TOWARDS POINTER ALGEBRA

Bernhard Möller

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

Abstract. We show that the well-known unfold/fold transformation strategy is also fruitful for the (formal) derivation of correct pointer algorithms. The key to this extension is an 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. Due to the implicit connections through paths within a pointer structure, the side effects of a pointer assignment are usually much harder to survey than those of an assignment to a simple variable. 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 map from addresses to certain values. A part of such a state that forms a logical unit can then be represented by a partial submap of that map. This gives the possibility of describing the state in a modularised way as the union of the submaps for its logical subunits. In the case of pointer structures this means that the usual "spaghetti" structure of the entire 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 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. This also makes it possible to stay, almost to the very end of the derivations, at the applicative level with its rich algebraic properties. Also, we can stay with equational reasoning rather than with more general inference rules as would be necessary for the special "pointer logics" introduced in [5,9,12]. Finally, our "pointer algebra" saves a lot of quantifiers as compared e.g. with [4].

We illustrate the approach with derivations of two algorithms on singly linked lists, viz. concatenation and reversal "in situ". However, the method 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.

2 Partial Maps and Overwriting

The use of algebraic operations on maps for describing the effect of a program dates back at least to [18]. Two of the most useful operations in our setting are map union (see also [17] for other applications) and map overwriting. Unfortunately, union is a partial operation on maps: if two maps disagree at a certain argument, their union is a proper relation and no longer a map. To simplify the algebraic treatment we therefore embed the algebra of maps into the algebra of binary relations [21]. In this section we present the definitions and the essential properties of our operations on relations and maps; a systematic development including further properties and full proofs is given in the appendix. There is also a more general algebraic framework accomodating these notions (see [13–16,19] for its use); however, a lot of the definitions there are not necessary when dealing with pointer structures alone, and so we do not want to burden the reader with them.

A **relation** R between two sets M and N is a subset of the cartesian product $M \times N$. Some of our notation derives from this set-theoretical view of relations. E.g., by $\emptyset$ we denote the empty relation between M and N. Moreover, we shall use union and intersection of relations. The **domain** $\mathsf{dom}\, R$ of a relation R is given by

$$\mathsf{dom}\, R \stackrel{\text{def}}{=} \{x \mid \exists\, y : (x, y) \in R\} \ .$$

Another essential operation on relations is that of **overwriting** one relation with another one (see e.g. [6, 7, 8] for the special case of maps): Given relations R, S between sets M and N, we define

$$R \,|\, S \stackrel{\text{def}}{=} R \cup \{(x, y) \mid (x, y) \in S \wedge x \notin \mathsf{dom}\, R\} \ .$$

In other words, $R \,|\, S$ results from S by changing the images of the elements of M according to the prescription of R (if any). For example, if S is a map then $\{(x, y)\} \,|\, S$ "updates" S to make y the value corresponding to x. This operation will be our main tool for describing selective updating of storage structures. We use the convention that $|$ binds tighter than all set-theoretic operations.

The following properties allow localising side effects to that part of a store they really affect:

Lemma 2.1 (Localisation)

(1) $\mathsf{dom}\, R \cap \mathsf{dom}\, S = \emptyset \quad \Rightarrow \quad R \,|\, (S \cup T) = S \cup R \,|\, T.$
(2) $\mathsf{dom}\, R \cap \mathsf{dom}\, T = \emptyset \quad \Rightarrow \quad (R \cup S) \,|\, T = R \cup S \,|\, T.$

Relations form a monoid under overwriting:

Lemma 2.2 (Monoid)

(1) $\emptyset \,|\, R = R = R \,|\, \emptyset$.
(2) $(R \,|\, S) \,|\, T = R \,|\, (S \,|\, T)$.

The following lemma describes the relation between overwriting and union in more detail.

Lemma 2.3 (Compatibility)

The following properties are equivalent:
(1) $R \,|\, S = R \cup S = S \,|\, R$.
(2) $R \,|\, S = S \,|\, R$.
(3) R and S agree on $\mathsf{dom}\, R \cap \mathsf{dom}\, S$.

If any of these conditions holds we call R and S **compatible**. From this we obtain two useful corollaries:

Corollary 2.4 (Sequentialisation)
$(R \cup S) \,|\, T = R \,|\, (S \,|\, T)$ provided R and S are compatible.

By this property, a complex overwriting may also be done by two successive simpler overwritings.

Corollary 2.5 (Annihilation)
Suppose $S \,|\, T = T$. Then $(R \cup S) \,|\, T = R \,|\, T$ provided R and S are compatible.

This means that an overwriting with values that are already present does not have any effect and thus can be skipped.

Now we consider the special case of maps. A relation $R \subseteq M \times N$ is a **(partial) map** from M to N if it associates every element of M with at most one value, i.e., if

$$\forall\, x \in M : \forall\, y, z \in N : (x, y) \in R \,\wedge\, (x, z) \in R \;\Rightarrow\; y = z \;.$$

We write $R : M \;\rightarrow\; N$ to indicate that R is a map between M and N. Note that $\emptyset$ is a map. Moreover, if R is a map and $S \subseteq R$ then S is a map as well.

In general, relational union does not preserve the map property. However,

Lemma 2.6 (Map Compatibility)
Let $(R_j)_{j \in J}$ be a family of maps. Then $\bigcup_{j \in J} R_j$ is a map iff the R_j are pairwise compatible.

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 modularisation is reflected by the distributive laws that allow propagation of operations to substates of a state.

Maps are closed under overwriting:

Lemma 2.7 (Closure)
Let R, S be maps. Then $R|S$ is a map as well.

For maps, inclusion implies compatibility, which is not necessarily the case for arbitrary relations:

Lemma 2.8 (Submap)
Let S be a map and $R \subseteq S$. Then R and S are compatible.

This gives a kind of idempotence property:

Lemma 2.9 (Idempotence)
Let S be a map and T a relation. Then

$$R \subseteq S \;\;\Rightarrow\;\; S \,|\, T \equiv S \,|\, R \,|\, T \;.$$

When viewed operationally, the direction from left to right allows early overwriting of parts of a map.

Finally, using Lemma 2.8, for maps the annihilation property can be stated more simply:

Corollary 2.10 (Annihilation)
Let S, T be maps such that $S \subseteq T$. If R is a map such that R and S are compatible, then $(R \cup S) \,|\, T = R \,|\, T$. In particular, setting $R = \emptyset$, we obtain $S \,|\, T = T$.

3 Chains

As an example of how to describe pointer structures within the algebra of maps we now study singly linked lists.

3.1 Notation

In notation and semantics we 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. For Boolean expression B we abbreviate $B \equiv \mathsf{true}$ by B.

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 B be a Boolean expression possibly involving the identifier x. Then the declaration

$$\mathsf{funct}\ f \equiv (\mathsf{t}\ x : B)\,\mathsf{u} : E$$

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

$$\mathsf{funct}\ f \equiv (\mathsf{t}\ x)\,\mathsf{u} : \mathsf{if}\ B\ \mathsf{then}\ E\ \mathsf{else}\ \mathsf{error}\ \mathsf{fi}\ .$$

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

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

$$\begin{aligned} E_1\ \mathsf{cand}\ E_2 &\stackrel{\text{def}}{\equiv} \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ \mathsf{false}\ \mathsf{fi}\ , \\ E_1\ \mathsf{cor}\ E_2 &\stackrel{\text{def}}{\equiv} \mathsf{if}\ E_1\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ E_2\ \mathsf{fi}\ . \end{aligned}$$

So they are asymmetric, sequentially evaluated variants of the usual Boolean conjunction and and disjunction or. Our main use of them is in shielding partialities. E.g., the expression

$$x \neq 0\ \mathsf{cand}\ 5/x = 1$$

is defined for all x, whereas

$$x \neq 0\ \mathsf{and}\ 5/x = 1$$

is undefined for $x \equiv 0$, since then $5/x$ is undefined and and is strict.

3.2 A Model of Pointer Structures

A pointer structure consists of a set of records connected by pointers. We abstract from the concrete contents of the records and consider only their interrelationship through the pointers, since this is the only source of problems in pointer algorithms. This is modelled by a binary **access relation** $R \subseteq \mathsf{rcd} \times \mathsf{rcd}$ where rcd is the set of records (represented, say, by their initial addresses). rcd is not required to be finite; this way also (conceptually) unbounded storage can be modelled.

In the case of regular record structures, R will be a union of maps. For instance, in the case of binary trees we would have two maps $\mathit{left}, \mathit{right} : \mathsf{rcd} \to \mathsf{rcd}$ and set $R \stackrel{\text{def}}{\equiv} \mathit{left} \cup \mathit{right}$.

In implementions one frequently uses a (pointer to) a special pseudo-record as a terminator common to all pointer structures considered (e.g., nil in Pascal). Let therefore $\Box \in \mathsf{rcd}$ be a distinguished element, called the **anchor**. The elements of $\mathsf{rcd} \backslash \{\Box\}$ are called **proper** records. A **state** is a finite relation $R \subseteq \mathsf{rcd} \times \mathsf{rcd}$ such that $\Box \notin \mathsf{dom}\, R$. Hence, in a state, $\Box$ is a sink not in relation with any other record. This implies that there can be no $\Box$ record properly within a pointer structure; if present, $\Box$ terminates the structure at that point.

3.3 Definition of Chains

We concentrate now on the special case of **chains**, i.e., of (finite) cycle-free singly linked lists. Since every record in such a list has at most one link, the associated access relation will actually be a (partial) map $\mathit{next} : \mathsf{rcd} \to \mathsf{rcd}$. A single record x with successor y is modeled by the map $[x \mapsto y] \stackrel{\text{def}}{\equiv} \{(x, y)\}$. Partialities in next model records which do not (yet) have a proper successor.

A chain contains a number of records in a certain order prescribed by the links in the list. This induces a sequence structure on these records: The first element in the sequence, if any, is the head record, followed by the others in the order of traversal. Since there is no cycle, the sequence is repetition-free.

The idea of following the pointers in a singly linked list within a state next is captured by considering the reflexive transitive closure next^* of next defined by

$$x\ \mathit{next}^*\ y \stackrel{\text{def}}{\Leftrightarrow} \exists\, n > 0 : \exists\, x_1, \ldots, x_n : x \equiv x_1 \wedge y \equiv x_n \wedge \bigwedge_{j=1}^{n-1} x_{j+1} \equiv \mathit{next}(x_j)\ .$$

Hence $x\ \mathit{next}^*\ y$ iff record y can be reached from record x following the links of next zero or more times. The set of all records reachable from x in next is denoted by

$$\mathit{next}^*(x) \stackrel{\text{def}}{\equiv} \{y \mid x\ \mathit{next}^*\ y\}\ .$$

From the definitions the following recursion equation is immediate:

Corollary 3.1

$$\mathit{next}^*(x) \equiv \{x\} \cup \mathsf{if}\ x \in \mathsf{dom}\ \mathit{next}\ \mathsf{then}\ \mathit{next}^*(\mathit{next}(x))\ \mathsf{else}\ \emptyset\ \mathsf{fi}\ .$$

Since we use the anchor $\Box$ to terminate chains, we call a record x **anchored** in a state next iff $\Box \in \mathit{next}^*(x)$. In particular, $\Box$ is anchored in every state. This is reasonable, since $\Box$ will be the "head" of the empty chain. Moreover,

Corollary 3.2

$\Box \in next^*(x)$ and $x \notin \mathsf{dom}\ next \Leftrightarrow x = \Box$.

Proof: Immediate from the definition. ∎

By the above considerations, anchored chains are in exact correspondence with finite repetition-free sequences of proper records. We denote the type of sequences of records by rcdsequ. For $s \in$ rcdsequ, we mean by $|s|$ the length of s and by $s[i]$ the i-th element of s where numbering starts with 1. If $i > |s|$ or $i = 0$, we set $s[i] \stackrel{\text{def}}{\equiv} \Box$. By $\langle\rangle$ we denote the empty sequence and by $+$ sequence concatenation. A singleton sequence is identified with the only element it contains. Finally,

$$\begin{array}{lcllcl} \mathit{first}(s) & \stackrel{\text{def}}{\equiv} & s[1] , & \mathit{last}(s) & \stackrel{\text{def}}{\equiv} & s[|s|] , \\ \mathit{rest}(s) & \stackrel{\text{def}}{\equiv} & \sum_{i=2}^{|s|} s[i] , & \mathit{lead}(s) & \stackrel{\text{def}}{\equiv} & \sum_{i=1}^{|s|-1} s[i] . \end{array}$$

Note that

$$\begin{array}{lcllcl} \mathit{first}(\langle\rangle) & \equiv & \Box & \equiv & \mathit{last}(\langle\rangle) , \\ \mathit{rest}(\langle\rangle) & \equiv & \langle\rangle & \equiv & \mathit{lead}(\langle\rangle) . \end{array}$$

We define a representation function from finite repetition-free sequences of proper records to anchored chains:

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

Note that

$$\mathit{chain}(\langle\rangle) \equiv \emptyset .$$

Moreover,

$$\mathsf{dom}\ \mathit{chain}(s) \equiv \mathit{set}(s) ,$$

where

$$\mathit{set}(s) \stackrel{\text{def}}{\equiv} \{s[i] : 1 \leq i \leq |s|\}$$

is the set of all records occurring in sequence s.

The precondition *ischainable* is defined inductively as follows:

$$\begin{array}{rcl} \mathit{ischainable}(\langle\rangle) & \equiv & \mathsf{true} , \\ \mathit{ischainable}(x) & \equiv & x \neq \Box , \\ \mathit{ischainable}(s+t) & \equiv & \mathit{ischainable}(s)\ \mathsf{and}\ \mathit{ischainable}(t)\ \mathsf{and} \\ & & \mathit{set}(s) \cap \mathit{set}(t) = \emptyset . \end{array}$$

We have

Lemma 3.3

Assume $s, t \in$ rcdsequ and $x \in$ rcd such that $\mathit{ischainable}(s + x + t)$. Then
(1) $\mathit{chain}(x + t) \equiv [x \mapsto \mathit{first}(t)] \cup \mathit{chain}(t)$.
(2) $\mathit{chain}(s + x + t) \equiv \mathit{chain}(x + t) \mid \mathit{chain}(s + x)$.
(3) $\mathit{chain}(s + x + t) \supseteq [x \mapsto \mathit{first}(t)]$.

Proof: (1) is immediate from the definition.

(2) First we note that for sequences u, v the property $ischainable(u + v)$ implies

$$(*) \quad ischainable(s) \wedge ischainable(t) \ ,$$
$$(**) \quad \mathsf{dom}\ chain(u) \cap \mathsf{dom}\ chain(v) \equiv \emptyset \ ,$$

and hence compatibility of $chain(u)$ and $chain(v)$. Now

$$\begin{array}{ll}
& chain(s + x + t) \\
\equiv & \quad \{\![\text{ definition of } chain \ ]\!\} \\
& \bigcup_{i=1}^{|s+x+t|} [(s + x + t)[i] \mapsto (s + x + t)[i + 1]] \\
\equiv & \quad \{\![\text{ definition of } + \text{ and union }]\!\} \\
& \bigcup_{i=1}^{|s|} [(s + x)[i] \mapsto (s + x)[i + 1]] \ \cup \ [x \mapsto t[1]] \ \cup \ \bigcup_{i=1}^{|t|} [t[i] \mapsto t[i + 1]] \\
\equiv & \quad \{\![\text{ definition of } chain \text{ and } | \ ]\!\} \\
& ([x \mapsto t[1]] \,|\, chain(s + x)) \cup chain(t) \\
\equiv & \quad \{\![\ (*) \text{ and } ischainable(s + x + t), (**), \text{localisation (Lemma 2.1)} \ ]\!\} \\
& [x \mapsto t[1]] \,|\, (chain(s + x) \cup chain(t)) \\
\equiv & \quad \{\![\ (*), \text{compatibility (Lemma 2.3)} \ ]\!\} \\
& [x \mapsto t[1]] \,|\, (chain(t) \,|\, chain(s + x)) \\
\equiv & \quad \{\![\text{ sequentialisation (Corollary 2.4)} \ ]\!\} \\
& ([x \mapsto t[1]] \ \cup \ chain(t)) \,|\, chain(s + x) \\
\equiv & \quad \{\![\text{ by (1)} \ ]\!\} \\
& chain(x + t) \,|\, chain(s + x) \ .
\end{array}$$

(3) is immediate from (1), (2) and the definition of |.

■

Conversely, given a record x and a state *next*, we can retrieve the sequence of records in the sublist starting from x (if any) using

$\mathsf{funct}\ sequ \equiv (\mathsf{rcd}\ x, \mathsf{state}\ next)\ \mathsf{rcdsequ}:$
$\quad \mathsf{if}\ x \notin \mathsf{dom}\ next\ \mathsf{then}\ \Diamond\ \mathsf{else}\ x + sequ(next(x), next)\ \mathsf{fi}\ .$

Thus, for a record x without successor or for $\Box$ we return the empty sequence. Note also that *sequ* will not terminate if the sublist within *next* 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 records. Then a record y can be reached from x following the links of *next* (zero or more times) iff $contains(sequ(x, next), y) \equiv \mathsf{true}$, where

$\mathsf{funct}\ contains \equiv (\mathsf{rcd}\ y, \mathsf{rcdsequ}\ s)\ \mathsf{bool}:$
$\quad \mathsf{if}\ s = \Diamond\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ y = first(s)\ \mathsf{cor}\ contains(rest(s), y)\ \mathsf{fi}\ .$

The functions *sequ* and *chain* are, under suitable restrictions, inverses of each other:

Lemma 3.4
For $x \in \mathsf{rcd}$, $next \in \mathsf{state}$ and $s \in \mathsf{rcdsequ}$,
(1) $\Box \in next^*(x) \;\Rightarrow\; \mathit{first}(\mathit{sequ}(x, next)) \equiv x$;
(2) $\Box \in next^*(x) \;\Rightarrow\; \mathit{chain}(\mathit{sequ}(x, next)) \subseteq next$;
(3) $\mathit{ischainable}(s) \;\Rightarrow\; \mathit{sequ}(\mathit{first}(s), \mathit{chain}(s)) \equiv s$.

Proof: (1) **Case 1**: $x \equiv \Box$. We get

$$\begin{array}{ll} & \mathit{first}(\mathit{sequ}(x, next)) \\ \equiv & \quad \{\![\text{ definition, } next \text{ state }]\!\} \\ & \mathit{first}(\Diamond) \\ \equiv & \quad \{\![\text{ definition }]\!\} \\ & \Box \ . \end{array}$$

Case 2: $x \not\equiv \Box$. Then $\Box \in next^*(x)$ implies that $x \in \mathsf{dom}\, next$ and that the recursive call $\mathit{sequ}(next(x), next)$ terminates. Hence

$$\begin{array}{ll} & \mathit{first}(\mathit{sequ}(x, next)) \\ \equiv & \quad \{\![\text{ definition }]\!\} \\ & \mathit{first}(x + \mathit{sequ}(next(x), next)) \\ \equiv & \quad \{\![\text{ definition }]\!\} \\ & x \ . \end{array}$$

(2) is a straightforward induction on the length of $\mathit{sequ}(x, next)$.
(3) is a straightforward induction on the length of s.

■

A fundamental property is that overwriting outside of a chain does not change the chain:

Lemma 3.5
For $x, u, v \in \mathsf{rcd}$ and $next \in \mathsf{state}$ such that $\neg\ x\ next^*\ u$,

$$\begin{array}{lrcl} (1) & \mathit{sequ}(x, [u \mapsto v] \mid next) & \equiv & \mathit{sequ}(x, next) \ . \\ (2) & ([u \mapsto v] \mid next)^*(x) & \equiv & next^*(x) \ . \end{array}$$

Proof: (1) We use the well-known technique of computational induction (see e.g. [11]) 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 continuous predicate

$$\begin{array}{rcl} P[f] & \stackrel{\text{def}}{\Leftrightarrow} & \forall\ next : \forall\ x : \forall\ u : \forall\ v : \\ & & \quad \neg\ x\ next^*\ u \;\Rightarrow\; f(x, next) \equiv f(x, [u \mapsto v] \mid next) \ . \end{array}$$

The induction base $P[\Omega]$ is trivial. For the induction step assume that $P[f]$ and $\neg\ x\ next^*\ u$ hold. The functional belonging to the definition of *sequ* is

$$\begin{array}{rcl} \tau\ :\ f & \mapsto & (\mathsf{rcd}\ x, \mathsf{state}\ next)\ \mathsf{rcdsequ}\ : \\ & & \quad \mathsf{if}\ x \notin \mathsf{dom}\, next\ \mathsf{then}\ \Diamond\ \mathsf{else}\ x + f(next(x), next)\ \mathsf{fi}\ . \end{array}$$

First we note that from the definitions it is straightforward that

$$(*) \quad \neg\, x\ next^*\ u \ \Rightarrow\ u \not\equiv x\ ,$$
$$(**) \quad x \in \mathsf{dom}\, next \,\mathsf{and}\, \neg\, x\ next^*\ u \ \Rightarrow\ \neg\ next(x)\ next^*\ u\ .$$

Now we calculate, assuming $\neg\, x\ next^*\ u$,

$$\begin{array}{ll} & \tau[f](x, next) \\ \equiv & \mathsf{if}\ x \notin \mathsf{dom}\, next\, \mathsf{then} \Diamond\, \mathsf{else}\, x + f(next(x), next)\ \mathsf{fi} \\ \equiv & \quad \{\![\text{ by } (**) \text{ and induction hypothesis } P[f] \ ]\!\} \\ & \mathsf{if}\ x \notin \mathsf{dom}\, next\, \mathsf{then} \Diamond\, \mathsf{else}\, x + f(next(x), [u \mapsto v] \,|\, next)\ \mathsf{fi} \\ \equiv & \quad \{\![\text{ by } (*) \ ]\!\} \\ & \mathsf{if}\ x \notin \mathsf{dom}\, ([u \mapsto v] \,|\, next) \\ & \quad \mathsf{then} \Diamond \\ & \quad \mathsf{else}\ \ x + f(([u \mapsto v] \,|\, next)(x), [u \mapsto v] \,|\, next)\ \mathsf{fi} \\ \equiv & \tau[f](x, [u \mapsto v] \,|\, next)\ . \end{array}$$

(2) is proved analogously using the recursion from Corollary 3.1.

■

4 Concatenation of Chains "in Situ"

We now want to specify and develop an algorithm for concatenating two non-overlapping anchored chains "in situ".

4.1 Specification and First Explicit Solution

First we give the precondition for our desired function:

$$\mathsf{funct}\ disjoint \equiv (\mathsf{rcd}\ x, \mathsf{rcd}\ y, \mathsf{state}\ next)\, \mathsf{bool}: \\ next^*(x) \cap next^*(y) = \{\Box\}\ .$$

So we consider a state *next* in which the sublists starting from the records x and y are anchored chains whose sets of proper records 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 records; moreover, the order of traversal within the sublists should be preserved, and all records from the sublist of x should precede all records in the sublist of y. This can be specified by

$$\mathsf{funct}\ conc \equiv (\mathsf{rcd}\ x, \mathsf{rcd}\ y, \mathsf{state}\ next : disjoint(x, y, next))\, \mathsf{state}: \\ chain(sequ(x, next) + sequ(y, next)) \,|\, next\ .$$

So the proper records of the subchains are collected in the right order, the resulting sequence is chained, and this chain is overwritten onto *next* while *re-using the same records*. Hence, no copying is involved and we really are specifying concatenation "in situ". Note that $disjoint(x, y, next)$ implies $ischainable(sequ(x, next) + sequ(y, next))$.

From this specification we now want to develop a direct recursion without the "detour" through sequences. We try to obtain it by the familiar unfold/fold technique. We consider the following cases:

Case 1: $x \notin \mathsf{dom}\, next$. Then $sequ(x, next) \equiv \Diamond$, and hence

$$
\begin{array}{ll}
& chain(sequ(x, next) + sequ(y, next)) \mid next \\
\equiv & \quad \{\![\text{ neutrality of } \Diamond \ ]\!\} \\
& chain(sequ(y, next)) \mid next \\
\equiv & \quad \{\![\text{ by annihilation, since } chain(sequ(y, next)) \subseteq next \text{ by Lemma 3.4(2) }]\!\} \\
& next \ .
\end{array}
$$

Case 2: $x \in \mathsf{dom}\, next$. We define an auxiliary function

$$
\begin{array}{l}
\mathsf{funct}\ neconc \equiv (\mathsf{rcd}\ x, \mathsf{rcd}\ y, \mathsf{state}\ next : disjoint(x, y, next)\ \mathsf{and}\ x \in \mathsf{dom}\, next)\ \mathsf{state} : \\
\quad chain(sequ(x, next) + sequ(y, next)) \mid next \ .
\end{array}
$$

We calculate:

$$
\begin{array}{ll}
& chain(sequ(x, next) + sequ(y, next)) \mid next \\
\equiv & \quad \{\![\text{ definition of } sequ \ ]\!\} \\
& chain(x + sequ(next(x), next) + sequ(y, next)) \mid next \\
\equiv & \quad \{\![\text{ abbreviation }]\!\} \\
& (*) \ .
\end{array}
$$

Now we perform a case analysis on $next(x)$.

Case 2.1: $next(x) \notin \mathsf{dom}\, next$. Then $sequ(next(x), next) \equiv \Diamond$, and hence

$$
\begin{array}{ll}
& (*) \\
\equiv & \quad \{\![\text{ by Lemma 3.3(1) and Lemma 3.4(1) }]\!\} \\
& ([x \mapsto y] \cup chain(sequ(y, next))) \mid next \\
\equiv & \quad \{\![\text{ annihilation, since } chain(sequ(y, next)) \subseteq next \ ]\!\} \\
& [x \mapsto y] \mid next \ .
\end{array}
$$

Case 2.2: $next(x) \in \mathsf{dom}\, next$. Then

$$
\begin{array}{ll}
& (*) \\
\equiv & \quad \{\![\text{ by Lemma 3.3(1) and Lemma 3.4(1) }]\!\} \\
& ([x \mapsto next(x)] \cup chain(sequ(next(x), next) + sequ(y, next))) \mid next \\
\equiv & \quad \{\![\text{ annihilation, since } [x \mapsto next(x)] \subseteq next \ ]\!\} \\
& chain(sequ(next(x), next) + sequ(y, next)) \mid next \\
\equiv & \quad \{\![\text{ fold } neconc \ ]\!\} \\
& neconc(next(x), y, next) \ .
\end{array}
$$

For the correctness of the folding step we also need to check the validity of the assertion of *neconc* for the recursive call. We calculate, assuming $x \in \mathsf{dom}\, next \ \ \mathsf{cand} \ \ next(x) \in \mathsf{dom}\, next$:

$$
\begin{array}{ll}
& disjoint(x, y, next) \\
\equiv & \quad \{\![\text{ definition of } disjoint \ ]\!\} \\
& next^*(x) \cap next^*(y) = \{\Box\} \\
\equiv & \quad \{\![\text{ by Corollary 3.1 }]\!\} \\
& (\{x\} \cup next^*(next(x))) \cap next^*(y) = \{\Box\}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\![\text{ distributivity }]\!\} \\
& (\{x\} \cap \mathit{next}^*(y)) \cup (\mathit{next}^*(\mathit{next}(x)) \cap \mathit{next}^*(y)) = \{\Box\} \\
\equiv & \{\![\; x \neq \Box, \{x\} \cap \mathit{next}^*(y) \subseteq \{x\} \;]\!\} \\
& \{x\} \cap \mathit{next}^*(y) = \emptyset \wedge \mathit{next}^*(\mathit{next}(x)) \cap \mathit{next}^*(y) = \{\Box\} \\
\equiv & \{\![\text{ definition of } \mathit{disjoint} \;]\!\} \\
& \{x\} \cap \mathit{next}^*(y) = \emptyset \wedge \mathit{disjoint}(\mathit{next}(x), y, \mathit{next}) \; .
\end{array}
$$

Thus,

$$
\begin{array}{l}
(\mathit{disjoint}(x, y, \mathit{next}) \;\mathsf{and}\; x \in \mathsf{dom}\; \mathit{next}) \;\mathsf{cand}\; \mathit{next}(x) \in \mathsf{dom}\; \mathit{next} \;\; \Rightarrow \\
\mathit{disjoint}(\mathit{next}(x), y, \mathit{next}) \;\mathsf{and}\; \mathit{next}(x) \in \mathsf{dom}\; \mathit{next} \; ,
\end{array}
$$

as required.

Altogether we obtain, using Corollary 3.2,

$$
\begin{array}{l}
\mathsf{funct}\; \mathit{conc} \equiv (\mathsf{rcd}\; x, \mathsf{rcd}\; y, \mathsf{state}\; \mathit{next} : \mathit{disjoint}(x, y, \mathit{next}))\; \mathsf{state} : \\
\quad \mathsf{if}\; x = \Box\; \mathsf{then}\; \mathit{next}\; \mathsf{else}\; \mathit{neconc}(x, y, \mathit{next})\; \mathsf{fi} \\
\mathsf{where} \\
\mathsf{funct}\; \mathit{neconc} \equiv (\mathsf{rcd}\; x, \mathsf{rcd}\; y, \mathsf{state}\; \mathit{next} : \mathit{disjoint}(x, y, \mathit{next})\; \mathsf{and}\; x \in \mathsf{dom}\; \mathit{next})\; \mathsf{state} : \\
\quad \mathsf{if}\; \mathit{next}(x) = \Box\; \mathsf{then}\; [x \mapsto y] \mid \mathit{next} \\
\qquad\qquad \mathsf{else}\; \mathit{neconc}(\mathit{next}(x), y, \mathit{next})\; \mathsf{fi} \; .
\end{array}
$$

Termination of *neconc* follows from $\Box \in \mathit{next}^*(x)$. It is quite reassuring that the fundamental unfold/fold technique for deriving recursions also applies to pointer algorithms in this setting.

We briefly discuss an alternative derivation for the same problem. In Case 2 above we know by definition of *sequ* that $\mathit{sequ}(x, \mathit{next}) \not\equiv \langle\rangle$. Hence we may calculate

$$
\begin{array}{ll}
& \mathit{chain}(\mathit{sequ}(x, \mathit{next}) + \mathit{sequ}(y, \mathit{next})) \mid \mathit{next} \\
\equiv & \{\![\text{ definition of } \mathit{lead}, \mathit{last} \;]\!\} \\
& \mathit{chain}(\mathit{lead}(\mathit{sequ}(x, \mathit{next})) + \mathit{last}(\mathit{sequ}(x, \mathit{next})) + \mathit{sequ}(y, \mathit{next})) \mid \mathit{next} \\
\equiv & \{\![\text{ by Lemma 3.3(2) and associativity }]\!\} \\
& \mathit{chain}(\mathit{last}(\mathit{sequ}(x, \mathit{next})) + \mathit{sequ}(y, \mathit{next})) \mid \\
& \mathit{chain}(\mathit{lead}(\mathit{sequ}(x, \mathit{next})) + \mathit{last}(\mathit{sequ}(x, \mathit{next}))) \mid \mathit{next} \\
\equiv & \{\![\text{ definition of } \mathit{lead}, \mathit{last} \;]\!\} \\
& \mathit{chain}(\mathit{last}(\mathit{sequ}(x, \mathit{next})) + \mathit{sequ}(y, \mathit{next})) \mid \mathit{chain}(\mathit{sequ}(x, \mathit{next})) \mid \mathit{next} \\
\equiv & \{\![\text{ by annihilation, since } \mathit{chain}(\mathit{sequ}(x, \mathit{next})) \subseteq \mathit{next} \;]\!\} \\
& \mathit{chain}(\mathit{last}(\mathit{sequ}(x, \mathit{next})) + \mathit{sequ}(y, \mathit{next})) \mid \mathit{next} \\
\equiv & \{\![\text{ by Lemma 3.3(1) }]\!\} \\
& ([\mathit{last}(\mathit{sequ}(x, \mathit{next})) \mapsto \mathit{first}(\mathit{sequ}(y, \mathit{next}))] \cup \mathit{chain}(\mathit{sequ}(y, \mathit{next}))) \mid \mathit{next} \\
\equiv & \{\![\text{ by annihilation, since } \mathit{chain}(\mathit{sequ}(y, \mathit{next})) \subseteq \mathit{next} \;]\!\} \\
& [\mathit{last}(\mathit{sequ}(x, \mathit{next})) \mapsto \mathit{first}(\mathit{sequ}(y, \mathit{next}))] \mid \mathit{next} \\
\equiv & \{\![\text{ by Lemma 3.4(1) }]\!\} \\
& [\mathit{last}(\mathit{sequ}(x, \mathit{next})) \mapsto y] \mid \mathit{next} \; .
\end{array}
$$

So a different specification of *neconc* would be

$$
\mathsf{funct}\; \mathit{neconc} \equiv (\mathsf{rcd}\; x, \mathsf{rcd}\; y, \mathsf{state}\; \mathit{next} : \mathit{disjoint}(x, y, \mathit{next})\; \mathsf{and}\; x \in \mathsf{dom}\; \mathit{next})\; \mathsf{state} :
$$

$$[last(sequ(x, next)) \mapsto y] \,|\, next \ .$$

Now one can use the standard recursion for *last* and the definition of *sequ* to show, for $x \in \mathsf{dom}\ next$,

$$last(sequ(x, next)) \ \equiv\ \mathsf{if}\ next(x) \in \mathsf{dom}\ next\ \mathsf{then}\ last(sequ(next(x), next))\ \mathsf{else}\ x\ \mathsf{fi}\ .$$

A straightforward unfold/fold transformation for *neconc* then yields the same result as before. This second derivation invests more lemmata, whereas the first one is more direct.

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

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

Note that this clearly specifies the state *next* as a transient parameter, whereas x and y are passed by value. Therefore the imperative version of *pconc* needs local variables for x and y, whereas it may operate on *next* directly. This is described by the following schematic rule for transforming an assignment that involves a tail-recursive function into a loop:

$$\begin{array}{l}a := f(a, E) \\ \mathsf{where} \\ \mathsf{funct}\ f\ \equiv\ (\mathsf{t}\ a, \mathsf{u}\ b)\,\mathsf{t} : \mathsf{if}\ C\ \mathsf{then}\ T\ \mathsf{else}\ f(K, L)\ \mathsf{fi}\end{array}$$

$$\begin{array}{l}\text{———————————}\updownarrow\text{———————————}\left[\begin{array}{l} b \notin \mathrm{FREE}[\![E]\!] \\ f \notin \mathrm{FREE}[\![E, C, T, K, L]\!]\end{array}\right.\end{array}$$

$$\begin{array}{l}\lceil\ \mathsf{var\ u}\ b := E\ ; \\ \ \ \mathsf{while}\ \neg\ C\ \mathsf{do}\ (a, b) := (K, L)\ \mathsf{od}; \\ \ \ a := T \qquad\qquad\qquad\qquad\qquad\qquad \rfloor\ .\end{array}$$

The double arrow indicates that, under the syntactic conditions at the side, the top scheme is semantically equivalent to the bottom scheme. a is a global variable; since it is modified by the assignment anyway, it may also be used for accumulating the intermediate results. E, C, T, K, and L stand for expressions of appropriate types, possibly involving a and b, but not f. Note that the rule holds also if a and b stand for tuples of distinct identifiers.

Application of this rule (under appropriate renaming within *neconc*) leads to

$$\begin{array}{l}\mathsf{proc}\ pconc \ \equiv\ (\mathsf{var\ state}\ next, \mathsf{rcd}\ x, y : disjoint(x, y, next)) : \\ \quad\lceil\ \mathsf{if}\ x = \square \\ \qquad \mathsf{then}\ next := next \\ \qquad \mathsf{else}\ \ (\mathsf{var\ rcd}\ vx, vy) := (x, y)\ ; \\ \qquad\quad \mathsf{while}\ next(vx) \neq \square\ \mathsf{do}\ (next, vx, vy) := (next, next(vx), vy)\ \mathsf{od}; \\ \qquad\quad next := [vx \mapsto vy] \,|\, next \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{fi}\rfloor\ .\end{array}$$

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

$$\begin{array}{l}\mathsf{proc}\ pconc \equiv (\mathsf{var\ state}\ next, \mathsf{rcd}\ x, y : disjoint(x, y, next)) : \\ \quad \lceil\ \mathsf{if}\ x = \Box \\ \qquad \mathsf{then\ skip} \\ \qquad \mathsf{else\ var\ rcd}\ vx := x\ ; \\ \qquad\quad \mathsf{while}\ next(vx) \neq \Box\ \mathsf{do}\ vx := next(vx)\ \mathsf{od}; \\ \qquad\quad next := [vx \mapsto y] \mid next \qquad\qquad\qquad\qquad \mathsf{fi} \rfloor\ .\end{array}$$

If we write the assignment

$$next := [vx \mapsto y] \mid next$$

in a Pascal-like way as

$$vx\uparrow.next := y\ ,$$

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 except for disjointness of the two lists headed by x and y. Indeed, if in *next* there are pointers from other data structures to (parts of) these lists, 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 formulate stronger preconditions that exclude sharing if this is desired.

5 Chain Reversal

Next we want to derive a procedure for reversing a non-empty chain "in situ".

5.1 Specification and First Explicit Solution

Again we first specify a purely applicative version. The reverse of a chain should contain exactly the same proper records as the original chain, however, in reverse order of traversal. We can express this as follows:

$$\begin{array}{l}\mathsf{funct}\ reverse \equiv (\mathsf{rcd}\ x, \mathsf{state}\ next : \Box \in next^*(x))\,\mathsf{state} : \\ \quad chain(rev(sequ(x, next))) \mid next\ ,\end{array}$$

where *rev* is the reversal function on sequences:

$$\begin{array}{l}\mathsf{funct}\ rev \equiv (\mathsf{rcdsequ}\ s)\,\mathsf{rcdsequ} : \\ \quad \mathsf{if}\ s = \langle\rangle\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ rev(rest(s)) + first(s)\ \mathsf{fi}\ .\end{array}$$

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 generalised function *rrev* with an additional parameter t that accumulates the intermediate results:

$$\begin{array}{l}\mathsf{funct}\ rrev \equiv (\mathsf{rcdsequ}\ s, t)\,\mathsf{rcdsequ} : \\ \quad rev(s) + t\ .\end{array}$$

rev is embedded into *rrev* by

$$rev(s) \equiv rrev(s, \langle\rangle)\ .$$

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

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

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 record. Hence we define

$$
\begin{array}{l}
\mathsf{funct}\ \mathit{rreverse} \equiv (\mathsf{rcd}\ x, y, \mathsf{state}\ \mathit{next} : \mathit{disjoint}(x,y,\mathit{next}))\,\mathsf{state}: \\
\quad \mathit{chain}(\mathit{rev}(\mathit{sequ}(x,\mathit{next})) + \mathit{sequ}(y,\mathit{next})) \mid \mathit{next}\ .
\end{array}
$$

An appropriate embedding is

$$
\mathit{reverse}(x,\mathit{next}) \equiv \mathit{rreverse}(x,\Box,\mathit{next})\ ,
$$

since $\mathit{sequ}(\Box,\mathit{next}) \equiv \Diamond$.

As before, we now perform a case analysis.

Case 1: $x \equiv \Box$. Then $\mathit{sequ}(x,\mathit{next}) \equiv \Diamond$, and hence $\mathit{rev}(\mathit{sequ}(x,\mathit{next})) \equiv \Diamond$. Thus

$$
\begin{array}{ll}
& \mathit{chain}(\mathit{rev}(\mathit{sequ}(x,\mathit{next})) + \mathit{sequ}(y,\mathit{next})) \mid \mathit{next} \\
\equiv & \mathit{chain}(\mathit{sequ}(y,\mathit{next})) \mid \mathit{next} \\
\equiv & \quad \{\!\![\ \text{since } \mathit{chain}(\mathit{sequ}(y,\mathit{next})) \subseteq \mathit{next} \text{ by Lemma 3.4(2)}\ ]\!\!\} \\
& \mathit{next}\ .
\end{array}
$$

Case 2: $x \not\equiv \Box$. Then $\mathit{sequ}(x,\mathit{next}) \equiv x + \mathit{sequ}(\mathit{next}(x),\mathit{next})$, and hence $\mathit{rev}(\mathit{sequ}(x,\mathit{next}))$ $\equiv \mathit{rev}(\mathit{sequ}(\mathit{next}(x),\mathit{next})) + x$. Thus

$$
\begin{array}{ll}
& \mathit{chain}(\mathit{rev}(\mathit{sequ}(x,\mathit{next})) + \mathit{sequ}(y,\mathit{next})) \mid \mathit{next} \\
\equiv & \mathit{chain}(\mathit{rev}(\mathit{sequ}(\mathit{next}(x),\mathit{next})) + x + \mathit{sequ}(y,\mathit{next})) \mid \mathit{next} \\
\equiv & \quad \{\!\![\ \text{by Lemma 3.3(3), Lemma 3.4(1), and idempotence (Lemma 2.9)}\ ]\!\!\} \\
& \mathit{chain}(\mathit{rev}(\mathit{sequ}(\mathit{next}(x),\mathit{next})) + x + \mathit{sequ}(y,\mathit{next})) \mid [x \mapsto y] \mid \mathit{next} \\
\equiv & \quad \{\!\![\ \text{by Lemma 3.5(1), since } \mathit{disjoint}(x,y,\mathit{next}) \text{ implies} \\
& \qquad x \notin \mathit{next}^*(y) \text{ and } x \notin \mathit{next}^*(\mathit{next}(x))\ ]\!\!\} \\
& \mathit{chain}(\mathit{rev}(\mathit{sequ}(\mathit{next}(x),[x \mapsto y] \mid \mathit{next})) + x + \mathit{sequ}(y,[x \mapsto y] \mid \mathit{next})) \mid [x \mapsto y] \mid \mathit{next} \\
\equiv & \quad \{\!\![\ \text{definition of } \mathit{sequ}\ ]\!\!\} \\
& \mathit{chain}(\mathit{rev}(\mathit{sequ}(\mathit{next}(x),[x \mapsto y] \mid \mathit{next})) + \mathit{sequ}(x,[x \mapsto y] \mid \mathit{next})) \mid [x \mapsto y] \mid \mathit{next} \\
\equiv & \quad \{\!\![\ \text{fold } \mathit{rreverse}\ ]\!\!\} \\
& \mathit{rreverse}(\mathit{next}(x), x, [x \mapsto y] \mid \mathit{next})\ .
\end{array}
$$

Again, we have to check the validity of the assertion for the recursive call. Assuming that $x \not\equiv \Box$ and $\mathit{disjoint}(x,y,\mathit{next})$ holds, we calculate:

$$
\begin{array}{ll}
& \mathit{disjoint}(\mathit{next}(x), y, [x \mapsto y] | \mathit{next}) \\
\equiv & \quad \{\!\![\ \text{definition of } \mathit{disjoint}\ ]\!\!\} \\
& ([x \mapsto y] | \mathit{next})^*(\mathit{next}(x)) \cap ([x \mapsto y] | \mathit{next})^*(x) = \{\Box\} \\
\equiv & \quad \{\!\![\ \text{by Lemma 3.5(2)}\ ]\!\!\} \\
& \mathit{next}^*(\mathit{next}(x)) \cap ([x \mapsto y] | \mathit{next})^*(x) = \{\Box\} \\
\equiv & \quad \{\!\![\ \text{definition of } ([x \mapsto y] | \mathit{next})^*\ ]\!\!\} \\
& \mathit{next}^*(\mathit{next}(x)) \cap (\{x\} \cup ([x \mapsto y] | \mathit{next})^*(y)) = \{\Box\} \\
\equiv & \quad \{\!\![\ \text{by Lemma 3.5(2) and } \mathit{disjoint}(x,y,\mathit{next})\ ]\!\!\}
\end{array}
$$

$$
\begin{array}{ll}
& next^*(next(x)) \cap (\{x\} \cup next^*(y)) = \{\Box\} \\
\equiv & \quad \{\![\text{ distributivity }]\!\} \\
& (next^*(next(x)) \cap \{x\}) \cup (next^*(next(x)) \cap next^*(y)) = \{\Box\} \\
\equiv & \quad \{\![\, x \neq \Box,\ next^*(next(x)) \cap \{x\} \subseteq \{x\} \,]\!\} \\
& next^*(next(x)) \cap \{x\} = \emptyset \wedge next^*(next(x)) \cap next^*(y) = \{\Box\} \ .
\end{array}
$$

Both conjuncts are implied by $disjoint(x, y, next)$, since that also implies $\Box \in next^*(x)$.

Altogether, we have

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

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

5.2 A Version With Selective Updating

Specifying a procedure

$$
\begin{array}{l}
\mathsf{proc}\ preverse \equiv (\mathsf{var\ state}\ next, \mathsf{rcd}\ x : \Box \in next^*(x)) : \\
\quad next := reverse(x, next)\ ,
\end{array}
$$

we obtain, analogously to the previous section, the final version

$$
\begin{array}{l}
\mathsf{proc}\ preverse \equiv (\mathsf{var\ state}\ next, \mathsf{rcd}\ x : \Box \in next^*(x)) : \\
\quad \lceil\ (\mathsf{var\ rcd}\ vx, vy) := (x, \Box)\ ; \\
\quad\ \mathsf{while}\ vx \neq \Box\,\mathsf{do}\,(vx, vy, next) := (next(x), vx, [vx \mapsto vy] \mid next)\ \mathsf{od}\ \rfloor\ .
\end{array}
$$

Note that sequentialisation 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. [5, 10]), 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 two examples of 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. [5, 9, 12]) 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 many quantifiers (as compared e.g. to [4]). Moreover, the operations of that algebra are expressive enough that we did not need to explain anything with the help of pictures. This may seem due to the simplicity of the algorithms. However, the situation almost seems to be reversed: when developing the intricate garbage collection algorithm described in [3] we quite soon stopped drawing pictures, because they became so complicated as to be ununderstandable. Contrarily, the algebraic formulation was clear and modular so that one exactly saw what was going on. Another advantage of the applicative treatment is that if additional predicates or operations on maps are needed, they are 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 [22].

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 "Algorithmic Languages and Calculi", notably by the algebraic way in which R. Bird and L. Meertens develop tree and list algorithms. I gratefully acknowledge helpful comments by F.L. Bauer, U. Berger, J. Desharnais, W. Dosch, H. Ehler, W. Meixner, O. de Moor, H. Partsch, P. Pepper, M. Russling, M. Sintzoff and the anonymous referees.

7 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. 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 (eds.): Methods of programming. Lecture Notes in Computer Science **544**. Berlin: Springer 1991, 137–192

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. E.C.R. Hehner: A practical theory of programming. Forthcoming book

7. A. Horsch: Functional programming with partially applicable operators. Fakultät für Mathematik und Informatik, Technische Universität München, Dissertation, 1989

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

9. A. Kausche: Modale Logiken von geflechtartigen Datenstrukturen und ihre Kombination mit temporaler Programmlogik. Fakultät für Mathematik und Informatik, Technische Universität München, Dissertation, 1989

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

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

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

13. B. Möller: Relations as a program development language. In B. Möller (ed.): Constructing programs from specifications. Proc. IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13–16 May 1991. Amsterdam: North-Holland 1991, 373–397

14. B. Möller: An algebraic treatment of sorting. Document Nr. 693 AUG-7 of IFIP WG 2.1, Sept. 1992

15. B. Möller: Derivation of graph and pointer algorithms. Institut für Mathematik der Universität Augsburg, Report No. 280, 1993. Also in B. Möller, H.A. Partsch, S.A. Schuman (eds.): Formal program development. Proc. IFIP TC2/WG 2.1 State of the Art Seminar, Rio de Janeiro, Jan. 1992. Lecture Notes in Computer Science. Berlin: Springer (to appear)

16. B. Möller, M. Russling: Shorter paths to graph algorithms. Proc. 1992 International Conference on Mathematics of Program Construction (to appear). Extended version: Institut für Mathematik der Universität Augsburg, Report Nr. 272, 1992. Also to appear in Science of Computer Programming

17. P. Pepper, B. Möller: Programming with (finite) mappings. In: M. Broy (ed.): Informatik und Mathematik. Berlin: Springer 1991, 381–405

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

19. M. Russling: Hamiltonian sorting. Institut für Mathematik der Universität Augsburg, Report Nr. 270, 1992

20. G. Schmidt, T. Ströhlein: Relationen und Graphen. Berlin: Springer 1989. English version: Relations and graphs (forthcoming)

21. A. Tarski: On the calculus of relations. J. Symbolic Logic **6**, 73–89 (1941)

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

8 Appendix: The Algebra of Relations and Partial Maps

We have given our definitions of the relational operations in set theoretic terms. However, in the literature (see e.g. [20]) one frequently works with an abstraction of this framework, viz. with abstract relational algebras. In this appendix we shall show that our laws hold in this more general setting as well.

8.1 Abstract Relational Algebras

An **abstract relational algebra** consists of a set A together with binary operations

$$\cup, \cap, ; : A \times A \to A \ ,$$

unary operations

$$\bar{\cdot}, \cdot^{-1} : A \to A \ ,$$

and constants

$$\emptyset, I, \mathbb{1} \in A \ ,$$

such that

1. $(A, \cup, \cap, \bar{\cdot}, \emptyset, \mathbb{1})$ forms a Boolean algebra, i.e., a complete, distributive, atomic, and complementary lattice, the corresponding lattice order being denoted by $\subseteq$;
2. $(A, ;, I)$ is a monoid;
3. Tarski's rule $R \neq \emptyset \Rightarrow \mathbb{1} ; R ; \mathbb{1} = \mathbb{1}$ holds;
4. Dedekind's rule $R ; S \cap T \subseteq (R \cap T ; S^{-1}) ; (S \cap R^{-1} ; T)$ is satisfied.

The elements of A are called **abstract relations**; the operation $\cdot^{-1}$ forms the **converse** of a relation, whereas ; is called **relational composition**. It is customary to use the convention that ; binds tighter than $\cup$ and $\cap$.

A concrete model of an abstract relational algebra is the set of all binary relations on a set M, where $\cup$, $\cap$, and $\bar{\cdot}$ are usual union, intersection, and complement, $\emptyset$ is the empty relation, $\mathbb{1}$ is the **universal relation** $M \times M$, and $I = \{(x, x) \mid x \in M\}$ is the **identity relation** on M. The converse is given by

$$R^{-1} = \{(y, x) \mid (x, y) \in R\} \ ,$$

while composition is defined as

$$R ; S = \{(x, z) \mid \exists\, y \in M : (x, y) \in R \wedge (y, z) \in S\} \ .$$

This is the generalisation of functional composition to relations.

The following properties are consequences of the definition. First, composition distributes through union:

$$\begin{aligned} (R \cup S) ; T &= R ; T \cup S ; T \ , \\ R ; (S \cup T) &= R ; S \cup R ; T \ . \end{aligned}$$

From this it follows that ; is monotonic w.r.t. inclusion of relations. However, it is only subdistributive w.r.t. intersection:

$$\begin{aligned} (R \cap S) ; T &\subseteq R ; T \cap S ; T \ , \\ R ; (S \cap T) &\subseteq R ; S \cap R ; T \ . \end{aligned}$$

The universal relation is idempotent:

$$\mathbb{1} ; \mathbb{1} = \mathbb{1} \ .$$

Composition is strict w.r.t. $\emptyset$:

$$\emptyset ; R = \emptyset = R ; \emptyset \ .$$

The following laws hold for the converse:

$$\begin{aligned} (R^{-1})^{-1} &= R\ , \\ (R\,;S)^{-1} &= S^{-1}\,;R^{-1}\ , \\ I^{-1} &= I\ . \end{aligned}$$

Moreover, the converse distributes through all lattice operations.

Interesting special relations arise from composition with universal relations. From the definitions it is straightforward that in the set theoretic model

$$R\,;\mathbb{1} \;=\; (\mathsf{dom}\,R) \times M\ .$$

Thus, $R\,;\mathbb{1}$ relates each element in the domain of R to each element of M; it therefore can serve as a relational representation of the domain of R (see e.g. [20]). Note that

$$R \;\subseteq\; R\,;\mathbb{1}\ .$$

Using this, we can form restriction operations. In the set theoretic model, the relation

$$R \cap S\,;\mathbb{1}$$

results from R by retaining only those pairs the first components of which are also in the domain of S. Similarly, forming

$$R \cap \overline{S\,;\mathbb{1}}$$

removes from R all pairs with a first component in the domain of S.

8.2 Overwriting

In this section we investigate the operation of **overwriting** one relation with another one. Given relations R, S we define

$$R\,|\,S \;\stackrel{\text{def}}{=}\; R \cup (S \cap \overline{R\,;\mathbb{1}})\ .$$

Hence, in the set theoretic model a pair (x, y) is in $R\,|\,S$ iff it is in R or x is not in the domain of R and (x, y) is in S, and thus the definition agrees with the one in Section 2. We use the convention that $|$ binds tighter than the lattice operations; however, no precedence is defined between ; and $|$.

We now prove a number of useful properties of overwriting. It should be noted that the proofs only use identities between relations and hence are valid in all abstract relational algebras.

Corollary 8.1 (Right-Distributivity)
$R\,|\,(S \cup T) \;=\; R\,|\,S \cup R\,|\,T\ .$

Proof: Immediate from the distributivity of $\cap$ over $\cup$. ∎

From this it follows that $|$ is monotonic in its right argument. Note, however, that $|$ is neither distributive nor monotonic in its left argument.

Lemma 8.2 (Extension)
(1) $R \subseteq R \,|\, S$.
(2) $R = R \,|\, S \cap R \,;\, \mathbb{1}$.

Proof: (1) is immediate from the definition.

(2)
$$\begin{array}{ll} & R \,|\, S \cap R \,;\, \mathbb{1} \\ = & \quad \{\![\text{ definition }]\!\} \\ & (R \cup (S \cap \overline{R \,;\, \mathbb{1}})) \cap R \,;\, \mathbb{1} \\ = & \quad \{\![\text{ distributivity }]\!\} \\ & (R \cap R \,;\, \mathbb{1}) \cup (S \cap \overline{R \,;\, \mathbb{1}} \cap R \,;\, \mathbb{1}) \\ = & \quad \{\![\text{ Boolean algebra, } R \subseteq R \,;\, \mathbb{1} \text{ }]\!\} \\ & R \ . \end{array}$$

■

Overwriting is idempotent:

Corollary 8.3 (Idempotence)
$R \,|\, R = R$.

Proof: Straightforward from the definition. ■

The following properties allow localising side effects to that part of a store they really affect:

Lemma 8.4 (Localisation)
(1) $R \,;\, \mathbb{1} \cap S \,;\, \mathbb{1} = \emptyset \Rightarrow R \,|\, (S \cup T) = S \cup R \,|\, T.$
(2) $R \,;\, \mathbb{1} \cap T \,;\, \mathbb{1} = \emptyset \Rightarrow (R \cup S) \,|\, T = R \cup S \,|\, T.$

Proof: (1) By the assumption, $S \subseteq S \,;\, \mathbb{1} \subseteq \overline{R \,;\, \mathbb{1}}$ $(*)$. Hence

$$\begin{array}{ll} & R \,|\, (S \cup T) \\ = & \quad \{\![\text{ right-distributivity (Corollary 8.1) }]\!\} \\ & R \,|\, S \cup R \,|\, T \\ = & \quad \{\![\text{ definition of } | \text{ }]\!\} \\ & R \cup (S \cap \overline{R \,;\, \mathbb{1}}) \cup R \,|\, T \\ = & \quad \{\![\text{ by } (*) \text{ and Boolean algebra }]\!\} \\ & R \cup S \cup R \,|\, T \\ = & \quad \{\![\text{ Lemma 8.2(1), Boolean algebra }]\!\} \\ & S \cup R \,|\, T \ . \end{array}$$

(2) By the assumption, $T \subseteq T \,;\, \mathbb{1} \subseteq \overline{R \,;\, \mathbb{1}}$ $(**)$. Hence

$$\begin{array}{ll} & (R \cup S) \,|\, T \\ = & \quad \{\![\text{ definition of } | \text{ }]\!\} \\ & R \cup S \cup (T \cap \overline{(R \cup S) \,;\, \mathbb{1}}) \\ = & \quad \{\![\text{ distributivity }]\!\} \end{array}$$

$$
\begin{array}{ll}
 & R \cup S \cup (T \cap \overline{R\,;\mathbb{1} \cup S\,;\mathbb{1}}) \\
= & \quad \{\![\text{ Boolean algebra }]\!\} \\
 & R \cup S \cup (T \cap \overline{R\,;\mathbb{1}} \cap \overline{S\,;\mathbb{1}}) \\
= & \quad \{\![\text{ by } (**) \text{ and Boolean algebra }]\!\} \\
 & R \cup S \cup (T \cap \overline{S\,;\mathbb{1}}) \\
= & \quad \{\![\text{ definition of } | \ ]\!\} \\
 & R \cup S \,|\, T \ .
\end{array}
$$

■

Now we show that $R \,|\, S$ lies in between $R \cap S$ and $R \cup S$. More precisely,

Lemma 8.5 (Median)
(1) $R \cup S = R \,|\, S \cup (S \cap R\,;\mathbb{1})$.
(2) $R \cap S = R \,|\, S \cap (S \cap R\,;\mathbb{1})$.
(3) $R \cap S \subseteq R \,|\, S \subseteq R \cup S$.

Proof: (1)

$$
\begin{array}{ll}
 & R \cup S \\
= & \quad \{\![\text{ Boolean algebra }]\!\} \\
 & R \cup (S \cap \overline{R\,;\mathbb{1}}) \cup (S \cap R\,;\mathbb{1}) \\
= & \quad \{\![\text{ definition }]\!\} \\
 & R \,|\, S \cup (S \cap R\,;\mathbb{1}) \ .
\end{array}
$$

(2) is immediate from Lemma 8.2(2).

(3) is immediate from (1) and (2).

■

Next we show that the domain of $R \,|\, S$ coincides with that of $R \cup S$.

Lemma 8.6 (Domain)
$(R \,|\, S)\,;\mathbb{1} = (R \cup S)\,;\mathbb{1}$.

Proof: By Lemma 8.5(1) and distributivity we have

$$(R \cup S)\,;\mathbb{1} = (R \,|\, S)\,;\mathbb{1} \cup (S \cap R\,;\mathbb{1})\,;\mathbb{1} \ .$$

However,

$$
\begin{array}{ll}
 & (S \cap R\,;\mathbb{1})\,;\mathbb{1} \\
\subseteq & \quad \{\![\text{ subdistributivity }]\!\} \\
 & S\,;\mathbb{1} \cap R\,;\mathbb{1}\,;\mathbb{1} \\
= & \quad \{\![\text{ idempotence of } \mathbb{1} \ ]\!\} \\
 & S\,;\mathbb{1} \cap R\,;\mathbb{1} \\
\subseteq & \quad \{\![\text{ Boolean algebra }]\!\} \\
 & R\,;\mathbb{1} \\
\subseteq & \quad \{\![\text{ Lemma 8.2(1), monotonicity }]\!\}
\end{array}
$$

$$(R \mid S)\,;\mathbb{1}\ ,$$

so that the claim follows by Boolean algebra. ∎

Relations form a monoid under overwriting:

Lemma 8.7 (Monoid)
(1) $\emptyset \mid R = R = R \mid \emptyset$.
(2) $(R \mid S) \mid T = R \mid (S \mid T)$.

Proof: (1) is straightforward from the definition.

(2)
$$\begin{array}{ll} & R \mid (S \mid T) \\ = & \quad \{\![\text{ definition }]\!\} \\ & R \cup ((S \cup (T \cap \overline{S\,;\mathbb{1}})) \cap \overline{R\,;\mathbb{1}}) \\ = & \quad \{\![\text{ distributivity }]\!\} \\ & R \cup (S \cap \overline{R\,;\mathbb{1}}) \cup (T \cap \overline{S\,;\mathbb{1}} \cap \overline{R\,;\mathbb{1}}) \\ = & \quad \{\![\text{ definition of } \mid \text{, de Morgan, distributivity }]\!\} \\ & R \mid S \cup (T \cap \overline{(S \cup R)\,;\mathbb{1}}) \\ = & \quad \{\![\text{ Lemma 8.6, Boolean algebra }]\!\} \\ & R \mid S \cup (T \cap \overline{(R \mid S)\,;\mathbb{1}}) \\ = & \quad \{\![\text{ definition of } \mid \ ]\!\} \\ & (R \mid S) \mid T\ . \end{array}$$

∎

Now we want to analyse the relation between overwriting and union in more detail.

Lemma 8.8 (Compatibility)
The following properties are equivalent:
(1) $R \mid S = R \cup S = S \mid R$.
(2) $R \mid S = S \mid R$.
(3) $R \cap S\,;\mathbb{1} = S \cap R\,;\mathbb{1}$.
(4) $R \cap S\,;\mathbb{1} = R \cap S = S \cap R\,;\mathbb{1}$.

Proof: (1) ⇒ (2) is immediate.

(2) ⇒ (3)
$$\begin{array}{ll} & R \cap S\,;\mathbb{1} \\ = & \quad \{\![\text{ Lemma 8.2(2) }]\!\} \\ & R \mid S \cap R\,;\mathbb{1} \cap S\,;\mathbb{1} \\ = & \quad \{\![\text{ assumption }]\!\} \\ & S \mid R \cap R\,;\mathbb{1} \cap S\,;\mathbb{1} \\ = & \quad \{\![\text{ Boolean algebra, Lemma 8.2(2) }]\!\} \\ & S \cap R\,;\mathbb{1}\ . \end{array}$$

(3) ⇒ (4)
$$\begin{array}{ll} & R \cap S \\ = & \quad \{\![\ S \subseteq S\,;\mathbb{1} \ ]\!\} \end{array}$$

$$
\begin{array}{ll}
 & R \cap S\,;\mathbb{1} \cap S \\
= & \quad \{\![\text{ assumption }]\!\} \\
 & S \cap R\,;\mathbb{1} \cap S \\
= & \quad \{\![\text{ Boolean algebra }]\!\} \\
 & S \cap R\,;\mathbb{1}\ .
\end{array}
$$

(4) $\Rightarrow$ (1)

$$
\begin{array}{ll}
 & R \cup S \\
= & \quad \{\![\text{ Lemma 8.5(1) }]\!\} \\
 & R \,|\, S \cup (S \cap R\,;\mathbb{1}) \\
= & \quad \{\![\text{ assumption }]\!\} \\
 & R \,|\, S \cup (R \cap S\,;\mathbb{1}) \\
= & \quad \{\![\text{ Boolean algebra, Lemma 8.2(1) }]\!\} \\
 & R \,|\, S\ .
\end{array}
$$

Now the claim follows from commutativity of $\cup$.

■

If any of these conditions holds we call R and S **compatible**. It should be noted that the weakening $R \,|\, S = R \cup S$ of property(1) does *not* imply (2): If $S \subseteq R$ we always have $R \,|\, S = R = R \cup S$. However, take in the set theoretic model $R = \{(1,1),(1,2),(2,1)\}$ and $S = \{(1,1)\}$. Then $S \,|\, R = \{(1,1),(2,1)\} \neq R = R \cup S$.

From the above lemma we obtain two useful corollaries:

Corollary 8.9 (Sequentialisation)
$(R \cup S) \,|\, T = R \,|\, (S \,|\, T)$ provided R and S are compatible.

Proof: Immediate from compatibility and associativity of $|$. ■

Corollary 8.10 (Annihilation)
$S \,|\, T = T \Rightarrow (R \cup S) \,|\, T = R \,|\, T$ provided R and S are compatible.

Proof: Immediate from sequentialisation. ■

8.3 Partial Maps

Now we consider the special case of maps. A relation R is a **(partial) map** if

$$R^{-1}\,;R \subseteq I\ .$$

In the set theoretic model this is equivalent to the definition given in Section 2.

Corollary 8.11 (Submap)
If S is a map and $R \subseteq S$ then R is a map as well.

Proof: Immediate by monotonicity. ■

Next, we show an auxiliary lemma:

Lemma 8.12 (Propagation)

(1) For relations R, S we have $R^{-1} ; S \subseteq I$ iff $S^{-1} ; R \subseteq I$.

(2) Let R be a map and S a relation. Then $R^{-1} ; S \subseteq I$ iff $S \cap R ; \mathbb{1} \subseteq R \cap S ; \mathbb{1}$.

Proof: (1)
$$
\begin{array}{ll}
 & R^{-1} ; S \subseteq I \\
\Leftrightarrow & \quad \{\![\text{ monotonicity }]\!\} \\
 & (R^{-1} ; S)^{-1} \subseteq I^{-1} \\
\Leftrightarrow & \quad \{\![\text{ converse }]\!\} \\
 & S^{-1} ; R \subseteq I \ .
\end{array}
$$

(2)$(\Rightarrow)$
$$
\begin{array}{ll}
 & R ; \mathbb{1} \cap S \\
\subseteq & \quad \{\![\text{ Dedekind }]\!\} \\
 & (R \cap S ; \mathbb{1}^{-1}) ; (\mathbb{1} \cap R^{-1} ; S) \\
= & \quad \{\![\text{ converse, Boolean algebra }]\!\} \\
 & (R \cap S ; \mathbb{1}) ; (R^{-1} ; S) \\
\subseteq & \quad \{\![\text{ assumption }]\!\} \\
 & (R \cap S ; \mathbb{1}) ; I \\
= & \quad \{\![\text{ neutrality }]\!\} \\
 & R \cap S ; \mathbb{1} \ .
\end{array}
$$

$(\Leftarrow)$
$$
\begin{array}{ll}
 & S^{-1} ; R \\
= & \quad \{\![\text{ Boolean algebra }]\!\} \\
 & S^{-1} ; R \cap \mathbb{1} \\
\subseteq & \quad \{\![\text{ Dedekind }]\!\} \\
 & (S^{-1} \cap \mathbb{1} ; R^{-1}) ; (R \cap (S^{-1})^{-1} ; \mathbb{1}) \\
= & \quad \{\![\text{ converse }]\!\} \\
 & (S \cap R ; \mathbb{1})^{-1} ; (R \cap S ; \mathbb{1}) \\
\subseteq & \quad \{\![\text{ assumption }]\!\} \\
 & (R \cap S ; \mathbb{1})^{-1} ; (R \cap S ; \mathbb{1}) \\
\subseteq & \quad \{\![\text{ Boolean algebra, monotonicity }]\!\} \\
 & R^{-1} ; R \\
\subseteq & \quad \{\![R \text{ map }]\!\} \\
 & I \ .
\end{array}
$$

Now the claim follows by (1).

■

In general, relational union does not preserve the map property. However, the above lemma gives us a good characterisation:

Corollary 8.13 (Compatibility)

Let $(R_j)_{j \in J}$ be a family of maps. Then $\bigcup_{j \in J} R_j$ is a map iff the R_j are pairwise compatible.

Proof: By distributivity we have

$$(\bigcup_{j\in J} R_j)^{-1} ; (\bigcup_{k\in J} R_k) \;=\; \bigcup_{j\in J}\bigcup_{k\in J} R_j^{-1} ; R_k$$

and hence

$$(\bigcup_{j\in J} R_j)^{-1} ; (\bigcup_{k\in J} R_k) \;\subseteq\; I \;\;\Leftrightarrow\;\; \forall\, j \in J : \forall\, k \in J : R_j^{-1} ; R_k \;\subseteq\; I \;.$$

Using the fact that the R_j are maps the claim now is immediate from Lemma 8.12(2) and Lemma 8.8(3). ∎

Maps are closed under overwriting:

Lemma 8.14 (Closure)
Let R, S be maps. Then $R|S$ is a map as well.

Proof: By definition, $R \,|\, S \;=\; R \cup (S \cap \overline{R ; \mathbb{1}})$. Corollary 8.11 shows that also $S \cap \overline{R ; \mathbb{1}}$ is a map, so that by Corollary 8.13 it suffices to show compatibility of R and $S \cap \overline{R ; \mathbb{1}}$. We calculate

$$\begin{array}{ll}
& R \cap (S \cap \overline{R ; \mathbb{1}}) ; \mathbb{1} \\
\subseteq & \quad \{\![\text{ subdistributivity }]\!\} \\
& R \cap S ; \mathbb{1} \cap \overline{R ; \mathbb{1}} ; \mathbb{1} \\
= & \quad \{\![\text{ Boolean algebra }]\!\} \\
& \overline{R ; \mathbb{1}} ; \mathbb{1} \cap R \cap S ; \mathbb{1} \\
\subseteq & \quad \{\![\text{ Dedekind, monotonicity }]\!\} \\
& (\overline{R ; \mathbb{1}} \cap R ; \mathbb{1}^{-1}) ; (\mathbb{1} \cap (\overline{R ; \mathbb{1}})^{-1} ; R) \cap S ; \mathbb{1} \\
= & \quad \{\![\text{ converse }]\!\} \\
& (\overline{R ; \mathbb{1}} \cap R ; \mathbb{1}) ; (\mathbb{1} \cap (\overline{R ; \mathbb{1}})^{-1} ; R) \cap S ; \mathbb{1} \\
= & \quad \{\![\text{ Boolean algebra }]\!\} \\
& \emptyset ; (\mathbb{1} \cap (\overline{R ; \mathbb{1}})^{-1} ; R) \cap S ; \mathbb{1} \\
= & \quad \{\![\text{ strictness }]\!\} \\
& \emptyset \;.
\end{array}$$

Moreover, Boolean algebra immediately shows $(S \cap \overline{R ; \mathbb{1}}) \cap R ; \mathbb{1} \;=\; \emptyset$. ∎

Next, we give a characterisation of inclusion:

Lemma 8.15 (Inclusion)
(1) For relations R, S we have $S \cap R ; \mathbb{1} \;=\; R \;\Rightarrow\; R \subseteq S$.
(2) If S is a map, the reverse implication holds as well.

Proof: (1) is immediate from Boolean algebra.

(2) We have

$$R ; \mathbb{1} \cap S$$

$$
\begin{array}{ll}
\subseteq & \quad \{\![\text{ Dedekind }]\!\} \\
& (R \cap S\,;\mathbb{1}^{-1})\,;(\mathbb{1} \cap R^{-1}\,;S) \\
\subseteq & \quad \{\![\text{ Boolean algebra, monotonicity }]\!\} \\
& R\,;R^{-1}\,;S \\
\subseteq & \quad \{\![\text{ assumption, monotonicity }]\!\} \\
& R\,;S^{-1}\,;S \\
\subseteq & \quad \{\![\text{ } S \text{ map, monotonicity }]\!\} \\
& R\,;I \\
= & \quad \{\![\text{ neutrality }]\!\} \\
& R\ .
\end{array}
$$

The reverse inclusion follows from the assumption and $R \subseteq R\,;\mathbb{1}$.

■

We show now that for maps inclusion implies compatibility, which is not necessarily the case for arbitrary relations.

Lemma 8.16 (Submap)
Let S be a map and $R \subseteq S$. Then R and S are compatible.

Proof:

$$
\begin{array}{ll}
& S \cap R\,;\mathbb{1} \\
= & \quad \{\![\text{ assumption, Lemma 8.15(2) }]\!\} \\
& R \\
= & \quad \{\![\text{ } R \subseteq R\,;\mathbb{1} \subseteq S\,;\mathbb{1} \text{ }]\!\} \\
& R \cap S\,;\mathbb{1}\ .
\end{array}
$$

■

This gives another kind of idempotence property:

Lemma 8.17 (Idempotence)
Let S be a map and T a relation. Then

$$R \subseteq S \Rightarrow S\,|\,T \equiv S\,|\,R\,|\,T\ .$$

Proof:

$$
\begin{array}{ll}
& S\,|\,T \\
\equiv & \quad \{\![\text{ assumption, Boolean algebra }]\!\} \\
& (S \cup R)\,|\,T \\
\equiv & \quad \{\![\text{ Lemma 8.16, Corollary 8.9 }]\!\} \\
& S\,|\,R\,|\,T\ .
\end{array}
$$

■

To proceed further, we now state some general facts about binary operations. Let $\oplus : N \times N \to N$ be a binary operation on some set N. Define, for $x, y \in N$,

$$x \leq_\oplus y \stackrel{\text{def}}{\Leftrightarrow} x \oplus y = y \ ,$$
$$x \ {}_\oplus\!\leq y \stackrel{\text{def}}{\Leftrightarrow} y \oplus x = y \ .$$

Lemma 8.18

(1) $\oplus$ is idempotent iff ${}_\oplus\!\leq$ and $\leq_\oplus$ are reflexive.

(2) If $\oplus$ is commutative then ${}_\oplus\!\leq$ and $\leq_\oplus$ are antisymmetric and coincide.

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

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

Proof: Immediate from the definitions. ∎

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

Lemma 8.19 (Preorders)

Let R and S be relations. Then
(1) $R \,|\, S = S \Leftrightarrow S \cap R \,;\, \mathbb{1} = R$.
(2) $S \,|\, R = S \Leftrightarrow R \,;\, \mathbb{1} \subseteq S \,;\, \mathbb{1}$.

Proof: (1)

$$\begin{array}{ll} & S = R \,|\, S \\ \Leftrightarrow & \quad \{\![\text{ Boolean algebra, definition }]\!\} \\ & (S \cap R \,;\, \mathbb{1}) \cup (S \cap \overline{R \,;\, \mathbb{1}}) = R \cup (S \cap \overline{R \,;\, \mathbb{1}}) \\ \Leftrightarrow & \quad \{\![\ (S \cap R \,;\, \mathbb{1}) \subseteq R \,;\, \mathbb{1} \text{ and } R \subseteq R \,;\, \mathbb{1} \ ]\!\} \\ & S \cap R \,;\, \mathbb{1} = R \ . \end{array}$$

(2)

$$\begin{array}{ll} & S \,|\, R = S \\ \Rightarrow & \quad \{\![\text{ equality }]\!\} \\ & (S \,|\, R) \,;\, \mathbb{1} = S \,;\, \mathbb{1} \\ \Leftrightarrow & \quad \{\![\text{ domain (Lemma 8.6) }]\!\} \\ & S \,;\, \mathbb{1} \cup R \,;\, \mathbb{1} = S \,;\, \mathbb{1} \\ \Leftrightarrow & \quad \{\![\text{ Boolean algebra }]\!\} \\ & R \,;\, \mathbb{1} \subseteq S \,;\, \mathbb{1} \\ \Rightarrow & \quad \{\![\ R \subseteq R \,;\, \mathbb{1}, \text{ Boolean algebra }]\!\} \\ & R \cap \overline{S \,;\, \mathbb{1}} = \emptyset \\ \Rightarrow & \quad \{\![\text{ definition }]\!\} \\ & S \,|\, R = S \ . \end{array}$$

∎

For maps the preorder $\leq_|$ has a simple characterisation:

Corollary 8.20 (Preorder)

Let R, S be maps. Then $R \,|\, S = S \Leftrightarrow R \subseteq S$.

Proof: ($\Rightarrow$) By definition, $R \cup (S \cap \overline{R\,;\mathbb{1}}) = S$ and hence $R \subseteq S$.

($\Leftarrow$) By Lemma 8.16, R and S are compatible and hence $R\,|\,S = R \cup S = S$.

■

From this and Corollary 8.10 we obtain immediately

Corollary 8.21 (Annihilation)

Let R, S, T be maps such that $S \subseteq T$ and R and S are compatible. Then

$$(R \cup S)\,|\,T = R\,|\,T\ .$$

We have given our derivations in the framework of an homogeneous abstract relational algebra. However, all proofs also extend directly to the case of heterogenous abstract relational algebras.