Compute G_s from G and s (reachability).
2. Compute $n_1 \equiv \mathit{blockrep}(G_s, g)$ from G_s and g.
3. Compute k from n_1.
4. Compute n_s from n_1 and k (copying).

Diagrammatically the situation can be described as follows:

$$\begin{array}{ccccc} G & \stackrel{reach}{\Longrightarrow} & G_s & & \\ \Big\Downarrow g & & \Big\Downarrow g & & g_s \\ \mathit{blockrep}(G,g) & \supseteq & \mathit{blockrep}(G_s,g) & \stackrel{k}{\Longrightarrow} & \mathit{blockrep}(G_s,g_s) \end{array}$$

Here the double arrows indicate that the respective functions are of second order, since their arguments, viz. pseudo-graphs and states, are mappings themselves.

8 Copying Pointer Structures

8.1 Statement of the Problem

We now treat the task of copying a state to another part of a memory. For a map p we define

$$\mathit{set}(p) \stackrel{\text{def}}{\equiv} \downarrow p \cup \uparrow p \ .$$

Let now m, n be states. We call n a **copy** of m if there is a total bijection

$$k : \mathit{set}(m) \cup \{\Box\} \longrightarrow \mathit{set}(n) \cup \{\Box\}$$

such that $k(\Box) \equiv \Box$ and the following diagram commutes:

$$\begin{array}{ccc} \downarrow m & \stackrel{m}{\longrightarrow} & \uparrow m \\ \Big\downarrow k & & \Big\downarrow k \\ \downarrow n & \stackrel{n}{\longrightarrow} & \uparrow n \end{array}$$

This means that $k \circ m \equiv n \circ k$ and, since k is bijective, that $n \equiv k \circ m \circ k^{-1}$. Hence, given k, we can compute n from m in two passes: First we form $k \circ m$ which means that the cell contents

as given by m are updated to contain the corresponding cells of the copy ("pointer relocation pass"); then we compose with k^{-1} which means the actual transport of the new contents to the new locations ("copying pass").

For the treatment of these two passes the following properties of general maps are useful:

Lemma 8.1
Consider a map $l : M \longrightarrow N$.

$$
\begin{array}{lll}
(1) & l \equiv \bigcup\limits_{x \in \downarrow l} [x \mapsto l(x)] & \text{(domain-oriented representation)} \\
(2) & l \equiv \bigcup\limits_{z \in \uparrow l} [z \downarrow l \mapsto z] & \text{(range-oriented representation)} \\
(3) & q \circ \bigcup\limits_{i \in I} l_i \equiv \bigcup\limits_{i \in I} (q \circ l_i) & \\
(4) & (\bigcup\limits_{i \in I} l_i) \circ r \equiv \bigcup\limits_{i \in I} (l_i \circ r) & \\
(5) & l^{-1} \equiv \bigcup\limits_{x \in \downarrow l} [l(x) \mapsto x] & \text{provided } l \text{ is injective.}
\end{array}
$$

8.2 Copying Pass

Given $p \equiv k \circ m$, the copying pass is easily performed. First, by totality of k, we have $\downarrow p \equiv \downarrow m$ ($\subseteq \downarrow k$). Now

$$
\begin{array}{ll}
 & p \circ k^{-1} \\
\equiv & \quad \text{(by Lemma 8.1(5))} \\
 & p \circ \bigcup\limits_{x \in \downarrow k} [k(x) \mapsto x] \\
\equiv & \quad \text{(by Lemma 8.1(3))} \\
 & \bigcup\limits_{x \in \downarrow k} p \circ [k(x) \mapsto x] \\
\equiv & \bigcup\limits_{x \in \downarrow k} [k(x) \mapsto p(x)] \\
\equiv & \bigcup\limits_{x \in \downarrow p} [k(x) \mapsto p(x)] \ ,
\end{array}
$$

since $[k(x) \mapsto p(x)] \equiv \emptyset$ for $x \in \downarrow k \backslash \downarrow p \equiv \downarrow k \backslash \downarrow m$. We set

$$
copass(p, k) \stackrel{\text{def}}{\equiv} \downarrow p \subseteq \downarrow k \rhd \bigcup_{x \in \downarrow p} [k(x) \mapsto p(x)] \ .
$$

8.3 Pointer Relocation

The more difficult subtask consists in computing the composition $k \circ m$ efficiently. According to Lemma 8.1 there are essentially two ways of forming $k \circ m$:

1. domain-oriented:
$$
k \circ m \equiv \bigcup_{x \in \downarrow m} [x \mapsto k(m(x))]
$$
If we look at the union as a loop, this way of forming $k \circ m$ needs an explicit representation of k, since the same value of k may be needed repeatedly at irregular intervals.

2. range-oriented:

$$k \circ m \;\equiv\; \bigcup_{z \in \uparrow m} [z{\downarrow}m \;\mapsto\; k(z)]$$

For evaluating this by a loop we only need one value of k at a time to process a whole subset of $\downarrow m$. Hence we can avoid explicit representation of the complete k, which is particularly important in garbage collection, where storage is almost exhausted. Moreover, the repeated lookups are avoided and thus also time-efficiency is improved. Of course, this latter aspect is interesting only if m is highly non-injective so that the inverse images $z{\downarrow}m$ are large.

We follow now the range-oriented variant. We need a way of representing the component maps $[z{\downarrow}m \;\mapsto\; y]$ suitably. For this we use an idea that is presented e.g. in [Dewar, McCann 77]: All elements of $z{\downarrow}m$ are chained into a linked list; then $[z{\downarrow}m \mapsto y]$ can be formed following the chain as $\bigcup_{x \in z{\downarrow}m} [x \mapsto y]$.

8.3.1 Chained Representation of Sets

Based on our notion of representation for sequences from section 4 we now introduce a chained representation of sets of proper cells: We represent such a set by a repetition-free sequence and this, in turn, by an anchored chain. Formally, a pair (x, m) **represents** a set s of proper cells if $set(sequ(x, m)) \;\equiv\; s$. We define

$$from(x, m) \;\stackrel{\text{def}}{\equiv}\; chain(sequ(x, m)) \;.$$

This is the subchain started by x in m. Furthermore, we set

$$ischain(x, m) \;\stackrel{\text{def}}{\equiv}\; m = from(x, m) \;.$$

Now a representation function for sets is specified by

$$\begin{array}{l}\mathsf{funct}\ chainset \;\equiv\; (\mathsf{cellset}\ s)\ (\mathsf{cell}, \mathsf{state}) : \\ \quad \mathsf{some}\ \mathsf{cell}\ x, \mathsf{state}\ s : ischain(x, m)\ \mathsf{cand}\ set(sequ(x, m)) \;=\; s \;.\end{array}$$

We want to develop an incrementation function

$$\begin{array}{rcl} add(y, x, m) & \stackrel{\text{def}}{\equiv} & ischain(x, m)\ \mathsf{cand}\ y \notin set(sequ(x, m)) \cup \{\square\} \rhd \\ & & chainset(set(sequ(x, m)) \cup \{y\}) \;.\end{array}$$

To this end we observe that

$$y \notin set(sequ(x, m)) \cup \{\square\} \quad \Rightarrow \quad ischainable(<y> + sequ(x, m)) \;,$$

and hence

$$\begin{array}{ll} & set(sequ(chainrep(<y> + sequ(x, m)))) \\ \equiv & \quad \text{(by Lemma 4.3(1))} \\ & set(<y> + sequ(x, m)) \\ \equiv & \{y\} \cup set(sequ(x, m)) \;,\end{array}$$

so that $(z,l) \stackrel{\text{def}}{\equiv} chainrep(<y> + sequ(x,m))$ satisfies the second conjunct of the specification of $chainset(set(sequ(x,m)) \cup \{y\})$. Using Lemma 4.1(1), Lemma 4.2(2), and $ischain(x,m)$ this simplifies to

$$\begin{aligned} z &\equiv y\ , \\ l &\equiv [y \mapsto x] \cup m\ . \end{aligned}$$

Now $ischain(z,l)$ is easily checked, so that

$$\begin{aligned} add(y,x,m) \equiv\ & ischain(x,m) \wedge y \notin set(sequ(x,m)) \cup \{\Box\} \rhd \\ & (y, [y \mapsto x] \cup m) \end{aligned}$$

is a correct refinement (i.e., a descendant) of our specification. Often we are interested just in the second component of the result of add. Hence we define

$$\begin{aligned} prefix(y,x,m) \stackrel{\text{def}}{\equiv}\ & ischain(x,m) \wedge y \notin set(sequ(x,m)) \cup \{\Box\} \rhd \\ & [y \mapsto x] \cup m\ . \end{aligned}$$

Corollary 8.2

Assume $ischain(x,m) \wedge y \notin set(sequ(x,m)) \cup \{\Box\}$. Then
(1) $prefix(y,x,m)(y) \equiv x$.
(2) $sequ(x, prefix(y,x,m)) \equiv sequ(x,m)$.

Proof: (1) is immediate from the definition.

(2) follows from Corollary 4.5, since $prefix(y,x,m) \equiv m \leftarrow [y \mapsto x]$.

■

8.3.2 Chained Representation of States

Let now m be a state. From the range-oriented decomposition $m \equiv \bigcup_{z \in \uparrow n} [z{\downarrow}m \mapsto z]$ we obtain the partition ${\downarrow}m \equiv \bigcup_{z \in \uparrow m} z{\downarrow}m$. We represent m by a union of chains each of which represents one of the sets $z{\downarrow}m$; the cell z is prefixed as a header cell to the respective chain. To avoid confusion between these chains we require that

$$ischainable(m) \stackrel{\text{def}}{\Leftrightarrow} {\downarrow}m \cap {\uparrow}m \equiv \emptyset$$

holds; otherwise there would be a link from one chain to the beginning of another and the partition would be lost. We now define

$$\begin{aligned} &\mathsf{funct}\ chainmap \equiv (\mathsf{state}\ m : ischainable(m))\ \mathsf{state} : \\ &\quad \bigcup_{z \in \uparrow m} prefix(z, chainset(z{\downarrow}m))\ . \end{aligned}$$

Since the elements of $z{\downarrow}m$ are the starting points of mutually unconnected chains, they are also sources (in the graph-theoretic sense) of the chained map. We set, for arbitrary state n,

$$src(n) \stackrel{\text{def}}{\equiv} {\downarrow}n \setminus {\uparrow}n\ ;$$

this is the set of cells to which no pointer exists in n. With the help of this notion we can characterize chainings of maps by the following predicate *ischaining* :

$$\begin{array}{rcl} ischaining(l) & \stackrel{\text{def}}{\equiv} & \forall\ z_1, z_2 \in src(l) : z_1 \neq z_2 \ \Rightarrow\ disjoint(z_1, z_2, l) \\ & \wedge & l = \bigcup_{z \in src(l)} from(z, l) \end{array}$$

Moreover, we can define the inverse operation to chaining:

$$unchain(l) \stackrel{\text{def}}{\equiv} ischaining(l) \ \rhd \bigcup_{z \in src(l)} [set(sequ(l(z), l)) \ \mapsto\ z] \ .$$

Lemma 8.3

$ischainable(m) \ \Rightarrow\ unchain(chainmap(m)) \ \equiv\ m.$

Proof: Set $l \stackrel{\text{def}}{\equiv} chainmap(m)$. Since $\downarrow m \cap \uparrow m \ \equiv\ \emptyset$ by the assumption, we have $src(l) \ \equiv\ \uparrow m$. Consider $z \in \uparrow m$ and $l_z \stackrel{\text{def}}{\equiv} chain(sequ(z, l))$. Then

$$l^*(l(z)) \ \equiv\ set(sequ(l(z)) \cup \{\Box\} \ \equiv\ z{\downarrow}m \cup \{\Box\}$$

and thus $l^*(l(z)) \cap \downarrow l_u \ \equiv\ \emptyset$ for all $u \in \uparrow m \backslash \{z\}$. Now we calculate

$$\begin{array}{ll} & unchain(l) \\ \equiv & \bigcup_{z \in \uparrow m} [set(sequ(l(z), l)) \ \mapsto\ z] \\ \equiv & \quad \text{(by Lemma 4.4, since } l \ \equiv \bigcup_{u \in \uparrow m} l_u) \\ & \bigcup_{z \in \uparrow m} [set(sequ(l(z), l_z)) \ \mapsto\ z] \\ \equiv & \quad \text{(by Corollary 8.2(2))} \\ & \bigcup_{z \in \uparrow m} [z{\downarrow}m \ \mapsto\ z] \\ \equiv & \quad \text{(by Lemma 8.1(2))} \\ & m \ . \end{array}$$

■

As in the case of set representations, we want to develop an incrementation function for extending a chained representation of a map into one of a larger map. Let therefore l be a map satisfying $ischaining(l)$ and $unchain(l) \ \equiv\ m$, and let x, y be cells such that $x \notin \downarrow m \cup \{\Box\}$ and $ischainable(n)$ holds for $n \stackrel{\text{def}}{\equiv} m \cup [x \ \mapsto\ y]$. First we calculate

$$\begin{array}{ll} & ischainable(n) \\ \Leftrightarrow & \downarrow(m \cup [x \ \mapsto\ y]) \cap \uparrow(m \cup [x \ \mapsto\ y]) \ \equiv\ \emptyset \\ \Leftrightarrow & (\downarrow m \cup \{x\}) \cap (\uparrow m \cup \{y\}) \ \equiv\ \emptyset \\ \Leftrightarrow & (\downarrow m \cap \uparrow m) \cup (\downarrow m \cap \{y\}) \cup (\{x\} \cap \uparrow m) \cup (\{x\} \cap \{y\}) \ \equiv\ \emptyset \\ \Leftrightarrow & \downarrow m \cap \uparrow m \ \equiv\ \emptyset \ \wedge\ \downarrow m \cap \{y\} \ \equiv\ \emptyset \ \wedge\ \{x\} \cap \uparrow m \ \equiv\ \emptyset \ \wedge\ \{x\} \cap \{y\} \ \equiv\ \emptyset \\ \Leftrightarrow & ischainable(m) \ \wedge\ y \notin \downarrow m \ \wedge\ x \notin \uparrow m \ \wedge\ x \not\equiv y \\ \Leftrightarrow & y \notin \downarrow m \ \wedge\ x \notin \uparrow m \ \wedge\ x \not\equiv y \\ \Leftrightarrow & y \notin set(l) \backslash src(l) \ \wedge\ x \notin src(l) \ \wedge\ x \not\equiv y \ . \end{array}$$

According to the definition of *chainmap* we have $l \equiv \bigcup_{z \in \uparrow m} l_z$ for certain chains l_z. To achieve a more uniform calculation we set $l_u \stackrel{\text{def}}{\equiv} \emptyset$ if $u \notin \uparrow m$. Moreover, we define

$$l(\!(y)\!) \stackrel{\text{def}}{\equiv} \mathsf{if}\ y \in \downarrow l\ \mathsf{then}\ l(y)\ \mathsf{else}\ \Box\ \mathsf{fi}\ .$$

Then for each z the pair $rest(l_z) \stackrel{\text{def}}{\equiv} (l(\!(z)\!), l_z \ominus z)$ represents $z{\downarrow}m$ and hence $chainset(z{\downarrow}m) \sqsupseteq_D rest(l_z)$. We have

$$\downarrow l \equiv \bigcup_{z \in \uparrow m} (\{z\} \cup z{\downarrow}m)\ ,$$

and hence $x \notin \downarrow l$. Consider now a $z \in \uparrow n$.

Case 1: $z \not\equiv y$. Then $z{\downarrow}n \equiv z{\downarrow}m$, and hence $prefix(z, chainset(z{\downarrow}n)) \sqsupseteq_D l_z$.
Case 2: $z \equiv y$. Then

$$\begin{array}{ll} & prefix(z, chainset(z{\downarrow}n)) \\ \equiv & prefix(z, chainset(z{\downarrow}m \cup \{x\})) \\ \sqsupseteq_D & prefix(z, add(x, chainset(z{\downarrow}m))) \\ \sqsupseteq_D & prefix(z, add(x, rest(l_z)))\ . \end{array}$$

Hence

$$\begin{array}{ll} & chain(n) \\ \equiv & \bigcup_{z \in \uparrow n} prefix(z, chainset(z{\downarrow}n)) \\ \equiv & \bigcup_{z \in \uparrow m \cup \{y\}} prefix(z, chainset(z{\downarrow}n)) \\ \equiv & \bigcup_{z \in \uparrow m \setminus \{y\} \cup \{y\}} prefix(z, chainset(z{\downarrow}n)) \\ \equiv & prefix(y, chainset(y{\downarrow}n)) \cup \bigcup_{z \in \uparrow m \setminus \{y\}} prefix(z, chainset(z{\downarrow}n)) \\ \sqsupseteq_D & prefix(y, add(x, rest(l_y))) \cup \bigcup_{z \in \uparrow m \setminus \{y\}} l_z \\ \equiv & prefix(y, add(x, rest(l_y))) \cup l \setminus l_y \\ \equiv & prefix(y, add(x, rest(l_y))) \cup l \ominus {\downarrow} l_y \\ \equiv & \quad (\text{since } x \notin \downarrow l) \\ & prefix(y, add(x, rest(l_y))) \cup l \ominus {\downarrow} prefix(y, add(x, rest(l_y))) \\ \equiv & l \lhd prefix(y, add(x, rest(l_y))) \\ \equiv & l \lhd ([y \mapsto x] \cup [x \mapsto l(\!(y)\!)] \cup l_y \ominus y) \\ \equiv & \quad (\text{by Lemma 3.8 (Annihilation), since } l_y \ominus y \subseteq l) \\ & l \lhd ([y \mapsto x] \cup [x \mapsto l(\!(y)\!)])\ . \end{array}$$

Hence we define

$$\begin{array}{ll} & insert(l, y, x) \\ \stackrel{\text{def}}{\equiv} & ischaining(l) \wedge x \notin set(l) \wedge y \notin set(l) \setminus src(l) \wedge x \neq y \rhd \\ & l \lhd ([y \mapsto x] \cup [x \mapsto l(\!(y)\!)])\ . \end{array}$$

The results of the above development then are summarized by

Lemma 8.4

Assume $ischaining(l) \ \wedge \ x \notin set(l) \ \wedge \ y \notin set(l) \backslash src(l) \ \wedge \ x \neq y$.
Then $chainmap(unchain(l) \cup [x \ \mapsto \ y]) \ \sqsupseteq_D \ insert(l, y, x)$.

8.3.3 Pointer Relocation Completed

We are now in the position to describe our efficient algorithm for computing $k \circ m$: We first construct a chained representation $l \ \equiv \ \bigcup_{z \in \uparrow m} l_z$ of m. Now we define

$$relocate(l, k) \ \stackrel{\text{def}}{\equiv} \ ischaining(l) \ \rhd \ k \circ unchain(l) \ .$$

We have

$$\begin{array}{ll} & relocate(l, k) \\ \equiv & k \circ \bigcup_{z \in src(l)} [set(sequ(l(z), l_z)) \ \mapsto \ z] \\ \equiv & \bigcup_{z \in src(l)} [set(sequ(l(z), l_z)) \ \mapsto \ k(z)] \ . \end{array}$$

This is the main loop of our algorithm. We now want to develop a more direct version of the inner loops that form the maps $[set(sequ(l(z), l_z)) \ \mapsto \ k(z)]$ for $z \in src(l)$. To this end we define

$$fibre(l, x, y) \ \stackrel{\text{def}}{\equiv} \ isanchored(x, l) \ \wedge \ x \in \downarrow l \ \rhd \ [set(sequ(l(x), l)) \ \mapsto \ y] \ .$$

We have

$$\begin{array}{ll} & fibre(l, x, y) \\ \equiv & \mathsf{if} \ l(x) = \square \ \mathsf{then} \ [\emptyset \ \mapsto \ y] \\ & \qquad \mathsf{else} \ [\{l(x)\} \cup set(sequ(l(l(x)), l)) \ \mapsto \ y] \ \mathsf{fi} \\ \equiv & \mathsf{if} \ l(x) = \square \ \mathsf{then} \ \emptyset \\ & \qquad \mathsf{else} \ [l(x) \ \mapsto \ y] \cup [set(sequ(l(l(x)), l)) \ \mapsto \ y] \ \mathsf{fi} \\ \equiv & \mathsf{if} \ l(x) = \square \ \mathsf{then} \ \emptyset \\ & \qquad \mathsf{else} \ [l(x) \ \mapsto \ y] \cup fibre(l, l(x), y) \ \mathsf{fi} \ . \end{array}$$

Hence we have the recursion (termination is obvious)

$$\begin{array}{l} \mathsf{funct} \ fibre \ \equiv \ (\mathsf{state} \ l, \mathsf{cell} \ x, y : isanchored(x, l) \ \wedge \ x \in \downarrow l) \ \mathsf{state} : \\ \quad \lceil \ \mathsf{cell} \ z \ \equiv \ l(x) \ ; \mathsf{if} \ z = \square \ \mathsf{then} \ \emptyset \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else} \ [z \ \mapsto \ y] \cup fibre(l, z, y) \ \mathsf{fi} \ \rfloor \ . \end{array}$$

Finally, we obtain

$$\begin{array}{ll} & relocate(l, k) \\ \equiv & \bigcup_{x \in src(l)} [set(sequ(l(x), l_x)) \ \mapsto \ k(x)] \\ \equiv & \bigcup_{x \in src(l)} fibre(l, x, k(x)) \ . \end{array}$$

8.4 Combining Relocation and Copying

Suppose that $unchain(l) \ \equiv \ m$. Then

$$
\begin{array}{ll}
& k \circ m \circ k^{-1} \\
\equiv & k \circ unchain(l) \circ k^{-1} \\
\equiv & relocate(l,k) \circ k^{-1} \\
\equiv & copass(relocate(l,k),k) \ .
\end{array}
$$

Therefore we define

$$
\begin{array}{rcl}
copy(l,k) & \stackrel{\text{def}}{\equiv} & ischaining(l) \ \wedge \ isinjective(k) \ \wedge \ \downarrow k = set(unchain(l)) \rhd \\
& & copass(relocate(l,k),k) \ .
\end{array}
$$

Then the following diagram commutes:

$$
\begin{array}{ccc}
set(m) & \stackrel{unchain(l)}{\longrightarrow} & set(m) \\
\Big\downarrow k & & \Big\downarrow k \\
set(m)\!\uparrow\! k & \stackrel{copy(l,k)}{\longrightarrow} & set(m)\!\uparrow\! k
\end{array}
$$

This concludes our treatment of the pointer relocation pass.

9 A Survey of the Further Development

9.1 Compressing Chained Graph States

The copying algorithm of the previous section can be specialized to the following problem: Given a graph G and an overlap-free and order-preserving (but not necessarily gap-free) allocation g, compute from

$$n \stackrel{\text{def}}{\equiv} blockrep(G,g)$$

the compressed state

$$n_c \stackrel{\text{def}}{\equiv} blockrep(G,g_c)$$

for the unique compressing allocation g_c of G.

In Section 7.4 we have already seen that n_c is the copy of n via the collapsing map k; i.e., $n_c \ \equiv \ k \circ n \circ k^{-1}$, where k now is defined by

$$\text{(K)} \qquad k(g(x)+i) \stackrel{\text{def}}{\equiv} g_c(x)+i$$

for $x \in \downarrow G$ and $i \in [0 : |G(x)|]$.

A central assumption for chainable maps was that their domains should be disjoint from their ranges. Since, however, pseudo-graph states do not have this property, we cannot chain whole substates, but only their *arcs*, where

$$arcs(m) \stackrel{\text{def}}{\equiv} m \ominus keys(m) \ .$$

Assuming now that we have a map l such that

$$m \stackrel{\text{def}}{\equiv} l\,|\bigcup_{z \in keys(n)} set(sequ(z,l))$$

is a chained representation of $arcs(n)$, we have the decomposition

$$
\begin{array}{ll}
 & n \\
\equiv & n_\Box \cup arcs(n) \\
\equiv & n_\Box \cup unchain(m)
\end{array}
$$

where

$$n_\Box \stackrel{\text{def}}{\equiv} n|keys(n) \ (\equiv \ n \backslash arcs(n)) \ .$$

Now we can employ the functions derived in Chapter 8 to calculate

$$
\begin{array}{ll}
 & n_c \\
\equiv & k \circ n \circ k^{-1} \\
\equiv & k \circ (n_\Box \cup unchain(m)) \circ k^{-1} \\
\equiv & (k \circ n_\Box \circ k^{-1}) \cup (k \circ unchain(m) \circ k^{-1}) \\
\equiv & \quad \text{(since } k(\Box) \ \equiv \ \Box) \\
 & (n_\Box \circ k^{-1}) \cup copass(relocate(m,k),k) \\
\equiv & copass(n_\Box, k) \cup copy(m,k) \ .
\end{array}
$$

By the definition of *copass* we have

$$
\begin{array}{ll}
 & copass(n_\Box, k) \\
\equiv & \bigcup\limits_{z \in \downarrow n_\Box} [k(z) \ \mapsto \ n_\Box(z)] \\
\equiv & \bigcup\limits_{z \in keys(n)} [k(z) \ \mapsto \ \Box] \ .
\end{array}
$$

Furthermore we define

$$
\begin{array}{l}
p \stackrel{\text{def}}{\equiv} \ relocate(m,k) \ \equiv \ k \circ arcs(n) \ , \\
size(z) \stackrel{\text{def}}{\equiv} \ succ_{\mathsf{cell} \backslash \downarrow arcs(n)}(z) - z \ .
\end{array}
$$

Hence $size(z)$ gives the number of cells in the block headed by z. Using that

$$
\begin{array}{ll}
 & \downarrow p \\
\equiv & \downarrow arcs(n) \\
\equiv & \bigcup\limits_{z \in keys(n)} [z+1 : z + size(z)]
\end{array}
$$

we obtain

$$
\begin{array}{ll}
 & copass(p,k) \\
\equiv & \bigcup\limits_{x \in \downarrow p} [k(x) \ \mapsto \ p(x)] \\
\equiv & \bigcup\limits_{z \in keys(n)} \ \bigcup\limits_{i \in [1:size(z)]} [k(z+i) \ \mapsto \ p(z+i)] \ .
\end{array}
$$

Therefore

$$
\begin{array}{ll}
 & n_c \\
\equiv & copass(n_\Box, k) \cup copass(p,k) \\
\equiv & \bigcup\limits_{z \in keys(n)} ([k(z) \ \mapsto \ \Box] \cup (\bigcup\limits_{i \in [1:size(z)]} [k(z)+i \ \mapsto \ p(z+i)]))
\end{array}
$$

since $k(z+i) \ \equiv \ k(z) + i$ for $i \in [1 : size(z)]$.

Thus our problem divides into two parts:

1. Compute $p \equiv \mathit{relocate}(m,k)$ from l.
2. Compute the union above.

By the definition of relocate (cf. Section 8.3.3) it is immediate that

$$\mathit{relocate}(m,k) \equiv \bigcup_{z \in \mathit{keys}(n)} \mathit{fibre}(m,z,k(z))$$

where

$$\mathit{fibre}(m,z,y) \equiv [\mathit{set}(\mathit{sequ}(m(z),m)) \mapsto y] ,$$

because $\mathit{src}(m) \equiv \mathit{keys}(n)$. From this specification we develop in [Berger et al. 89] a loop that traverses the set $\mathit{keys}(n)$ in ascending order.

The body of the function *copass* is, like the body of *relocate*, a union over the index set $\mathit{keys}(n)$. Thus, forming *copass* requires a second traversal of $\mathit{keys}(n)$. However, the successor function on $\mathit{keys}(n)$ can be *computed* (as a map) simultaneously with *relocate* and overwritten onto the state; afterwards we can use it to traverse $\mathit{keys}(n)$, thus improving speed efficiency considerably.

The relocation pass thus returns a union lp of the state $p \stackrel{\text{def}}{\equiv} \mathit{relocate}(m,k)$ and the map l which is a chained representation of $\mathit{keys}(n)$. Now we have to construct n_c from this union based on the values $\mathit{size}(z)$ for $z \in \mathit{keys}(n)$. As stated before, n_c is represented by

$$n_c \equiv \bigcup_{z \in \mathit{keys}(n)} ([k(z) \mapsto \square] \cup \bigcup_{i \in [1:\mathit{size}(z)]} [k(z)+i \mapsto p(z+i)]) ,$$

where now lp can be substituted for p. The outer union represents the main loop of the algorithm. In evaluating it we only need one value of k at a time; the value for the next cycle can be computed using the recursion relation (cf. Section 7.2)

$$k(\mathit{next}_n(y)) \equiv k(y) + \mathit{size}(y)$$

for $y \in \mathit{keys}(n) \backslash \{\mathit{max}(\mathit{keys}(n))\}$. Hence we can eliminate k and use its values on the single cells instead. So no extra space for k is necessary.

9.2 Determining the Reachable Subgraph

We now turn to the problem of determining the reachable part of the store. We prepare this step at the level of pseudo-graphs. Let pgraph be the set of pseudo-graphs and nodeset be the set of finite subsets of node. The specification of the reachability problem now reads

$$\begin{array}{l}\mathsf{funct}\ \mathit{reach} \equiv (\mathsf{pgraph}\ G, \mathsf{nodeset}\ s : s \subseteq {\downarrow}G)\mathsf{pgraph} : \\ \quad G|\mathit{rnset}_G(s) .\end{array}$$

From this specification the following algorithm is derived in [Berger et al. 89]:

$$\begin{array}{l}\mathsf{funct}\ \mathit{reach} \equiv (\mathsf{pgraph}\ G, \mathsf{nodeset}\ s : s \subseteq {\downarrow}G)\mathsf{pgraph} : \\ \quad \mathsf{if}\ s = \emptyset\ \mathsf{then}\ \emptyset \\ \qquad \mathsf{else}\ \ \mathsf{node}\ z \equiv \mathit{elem}(s) ; \\ \qquad\quad \mathsf{pgraph}\ G_1 \equiv G \ominus z ; \\ \qquad\quad \mathsf{nodeset}\ s_1 \equiv (s \cup \mathit{set}(G(z))) \cap {\downarrow}G_1 ; \\ \qquad\quad [z \mapsto G(z)] \cup \mathit{reach}(G_1, s_1) \qquad\qquad \mathsf{fi} ,\end{array}$$

where

$$elem(s) \stackrel{\text{def}}{\equiv} s \neq \emptyset \rhd \text{some node } x : x \in s \ .$$

Termination is guaranteed, since $|{\downarrow}G_1| < |{\downarrow}G|$.

9.3 Merging Reachability and Chaining

In Section 9.1 we have described an efficient compressing algorithm based on a chaining of the arcs of the state to be copied. We now discuss the integration of the construction of such a representation with the computation of the reachable part.

The first step consists in transforming the reachability algorithm on pseudo-graphs into a corresponding algorithm for their state representations. An additional requirement for this algorithm is that it should not use a separate parameter for the set t of cells already visited (corresponding to the parameter s of *reach*), since this would occupy a lot of storage space. Everything should be done on the store and on some auxiliary cells.

A first idea how to realize this would be to represent the set t as a chain and to overwrite the store with this chain. However, one sees immediately that this is in conflict with the chaining for representing the arcs which requires a different overwriting of the store. These difficulties are overcome by not storing t itself but (a code of) a set of cells pointing to the elements of t and, when adding a piece $block(x, g)$, by chaining the cells not in their original order but in a different one arising during the traversal of the reachable part. In fact, we use a chaining of the heads of the block fragments that have not been visited yet. This chaining arises from a specialization of the reachability algorithm above: The set s is represented by a sequence, and *elem* is refined to *first*. In this way we obtain a depth-first traversal of the storage structure in which the representation of s acts as a stack. This stack is then compressed considerably by retaining only the leading cells of the unprocessed block fragments.

Next we add a parameter that accumulates a chaining of the map that is the already visited storage part. The resulting algorithm still uses the three "large" parameters, one for the not yet visited part of the store, one for the stack, and one for the chaining of the visited part of the store. In the case of garbage collection, however, there is no space available for three separate parameters. So we need to represent them by *one* map parameter, viz. by the overall store. This is possible since the three original parameters are maps with pairwise disjoint domains which can be united into a single one. The complete algorithm (cf. [Berger et al. 89]) is linear in the size of the store in which garbage collection takes place.

10 Conclusion

We have shown with several 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. [Mason 88, Kausche 89]) 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. Using this algebra one saves an

enormous amount of quantifiers (as compared e.g. with [Bijlsma 88]). Moreover, the operations of that algebra are expressive enough that we did not need to explain anything with the help of diagrams. Even when developing the intricate garbage collection algorithm described above 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 [Suzuki 80].

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

Acknowledgement
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 helpful remarks from F.L. Bauer, U. Berger, R. Berghammer, R. Dewar, W. Dosch, F. Erhard, M. Lichtmannegger, W. Meixner, H. Partsch, P. Pepper, M. Sintzoff, and, particularly, H. Ehler. C. Karpf has pointed out a significant simplification in the derivation of the concatenation algorithm.

11 References

[Bauer, Wössner 82]
F.L. Bauer, H. Wössner: Algorithmic language and program development. New York: Springer 1982

[Bauer et al. 85]
F.L. Bauer et al.: The Munich project CIP. Volume I: The wide spectrum language CIP-L. Lecture Notes in Computer Science **183**. New York: Springer 1985

[Bauer et al. 89]
F.L. Bauer, B. Möller, H. Partsch, P. Pepper: Formal program construction by transformations — Computer-aided, Intuition-guided Programming. IEEE Transactions on Software Engineering **15**, 165–180 (1989)

[Berger et al. 89]
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

[Bijlsma 88]
A. Bijlsma: Calculating with pointers. Science of Computer Programming **12**, 191–205 (1988)

[Broy, Pepper 82]
M. Broy, P. Pepper: Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite-Algorithm. ACM TOPLAS **4**, 362–381 (1982)

[Burstall 72]
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

[Dewar, McCann 77]
R. Dewar, A. McCann: MACRO SPITBOL — a SNOBOL4 compiler. Software — Practice and Experience **7**, 95–113 (1977)

[Dewar et al. 82]
R. Dewar, M. Sharir, E. Weixelbaum: Transformational derivation of a garbage collection algorithm. ACM TOPLAS **4**, 650–667 (1982)

[van Diepen, de Roever 86]
N. van Diepen, W. de Roever: Program derivation through transformations: The evolution of list-copying algorithms. Science of Computer Programming **6**, 213–272 (1986)

[Jones 80]
C.B. Jones: Software development: A rigorous approach. Englewood Cliffs: Prentice-Hall 1980

[Kausche 89]
A. Kausche: Modale Logiken von geflechtartigen Datenstrukturen und ihre Kombination mit temporaler Programmlogik. Fakultät für Mathematik und Informatik der TU München, Dissertation, 1989

[Levy 78]
M. Levy: Verification of programs with data referencing. Proc. 3me Colloque sur la Programmation 1978, 413–426

[Manna 74]
Z. Manna: Mathematical theory of computation. New York: McGraw-Hill 1974

[Mason 88]
I. Mason: Verification of programs that destructively manipulate data. Science of Computer Programming **10**, 177–210 (1988)

[Möller 89]
B. Möller: Applicative assertions. In : J.L.A. van de Snepscheut (ed.): Mathematics of Program Construction, Groningen, 26–30 June 1989. Lecture Notes in Computer Science **375**. Berlin: Springer 1989, 348–362

[Pepper, Möller 89]
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

[Reynolds 79]
J. Reynolds: Reasoning about arrays. Commun. ACM **22**, 290–299 (1979)

[Suzuki 80]
N. Suzuki: Analysis of pointer rotation. Conf. Record 7th POPL, 1980, 1–11. Revised version: Commun. ACM **25**, 330–335 (1982)