Derivation of Graph and Pointer Algorithms

Bernhard Möller

Institut für Mathematik, Universität Augsburg, D-86135 Augsburg, Germany,
e-mail: moeller@uni-augsburg.de

Abstract. We introduce operators and laws of an algebra of formal languages, a subalgebra of which corresponds to the algebra of (multiary) relations. This algebra is then used in the formal specification and derivation of some graph and pointer algorithms. This study is part of an attempt to single out a framework for program development at a very high level of discourse, close to informal reasoning but still with full formal precision.

1 Introduction

The transformational or calculational approach to program development has by now a long tradition [9, 1, 2, 27, 6]. There one starts from a (possibly non-executable) specification and transforms it into a (hopefully efficient) program using semantics-preserving rules. Many derivations, however, suffer from the use of lengthy and obscure expressions involving formulae from predicate calculus. This makes writing tedious and error-prone and reading difficult. Moreover, the lack of structure leads to large search spaces for supporting tools.

The aim of modern algebraic approaches (see e.g. [27, 6]) is to make program specification and calculation more compact and perspicuous. They attempt to identify frequently occurring patterns and to express them by operators satisfying strong algebraic properties. This way the formulas involved become smaller and contain less repetition, which also makes their writing safer and speeds up reading (once one is used to the operators). The intention is to raise the level of discourse in formal specification and derivation as closely as possible to that of informal reasoning, so that both formality and understandability are obtained at the same time. In addition, the search spaces for machine assistance become smaller, since the search can be directed by the combinations of occurring operators.

If one succeeds in finding equational laws, proof chains rather than complicated proof trees result as another advantage. Moreover, frequently aggregations of quantifiers can be packaged into operators; an equational law involving them usually combines a series of inference steps in pure predicate calculus into a single one.

We illustrate these ideas by treating some graph and pointer algorithms within a suitable algebra of formal languages which is a straigthened and simplified version of the framework presented in [29]. Our approach differs from that of [27, 6] in that we concentrate on properties of the underlying problem domain rather than on those of standard recursions over inductively defined data types.

2 The Algebra of Formal Languages

A formal language is a set of words over letters from some alphabet. Usually these letters are considered as "atomic". We shall take a more liberal view and allow arbitrary objects as letters. Then words over these "letters" can be viewed as representations for tuples or sequences of objects. In particular, if the alphabet consists of (names for) nodes of a directed graph, words can be considered as sequences of nodes and can thus model paths in the graph.

Relations are formal languages in which all words (or tuples) have equal length, called the arity of the relation. Relations of arity 1 are simply sets of letters and hence can represent sets of graph nodes, whereas binary relations can represent the edge sets of directed graphs. The only two nullary relations (the singleton relation consisting just of the empty word, and the empty relation) play the role of the Boolean values. This also allows easy definitions of assertions, conditional, and guards.

We extend some of the classical operations on relations to arbitrary languages in order to obtain a uniform overall framework. Essential operations on languages are concatenation, composition, and join. As special cases of composition we obtain image and inverse image as well as tests for intersection, emptiness, and membership. The join corresponds to path concatenation on directed graphs; special cases yield restriction.

Proofs of the properties in this section are either straightforward or given in [29] and therefore omitted here.

2.1 Words, Languages and Relations

Consider a (not necessarily finite) alphabet A. The set of all finite words over A is denoted by $A^{(*)}$. A **(formal) language** V over A is a subset $V \subseteq A^{(*)}$. As is customary in formal language theory, a singleton language is identified with its only word to save braces; moreover, a word consisting just of one letter is not distinguished from that letter.

By ε we denote the empty word over A and set $A^{(+)} \stackrel{\text{def}}{=} A^{(*)} \backslash \varepsilon$.

A **relation of arity** n is a language R such that all words in R have length n. In particular, the empty language $\emptyset$ is a relation of any arity. There are only two 0-ary relations, viz. $\emptyset$ and ε.

2.2 Pointwise Extension

We define our operations on languages first for single words and extend them pointwise to languages. We explain this mechanism for a unary operation; the extension to multiary ones is straightforward. Since we also need partial operations, we choose $\mathcal{P}(A^{(*)})$, the powerset of $A^{(*)}$, as the codomain for such an operation. The operation will then return a singleton language consisting of the result word, if this is defined, and the empty language $\emptyset$ otherwise. Thus $\emptyset$ plays the role of the error "value" $\perp$ in denotational semantics. Consider now such an operation $f : A^{(*)} \to \mathcal{P}(A^{(*)})$. Then the **pointwise extension** of f is denoted by the same symbol, has the functionality $f : \mathcal{P}(A^{(*)}) \to \mathcal{P}(A^{(*)})$ and is defined by

$$f(U) \stackrel{\text{def}}{=} \bigcup_{x \in U} f(x)$$

for $U \subseteq A^{(*)}$. By this definition, the extended operation distributes through union:

$$f(\cup \mathcal{V}) = \cup \{f(V) : V \in \mathcal{V}\}$$

for $\mathcal{V} \subseteq \mathcal{P}(A^{(*)})$. By taking $\mathcal{V} = \emptyset$ we obtain strictness of the pointwise extension with respect to $\emptyset$:

$$f(\emptyset) = \emptyset \ .$$

Moreover, taking $\mathcal{V} = \{U, V\}$ and using the equivalence

$$U \subseteq V \ \Leftrightarrow \ U \cup V = V \ ,$$

we also obtain monotonicity with respect to $\subseteq$:

$$U \subseteq V \ \Rightarrow \ f(U) \subseteq f(V) \ .$$

Monotonicity is crucial for least fixpoint semantics of recursion (see [22, 39]).

Finally, bilinear equational laws for f, i.e., laws in which each side has at most one occurrence of every variable, are inherited by the pointwise extension (see e.g. [23]).

2.3 Concatenation and Reversal

We now apply this mechanism to the operations of concatenation and reversal. Concatenation is denoted by $\bullet$. It is associative, with ε as its neutral element:

$$\begin{aligned} u \bullet (v \bullet w) &= (u \bullet v) \bullet w \ , \\ \varepsilon \bullet u \ = \ u \ &= \ u \bullet \varepsilon \ . \end{aligned}$$

The **reverse** u^{-1} of a word $u \in A^{(*)}$ is defined inductively by

$$\begin{aligned} \varepsilon^{-1} &= \varepsilon \ , \\ a^{-1} &= a \qquad\qquad (a \in A) \ , \\ (u \bullet v)^{-1} &= v^{-1} \bullet u^{-1} \ . \end{aligned}$$

Since associativity of concatenation, neutrality of ε and the defining laws for reversal are bilinear equational laws, they also hold for the pointwise extensions of these operations to languages:

$$\begin{aligned} U \bullet (V \bullet W) &= (U \bullet V) \bullet W \ , \\ \varepsilon \bullet U \ = \ U \ &= \ U \bullet \varepsilon \ , \\ (U \bullet V)^{-1} &= V^{-1} \bullet U^{-1} \ . \end{aligned}$$

The **identity** I_a over a letter $a \in A$ is defined by

$$I_a \stackrel{\text{def}}{=} a \bullet a$$

and extended pointwise to sets of letters. In particular, I_A is the binary identity relation (or diagonal) on A.

The operation set calculates the set of letters occurring in a word. It is defined inductively by

$$\begin{aligned} \mathsf{set}\,\varepsilon &= \emptyset \ , \\ \mathsf{set}\,a &= a && (a \in A) \ , \\ \mathsf{set}\,(u \bullet v) &= \mathsf{set}\,u \ \cup \ \mathsf{set}\,v && (u, v \in A^{(*)}) \ , \end{aligned}$$

and, again, extended pointwise to languages.

Finally, the first and last letters of a word (if any) are given by the operations fst and lst defined by

$$\begin{aligned} \mathsf{fst}\,\varepsilon \ &= \ \emptyset \ = \ \mathsf{lst}\,\varepsilon \ , \\ \mathsf{fst}\,(a \bullet u) \ &= \ a \qquad \mathsf{lst}\,(v \bullet b) \ = \ b \end{aligned}$$

for $a, b \in A$ and $u, v \in A^{(*)}$. Again, these operations are extended pointwise to languages. For a binary relation $R \subseteq A \bullet A$ they have special importance: $\mathsf{fst}\,R$ is the domain of R, whereas $\mathsf{lst}\,R$ is the codomain of R.

As unary operators, fst, lst and set bind strongest.

2.4 Join and Composition

For words s and t over alphabet A we define their **join** $s \bowtie t$ and their **composition** $s\,;t$ by

$$\varepsilon \bowtie s \ = \ \emptyset \ = \ s \bowtie \varepsilon \ , \qquad \varepsilon\,;s \ = \ \emptyset \ = \ s\,;\varepsilon \ ,$$

and, for $s, t \in A^{(*)}$ and $a, b \in A$, by

$$(s \bullet a) \bowtie (b \bullet t) \stackrel{\text{def}}{=} \begin{cases} s \bullet a \bullet t & \text{if } a = b \ , \\ \emptyset & \text{otherwise} \ , \end{cases} \qquad (s \bullet a) \ ; \ (b \bullet t) \stackrel{\text{def}}{=} \begin{cases} s \bullet t & \text{if } a = b \ , \\ \emptyset & \text{otherwise} \ . \end{cases}$$

These operations provide two different ways of "glueing" two words together upon a one-letter overlap: join preserves one copy of the overlap, whereas composition erases it. Again, they are extended pointwise to languages. On relations, the join is a special case of the one used in data base theory (see e.g. [19]). On binary relations, composition coincides with usual relational composition (see e.g. [38, 36]). To save parentheses we use the convention that $\bullet$, $\bowtie$ and ; bind stronger than all set-theoretic operations.

To exemplify the close connection between join and composition further, we consider a binary relation $R \subseteq A \bullet A$ modelling the edges of a directed graph with node set A. Then

$$\begin{aligned} R \bowtie R &= \{a \bullet b \bullet c \ : \ a \bullet b \in R \ \wedge \ b \bullet c \in R\} \ , \\ R\,;R &= \{a \bullet c \ : \ a \bullet b \in R \ \wedge \ b \bullet c \in R\} \ . \end{aligned}$$

Thus, the relation $R \bowtie R$ consists of exactly those paths $a \bullet b \bullet c$ which result from glueing two edges together at a common intermediate node. The composition $R\,;R$ is an abstraction of this; it just states whether there is a path from a to c via some intermediate point without making that point explicit. Iterating this observation shows that the relations

$$R, \quad R \bowtie R, \quad R \bowtie (R \bowtie R), \ \ldots$$

consist of the paths with exactly $1, 2, 3, \ldots$ edges in the directed graph associated with R, whereas the relations

$$R, \quad R\,;R, \quad R\,;(R\,;R), \ \ldots$$

just state existence of these paths between pairs of vertices.

The operations associate nicely with each other and with concatenation:

$$\left.\begin{array}{rll} U \bullet (V \bullet W) = (U \bullet V) \bullet W & , & \\ U \bowtie (V \bowtie W) = (U \bowtie V) \bowtie W & , & \\ U\,;(V\,;W) = (U\,;V)\,;W & \Leftarrow & V \cap A = \emptyset \ , \\ U\,;(V \bowtie W) = (U\,;V) \bowtie W & \Leftarrow & V \cap A = \emptyset \ , \\ (U \bowtie V)\,;W = U \bowtie (V\,;W) & \Leftarrow & V \cap A = \emptyset \ , \\ U \bullet (V \bowtie W) = (U \bullet V) \bowtie W & \Leftarrow & V \cap \varepsilon = \emptyset \ , \\ U \bowtie (V \bullet W) = (U \bowtie V) \bullet W & \Leftarrow & V \cap \varepsilon = \emptyset \ , \\ U \bullet (V\,;W) = (U \bullet V)\,;W & \Leftarrow & V \cap \varepsilon = \emptyset \ , \\ U\,;(V \bullet W) = (U\,;V) \bullet W & \Leftarrow & V \cap \varepsilon = \emptyset \ . \end{array}\right\} \tag{1}$$

We shall omit parentheses whenever one of these laws applies.

Interesting special cases arise when one of the operands of join or composition is a relation of arity 1. Suppose $R \subseteq A$. Then

$$R \bowtie S = \{a \bullet u : a \in R \wedge a \bullet u \in S\} \ , \qquad R\,;S = \{u : a \in R \wedge a \bullet u \in S\} \ .$$

In other words, $R \bowtie S$ selects all words in S that start with a letter in R, whereas $R\,;S$ not only selects all those words but also removes their first letters. Therefore, if S is binary, $R \bowtie S$ is the restriction of S to R, whereas $R\,;S$ is the image of R under S. Likewise, if $T \subseteq A$ then $S \bowtie T$ selects all words in S that end with a letter in T, whereas $S\,;T$ not only selects all those words but also removes their last letters. Therefore, if S is binary, $S \bowtie T$ is the corestriction of S to T, whereas $S\,;T$ is the inverse image of T under S. For these reasons we obtain domain and codomain of a binary relation $R \subseteq A \bullet A$ by

$$\mathsf{fst}\, R = R\,;A \qquad \mathsf{lst}\, R = A\,;R \ . \tag{2}$$

For binary $R \subseteq A \bullet A$ and $S, T \subseteq A$ we have, moreover,

$$S \bowtie R \bowtie T = S \bullet T \cap R \ , \qquad S\,;R\,;T = \begin{cases} \varepsilon & \text{if } S \bullet T \cap R \neq \emptyset \ , \\ \emptyset & \text{otherwise} \ . \end{cases} \tag{3}$$

Setting $S = T = A$ we obtain

$$A \bowtie R \bowtie A = \emptyset \ \Leftrightarrow \ R = \emptyset \ \Leftrightarrow \ A\,;R\,;A = \emptyset \ . \tag{4}$$

This is related to Tarski's rule $R \neq \emptyset \Rightarrow \mathbb{1}\,;R\,;\mathbb{1} = \mathbb{1}$ where $\mathbb{1} \stackrel{\text{def}}{=} A \bullet A$ (see [38, 36]).

If both $R \subseteq A$ and $S \subseteq A$ we have

$$R \bowtie S = R \cap S \ , \qquad R\,;S = \begin{cases} \varepsilon & \text{if } R \cap S \neq \emptyset \ , \\ \emptyset & \text{if } R \cap S = \emptyset \ . \end{cases} \tag{5}$$

Using (2) and (4) we obtain from this, for $R, S \subseteq A \bullet A$,

$$R\,;S = \emptyset \ \Leftrightarrow \ \mathsf{lst}\, R \cap \mathsf{fst}\, S = \emptyset \ . \tag{6}$$

We also have neutral elements for join and composition. Assume $A \supseteq P \supseteq \mathsf{fst}\, V$ and $A \supseteq Q \supseteq \mathsf{lst}\, V$ and $V \cap \varepsilon = \emptyset$. Then

$$P \bowtie V = V = V \bowtie Q \ , \qquad I_P \,;\, V = V = V \,;\, I_Q \ . \tag{7}$$

In special cases join and composition can be transformed into each other: assume $P, Q \subseteq A$ and let R be an arbitrary language. Then

$$P \bowtie R = I_P \,;\, R \ , \qquad R \bowtie Q = R \,;\, I_Q \ , \tag{8}$$

$$P \,;\, (Q \bowtie R) = (P \bowtie Q) \,;\, R \ , \qquad (R \bowtie P) \,;\, Q = R \,;\, (P \bowtie Q) \ . \tag{9}$$

Our final remark concerns decompositions of a relation into subrelations. Consider an n-ary relation R with $n \geq 1$. For $a \in \mathsf{fst}\, R$ the set $a \,;\, R$ is the image of a under R, and so $a \bullet a \,;\, R$ is the subrelation of R connecting a with all its images under R. Likewise, for $b \in \mathsf{lst}\, R$ the subrelation $R \,;\, b \bullet b$ connects all inverse images of b with b. Using this, we can decompose R in a domain-oriented and a range-oriented way:

$$R = \bigcup_{a \in \mathsf{fst}\, R} a \bullet a \,;\, R \qquad \text{(domain-oriented)} \ , \tag{10}$$

$$R = \bigcup_{b \in \mathsf{lst}\, R} R \,;\, b \bullet b \qquad \text{(range-oriented)} \ . \tag{11}$$

Note that this is just a restatement of equations (7) taking distributivity and associativity into account.

If we take as our relation R the partial map representing the dereferencing operation on a von Neumann storage, the range-oriented decomposition of R plays a prominent role in the garbage collection problem of Section 5.9; it leads to a chaining of all cells which point to a given other cell, so that all these pointers can be adjusted in one pass in the compactifying relocation.

2.5 Assertions and Conditional

As we have seen in (3) and (5), the nullary relations ε and $\emptyset$ behave like the outcomes of certain tests. Therefore they can be used instead of Boolean values, and we call relational expressions yielding nullary relations **assertions**. Note that in this view "false" and "undefined" both are represented by $\emptyset$. Negation is defined by

$$\overline{\emptyset} \stackrel{\text{def}}{=} \varepsilon \ , \qquad \overline{\varepsilon} \stackrel{\text{def}}{=} \emptyset \ .$$

Conjunction and disjunction of assertions are represented by their intersection and union. To improve readability, we write $B \wedge C$ for $B \cap C = B \bullet C$ and $B \vee C$ for $B \cup C$.

For assertion B and language U we have

$$B \bullet U = U \bullet B = \begin{cases} U & \text{if } B = \varepsilon \ , \\ \emptyset & \text{if } B = \emptyset \ . \end{cases}$$

Hence $B \bullet U$ (and $U \bullet B$) behaves like the expression

$$B \rhd U = \mathsf{if}\ B\ \mathsf{then}\ U\ \mathsf{else}\ \mathsf{error}\ \mathsf{fi}$$

in [28] and can be used for propagating assertions through recursions.

Using assertions we can also define a conditional by

$$\mathsf{if}\ B\ \mathsf{then}\ U\ \mathsf{else}\ V\ \mathsf{fi} \stackrel{\text{def}}{=} B \bullet U \ \cup\ \overline{B} \bullet V$$

for assertion B and languages U, V. Although this construct is not monotonic in B, it is monotonic in U and V. So we can still use it in recursions provided recursion occurs only in the branches and not in the condition.

3 Closure Operations

We now study in more detail iterated join and composition of a binary relation with itself, which was already seen to be important for path problems. A useful notion for treating both operations uniformly is that of a Kleene algebra.

3.1 Kleene Algebras and Closures

A **Kleene algebra** (see [11]) is a tuple $(S, \Sigma, \circ, 0, 1)$ consisting of a set S and operations $\Sigma : \mathcal{P}(S) \to S$ and $\circ : S \bullet S \to S$ as well as elements $0, 1 \in S$ such that $(S, \circ, 1)$ is a monoid and

$$\begin{aligned} \Sigma\,\emptyset &= 0 \ , & \\ \Sigma\,\{x\} &= x & (x \in S) \ , \\ \Sigma(\cup\mathcal{K}) &= \Sigma\,\{\Sigma\,K : K \in \mathcal{K}\} & (\mathcal{K} \subseteq \mathcal{P}(S)) \ , \\ \Sigma\,(K \circ L) &= (\Sigma\,K) \circ (\Sigma\,L) & (K, L \in \mathcal{P}(S)) \ , \end{aligned}$$

where in this latter equation $\circ$ is the pointwise extension of the monoid operation. Note that this implies that 0 is a zero with respect to $\circ$ or, in other words, that $\circ$ is strict with respect to 0:

$$0 \circ x \ = \ 0 \ = \ x \circ 0 \ . \tag{12}$$

The binary version of Σ is

$$x + y \stackrel{\text{def}}{=} \Sigma\,\{x, y\} \ ,$$

which makes $(S, +, 0)$ a commutative monoid.

A straightforward way of obtaining a Kleene algebra is to start from a monoid $(M, \circ, 1)$, to set $S \stackrel{\text{def}}{=} \mathcal{P}(M)$, $\Sigma \stackrel{\text{def}}{=} \bigcup$, $0 \stackrel{\text{def}}{=} \emptyset$ and to use the pointwise extension of $\circ$ to S. Therefore, given an alphabet A, by (1)

$$LAN \stackrel{\text{def}}{=} (\mathcal{P}(A^{(*)}),\ \bigcup,\ \bullet,\ \emptyset,\ \varepsilon) \ ,$$

forms a Kleene algebra. In addition, it is easily checked that also

$$\begin{aligned} REL &\stackrel{\text{def}}{=} (\mathcal{P}(A \bullet A),\ \bigcup,\ ;,\ \emptyset,\ I_A) \ , \\ PAT &\stackrel{\text{def}}{=} (\mathcal{P}(A^{(+)}),\ \bigcup,\ \bowtie,\ \emptyset,\ A) \end{aligned}$$

form Kleene algebras.

Given a Kleene algebra $(S, \Sigma, \circ, 0, 1)$, one can define a partial order $\leq$ on S by

$$x \leq y \stackrel{\text{def}}{\Leftrightarrow} x + y = y \ .$$

In our examples $\leq$ coincides with $\subseteq$. The pair $(S, \leq)$ forms a complete lattice with $\mathsf{lub}\, T = \Sigma\, T$. Moreover, $\circ$ and $+$ are continuous with respect to $\leq$. Therefore, recursive definitions using $\circ$ and $+$ can be given least fixpoint semantics using the fixpoint theorem of [22, 39]. In the case of LAN, systems of mutually recursive equations for languages give the power of context-free grammars (see e.g. [15]).

We use recursion to define the operators of **closure** $.^*$ and **proper closure** $.^+$ by

$$x^* \stackrel{\text{def}}{=} \mu\, y\, .\, 1 + x \circ y \ , \qquad x^+ \stackrel{\text{def}}{=} \mu\, y\, .\, x + x \circ y \ , \tag{13}$$

where μ is the least fixpoint operator. Using continuity we can also represent the closures by Kleene's approximation sequence [21] as

$$x^* = \Sigma\{x^j : j \in \mathbb{N}\} \ , \qquad x^+ = \Sigma\{x^j : j \in \mathbb{N}\backslash 0\} \ , \tag{14}$$

where the x^j are the powers of x in the monoid $(S, \circ, 1)$, defined as usual by $x^0 \stackrel{\text{def}}{=} 1$ and $x^{j+1} \stackrel{\text{def}}{=} x \circ x^j$. By strictness (12) we have

$$x^j \neq 0 \ \Rightarrow \ x^i \neq 0 \text{ for all } i \leq j \ . \tag{15}$$

We now state an important induction principle for closures; it is a specialised version of fixpoint induction (see e.g. [25]). We call a predicate P over a Kleene algebra $(S, \Sigma, \circ, 0, 1)$ **continuous** if for all $T \subseteq S$

$$(\bigwedge_{x \in T} P[x]) \Rightarrow P[\Sigma\, T] \ .$$

Lemma 1 (Closure Induction). *Consider a fixed $z \in S$ and let P be continuous and satisfy $\forall\, x : P[x] \Rightarrow P[z \circ x]$ or $\forall\, x : P[x] \Rightarrow P[x \circ z]$.*

(a) *If $\bigwedge_{i=0}^{n} P[z^i]$ holds for some n, then $P[z^*]$ holds as well.*

(b) *If $\bigwedge_{i=1}^{n} P[z^i]$ holds for some $n \geq 1$, then $P[z^+]$ holds as well.*

Proof. A straightforward induction shows $P[z^i]$ for all $i \in \mathbb{N}$ in (a) and for all $i \in \mathbb{N}\backslash 0$ in (b). Now the claims are immediate from Kleene's approximation (14) and continuity. □

The generalisation of this principle to a simultaneous induction on several variables should be obvious.

Using this principle one can show the following properties which are well-known from the algebra of regular expressions:

$$x \circ x^* = x^* \circ x \ , \tag{16}$$

$$(x + y)^* = x^* \circ (y \circ x^*)^* \ . \tag{17}$$

From the latter we obtain by the fixpoint property of the closure and distributivity

$$(x + y)^* = x^* + x^* \circ y \circ (x + y)^* \ . \tag{18}$$

3.2 Application to Paths

For our particular Kleene algebras LAN, REL and PAT we denote the closure operations by $.^{(*)}$, $.^*$ and $.^{\bowtie}$ and the proper closures by $.^{(+)}$, $.^+$ and $.^{\Rightarrow}$, respectively. Consider now a binary relation $R \subseteq A \bullet A$ and let G be the directed graph associated with R, i.e., the graph with node set A and arcs between the vertices corresponding to the pairs in R. By (3) we have, in REL,

$$a \,;\, R^i \,;\, b = \begin{cases} \varepsilon \text{ if there is a path with } i \text{ edges from } a \text{ to } b \text{ in } G \ , \\ \emptyset \ \text{ otherwise } . \end{cases}$$

Hence,

$$a \,;\, R^* \,;\, b = \begin{cases} \varepsilon \text{ if there is a path from } a \text{ to } b \text{ in } G \ , \\ \emptyset \ \text{ otherwise } , \end{cases}$$
$$a \,;\, R^+ \,;\, b = \begin{cases} \varepsilon \text{ if there is a path with at least one edge from } a \text{ to } b \text{ in } G \ , \\ \emptyset \ \text{ otherwise } . \end{cases}$$

So R^* is the reflexive-transitive closure and R^+ is the transitive closure of R. Hence, for $S \subseteq A$, the set $S \,;\, R^*$ gives all points in A reachable from points in S via paths in G, whereas $R^* \,;\, S$ contains all points in A from which some point in S can be reached. Finally,

$$S \,;\, R^* \,;\, T = \begin{cases} \varepsilon \text{ if } S \text{ and } T \text{ are connected by some path in } G \ , \\ \emptyset \ \text{ otherwise } . \end{cases}$$

Analogously, the **path closure** $R^{\bowtie}$ of R in PAT consists of all finite paths in G (with $A \subseteq R^{\bowtie}$ being the set of all paths with 0 edges), whereas the **proper path closure** $R^{\Rightarrow}$ consists of all paths with at least one edge. Hence, by (3), $a \bowtie R^{\bowtie} \bowtie b$ is the language of all paths between a and b in G, whereas $a \bowtie R^{\Rightarrow} \bowtie b$ is the language of all proper paths between a and b in G.

Moving away from the graph view, the path closure also is useful for general binary relations. Let e.g. $\leq$ be a partial order on A. Then $\leq^{\bowtie}$ is the language of all $\leq$-non-decreasing sequences. If $\leq$ is even a total order, then $\leq^{\bowtie}$ is the language of all sequences which are sorted with respect to $\leq$. This is exploited in [31, 35] for the derivation of sorting algorithms.

We can establish a relationship between PAT and REL by the mapping

$$\phi : \mathcal{P}(A^{(+)}) \ \to \ \mathcal{P}(A \bullet A) \ ,$$

defined on words by

$$\phi(u) \stackrel{\text{def}}{=} \mathsf{fst}\, u \bullet \mathsf{lst}\, u \tag{19}$$

and extended pointwise to languages. Hence ϕ is continuous and strict. For binary $R \subseteq A \bullet A$ we obtain from the definition $\phi(R) = R$, so that, for $S, T \subseteq A$ and $R \subseteq A \bullet A$, we get by (3),

$$\phi(S \bowtie R \bowtie T) \ = \ S \bullet T \cap R \ . \tag{20}$$

It turns out that ϕ is a homomorphism between the Kleene algebras PAT and REL. In particular, for $T \subseteq A$ and $U, V \subseteq A^{(+)}$,

$$\phi(T) = I_T \ , \tag{21}$$
$$\phi(U \bowtie V) = \phi(U) \,;\, \phi(V) \ , \tag{22}$$

and, for binary relation $R \subseteq A \bullet A$,

$$\phi(R^{\bowtie}) = R^* \ , \qquad \phi(R^{\Rightarrow}) = R^+ \ . \tag{23}$$

For $T \subseteq A$ and $U \subseteq A^{(+)}$ we calculate, using neutrality (7) and (21, 22),

$$T \,;\, \phi(U) = T \,;\, I_T \,;\, \phi(U) = T \,;\, \phi(T \bowtie U) \ . \tag{24}$$

We now apply these properties to *PAT* and *REL* to obtain some results about localising traversals of a directed graph to a subgraph. This can be expressed using a decomposition of the corresponding relation into a union of subrelations. Therefore we investigate the behaviour of the path closure with respect to a union of relations. Consider a set $T \subseteq A$ of nodes and binary relations $R, S \subseteq A \bullet A$. We want to calculate the set of all paths starting in T:

$$\begin{array}{ll} & T \bowtie (R \cup S)^{\bowtie} \\ = & \quad \{\![\text{ by (18) and distributivity }]\!\} \\ & T \bowtie R^{\bowtie} \ \cup \ T \bowtie R^{\bowtie} \bowtie S \bowtie (R \cup S)^{\bowtie} \ . \end{array}$$

If the second summand is $\emptyset$, we have succeeded in localising the traversal to the part of the graph described by R. So we want a criterion involving S that guarantees this. By strictness, a sufficient condition is $T \bowtie R^{\bowtie} \bowtie S = \emptyset$. Since sets of paths are computationally unwieldy, we want to find an equivalent characterisation in terms of smaller objects. We use the homomorphism ϕ to calculate

$$\begin{array}{ll} & T \bowtie R^{\bowtie} \bowtie S = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ since } \phi \text{ is a total mapping }]\!\} \\ & \phi(T \bowtie R^{\bowtie} \bowtie S) = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ by (21, 22, 23) and totality of } \phi \ ]\!\} \\ & I_T \,;\, R^* \,;\, S = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ by (6) }]\!\} \\ & \mathsf{lst}\,(I_T \,;\, R^*) \cap \mathsf{fst}\, S = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ by (2) and definition of } I_T \ ]\!\} \\ & T \,;\, R^* \cap \mathsf{fst}\, S = \emptyset \ . \end{array}$$

So we have shown

Lemma 2. *Assume $R, S \subseteq A \bullet A$ and $T \subseteq A$ such that $T \,;\, R^* \cap \mathsf{fst}\, S = \emptyset$. Then*

$$T \bowtie (R \cup S)^{\bowtie} = T \bowtie R^{\bowtie} \ .$$

Using again the homomorphism ϕ we obtain an analogous property for the reflexive transitive closure:

Corollary 3. *Assume* $R, S \subseteq A \bullet A$ *and* $T \subseteq A$ *such that* $T\,;R^* \cap \mathsf{fst}\,S = \emptyset$. *Then*

$$T\,;(R \cup S)^* \;=\; T\,;R^* \;\;.$$

Proof. We calculate

$$\begin{array}{ll} & T\,;(R \cup S)^* \\ = & \quad \{\![\text{ by (23, 24) }]\!\} \\ & T\,;\phi(T \bowtie (R \cup S)^{\bowtie}) \\ = & \quad \{\![\text{ by Lemma 2 }]\!\} \\ & T\,;\phi(T \bowtie R^{\bowtie}) \\ = & \quad \{\![\text{ by (23, 24) }]\!\} \\ & T\,;R^* \;\;. \end{array}$$

□

Since the path closure $R^{\bowtie}$ records all intermediate points of the paths, we need the strong precondition $T\,;R^* \cap \mathsf{fst}\,S = \emptyset$ which states that none of the intermediate points lies in the domain of S. However, the reflexive transitive closure $.^*$ abstracts from these intermediate points and so we can hope to find a more liberal precondition in this case.

By the fixpoint property of R^* and distributivity we have

$$T\,;R \;=\; T \;\cup\; T\,;R\,;R^* \;\;.$$

However, since on the right hand side T is already covered by the first summand, we can restrict our attention in the second summand to paths outside T. This is stated in

Lemma 4. *Assume* $R \subseteq A \bullet A$ *and* $T \subseteq A$. *Then*

$$T\,;R^* \;\subseteq\; T \;\cup\; T\,;R\,;(\overline{T} \bowtie R)^* \;\;,$$

where $\overline{T} \stackrel{\text{def}}{=} A \backslash T$.

Proof. Define $U \stackrel{\text{def}}{=} \overline{T} \bowtie R$. We use closure induction (Lemma 1(a)) on the continuous predicate

$$P[X] \;\stackrel{\text{def}}{\Leftrightarrow}\; T\,;X \;\subseteq\; T \;\cup\; T\,;R\,;U^* \;\;.$$

The base case $P[I_A]$ is immediate from neutrality of I_A. For the induction step assume $P[X]$. Then

$$\begin{array}{ll} & T\,;X\,;R \\ \subseteq & \quad \{\![\text{ by } P[X] \text{, monotonicity and distributivity }]\!\} \\ & T\,;R \;\cup\; T\,;R\,;U^*\,;R \\ = & \quad \{\![\text{ Boolean algebra, distributivity }]\!\} \\ & T\,;R \;\cup\; T\,;R\,;U^*\,;(\overline{T} \bowtie R) \;\cup\; T\,;R\,;U^*\,;(T \bowtie R) \end{array}$$

$$
\begin{array}{ll}
= & \{\![\text{ by definition of } U \text{ and (9) }]\!\} \\
& T\,;R \;\cup\; T\,;R\,;U^*\,;U \;\cup\; (T\,;R\,;U^* \bowtie T)\,;R \\
= & \{\![\text{ by } T\,;R\,;U^* \bowtie T \;\subseteq\; T, \text{ monotonicity, Boolean algebra }]\!\} \\
& T\,;R \;\cup\; T\,;R\,;U^*\,;U \\
= & \{\![\text{ by (16) }]\!\} \\
& T\,;R \;\cup\; T\,;R\,;U\,;U^* \\
= & \{\![\text{ distributivity, fixpoint property of } U^* \text{ }]\!\} \\
& T\,;R\,;U^* \;\;.
\end{array}
$$

□

Corollary 5. *Assume T and U as above. Then*

$$T\,;R^* \;=\; T \;\cup\; T\,;R\,;U^* \;\;.$$

Proof. Since $U \subseteq R$, monotonicity shows the inclusion ($\supseteq$); The reverse inclusion was shown in Lemma 4. □

4 Graph Algorithms

We now want to use our framework to derive two simple graph algorithms, viz. a reachability problem and cycle detection. As further applications, [32] calculates an algorithm computing the length of a shortest connecting path between two graph vertices, whereas [35] deals with an algorithm for finding Hamiltonian cycles.

4.1 A Simple Reachability Algorithm

We consider the following problem:

Given a directed graph, represented by a binary relation $R \subseteq A \bullet A$ over a finite set A of vertices, and a subset $S \subseteq A$, compute the set of vertices reachable by paths starting in S.

Hence we define

$$\mathit{reach}(S) \stackrel{\text{def}}{=} S\,;R^* \;\;.$$

The aim is to derive a recursive variant of *reach* from this specification. A termination case is given by $\mathit{reach}(\emptyset) = \emptyset\,;R^* = \emptyset$. Moreover, we can exploit Corollary 5:

$$S\,;R^* = S \;\cup\; S\,;R\,;U^* \;\;,$$

where $U \stackrel{\text{def}}{=} \overline{S} \bowtie R$ and $\overline{S} \stackrel{\text{def}}{=} A \backslash S$. However, since on the right hand side we have U^* rather than R^*, we cannot fold this into a recursive call to *reach*. To gain flexibility we use the technique of *generalisation* (see e.g. [33]): we introduce a second parameter T for the set that restricts R by defining

$$\mathit{re}(S,T) = S\,;(\overline{T} \bowtie R)^* \;\;.$$

By specialising this additional parameter we obtain an *embedding* of the original problem into the generalised one by $reach(S) = re(S, \emptyset)$. As the termination case we get again $re(\emptyset, T) = \emptyset$. Moreover, we calculate:

$$
\begin{array}{ll}
 & re(S,T) \\
= & \quad \{\![\text{ definition of } re \ ]\!\} \\
 & S\,;(\overline{T} \bowtie R)^* \\
= & \quad \{\![\text{ by Corollary 5 }]\!\} \\
 & S \ \cup \ S\,;(\overline{T} \bowtie R)\,;(\overline{S} \bowtie \overline{T} \bowtie R)^* \\
= & \quad \{\![\text{ by (9) }]\!\} \\
 & S \ \cup \ (S \bowtie \overline{T})\,;R\,;(\overline{S} \bowtie \overline{T} \bowtie R)^* \\
= & \quad \{\![\text{ by (5) and Boolean algebra }]\!\} \\
 & S \ \cup \ (S \backslash T)\,;R\,;(\overline{S \cup T} \bowtie R)^* \\
= & \quad \{\![\text{ definition of } re \ ]\!\} \\
 & S \ \cup \ re((S \backslash T)\,;R,\ S \cup T) \ .
\end{array}
$$

Altogether,

$$re(S,T) \ = \ \mathsf{if} \ S = \emptyset \ \mathsf{then} \ \emptyset \ \mathsf{else} \ S \ \cup \ re((S \backslash T)\,;R,\ S \ \cup \ T) \ \mathsf{fi} \ .$$

Note that by (5) the test $S = \emptyset$ can be expressed by the assertion $S\,;S$. We see that T keeps track of the vertices "already visited", while S is the set of vertices the successors of which still have to be visited.

To see whether this can be used as a recursive routine, we need to analyse the termination behaviour. An obvious idea is to inspect the cardinalities of the sets involved. Whereas the first parameter of *re* can shrink and grow according to the varying outdegrees of nodes, the second parameter never shrinks and is bounded from above by $|A|$. The cardinality actually increases unless $S \subseteq T$. However, in that latter case we have $S \backslash T = \emptyset$, so that the recursion moves into the termination case anyway. So the cardinality of the second parameter can indeed be used as a termination function. By standard techniques using an accumulator and associativity of $\cup$ (see e.g. [33]) one can finally transform this into a tail recursion and from there into loop form.

4.2 Cycle Detection

Formal Specification. Consider again a finite set A of vertices and a binary relation $R \subseteq A \bullet A$. The problem is now:

Determine whether R contains a **cyclic path***, i.e., a proper path in which some node occurs twice.*

The set of all proper paths is given by the proper path closure $R^{\Rightarrow}$. We now investigate the set of all proper paths that begin and end in the same node, viz.

$$cyc(R) \stackrel{\text{def}}{=} \bigcup_{a \in A} a \bowtie R^{\Rightarrow} \bowtie a \ .$$

Obviously, R contains a cyclic path iff $cyc(R) \neq \emptyset$. However, $cyc(R)$ will be infinite in case R actually contains a cycle, and so this test cannot be evaluated directly. Rather we have to find equivalent characterisations of the problem. Again we can use the homomorphism ϕ from (19) to calculate

$$\begin{array}{ll} & \phi(cyc(R)) \\ = & \quad \{\![\text{ by definition of } cyc \text{ and continuity of } \phi \ ]\!\} \\ & \bigcup_{a \in A} \phi(a \bowtie R^{\Rightarrow} \bowtie a) \\ = & \quad \{\![\text{ by (20, 23) }]\!\} \\ & \bigcup_{a \in A} a \bullet a \cap R^+ \\ = & \quad \{\![\text{ distributivity }]\!\} \\ & (\bigcup_{a \in A} a \bullet a) \cap R^+ \\ = & \quad \{\![\text{ definition of } I \ ]\!\} \\ & I_A \cap R^+ \ . \end{array}$$

Since ϕ is a total mapping, we infer

Lemma 6. $cyc(R) \neq \emptyset \ \Leftrightarrow \ R^+ \cap I_A \neq \emptyset$.

For finite A also R^+ and I_A are finite, and so progress has been made. Still R^+ can be a large object and so a direct evaluation is not advisable. Fortunately we can also show

Lemma 7. *The following statements are equivalent:*
(a) $cyc(R) \neq \emptyset$.
(b) $R^{|A|} \neq \emptyset$.
(c) $R^{|A|} ; A \neq \emptyset$.
(d) $A ; R^{|A|} \neq \emptyset$.

Proof. (a) $\Rightarrow$ (b) Assume $w \in cyc(R)$. Then $A \neq \emptyset$. Moreover, $\emptyset \neq w \bowtie w \in cyc(R)$. Now a straightforward induction shows $\emptyset \neq \bigbowtie_{i=1}^{m} w$ for all $m \geq 1$. Since $cyc(R) \subseteq R^{\Rightarrow}$, there is an $n \geq 1$ with $w \in \bigbowtie_{j=1}^{n} R$ and hence, by iterated application of (22), $\phi(w) \in \phi(\bigbowtie_{j=1}^{n} R) = R^n$. By totality of ϕ therefore also $\emptyset \neq \phi(\bigbowtie_{i=1}^{m} w) = \phi(w)^m \in R^{m \cdot n}$. Setting $m = |A|$ and using $n \geq 1$ and (12) we obtain the claim.

(b) $\Rightarrow$ (a) Let $a, b \in A$ with $a \bullet b \in R^{|A|}$. We have $a \bullet b \cap R^{|A|} = \phi(a \bowtie (\bigbowtie_{i=1}^{|A|} R) \bowtie b)$ by (20). However, every path in $\bigbowtie_{i=1}^{|A|} R$ has $|A| + 1$ nodes and thus, by the pigeonhole principle, contains at least one node twice. This shows $cyc(R) \neq \emptyset$.

(b) $\Leftrightarrow$ (c) $\Leftrightarrow$ (d) is trivial. □

Among these equivalent formulations, (c) and (d) seem computationally most promising, since they deal with unary relations which in general are much smaller objects than binary ones. We choose (d) as our starting point and specify our problem as

$$\mathit{hascycle} \stackrel{\text{def}}{=} (A\,;R^{|A|} \neq \emptyset)\ .$$

An Iteration Principle. To compute $A\,;R^{|A|}$ we define $A_i \stackrel{\text{def}}{=} A\,;R^i$ and use the properties of the powers of R:

$$\begin{array}{rcl} A_0 = A\,;R^0 & = A\,;I_A & = A\ , \\ A_{i+1} = A\,;R^{i+1} & = A\,;(R^i\,;R) & = (A\,;R^i)\,;R = A_i\,;R\ . \end{array}$$

The associated function $f : X \mapsto X\,;R$ is monotonic. We now state a general theorem about monotonic functions on noetherian partial orders. A partial order $(M,\leq)$ is called **noetherian** if each of its nonempty subsets has a minimal element with respect to $\leq$. An element $x \in S \subseteq M$ is **minimal** in S if $y \in S$ and $y \leq x$ imply $y = x$. Viewing a function $f : M \to M$ as a binary relation, we can form its closure f^*. Then, for $x \in M$, we have $x\,;f^* = \{f^i(x) : i \in \mathbb{N}\}$.

Theorem 8. *Let $(M,\leq)$ be a noetherian partial order and $f : M \to M$ a monotonic total function.*
(a) *If for $x \in M$ we have $f(x) \leq x$ then $x_\infty = \mathsf{glb}\ (x\,;f^*)$ exists and is a fixpoint of f. Moreover, it is the only fixpoint of f in $x\,;f^*$.*
(b) *Assume x as in (a) and $y \in M$ with $x_\infty \leq y \leq x$. Then also $y_\infty = \mathsf{glb}\ (y\,;f^*)$ exists and $y_\infty = x_\infty$.*
(c) *If M has a greatest element $\top$, then $\top_\infty$ exists and is the greatest fixpoint of f.*

For the proof see [32]. A similar theorem has been stated in [10].

To actually calculate x_∞ we define a function *inf* by

$$\mathit{inf}(y) \stackrel{\text{def}}{=} (x_\infty \leq y \leq x) \bullet x_\infty\ ,$$

which for fixed x determines x_∞ using an upper bound y. We have the embedding $x_\infty = \mathit{inf}(x)$. Now from the above theorem and the fixpoint property of f^* the following recursion is immediate:

$$\mathit{inf}(y) = (x_\infty \leq y \leq x)\ \bullet\ \mathsf{if}\ y = f(y)\ \mathsf{then}\ y\ \mathsf{else}\ \mathit{inf}(f(y))\ \mathsf{fi}\ .$$

Since M is noetherian, this recursion terminates for every y satisfying $f(y) \leq y$, because monotonicity then also shows $f(f(y)) \leq f(y)$, so that in each recursive call the parameter decreases properly. In particular, the call $\mathit{inf}(x)$ terminates. This algorithm is an abstraction of many iteration methods on finite sets.

A Recursive Solution. We now return to the special case of cycle detection. By finiteness of A the partial order $(\mathcal{P}(A), \subseteq)$ is noetherian with greatest element A. Therefore A_∞ exists. Moreover, we have

Corollary 9. $A_{|A|} = A_\infty$.

Proof. The length of any properly descending chain in $\mathcal{P}(A)$ is at most $|A|+1$. Hence we have $A_{|A|+1} = A_{|A|}$ and thus $A_{|A|} = A_\infty$. □

So we have reduced our task to checking whether $A_\infty \neq \emptyset$, i.e., whether $inf(A) \neq \emptyset$. For our special case the recursion for *inf* reads (omitting the trivial part $W \subseteq A$)

$$inf(W) \;=\; (A_\infty \subseteq W) \;\bullet\; \mathsf{if}\; W = W ; R \;\mathsf{then}\; W \;\mathsf{else}\; inf(W ; R) \;\mathsf{fi}\;\; .$$

We want to improve this by avoiding the computation of $W ; R$. By the above considerations we may strengthen the assertion of *inf* by adding the conjunct $W ; R \subseteq W$. Thus we only need to worry about the difference between W and $W ; R$. We define

$$src(W) \;\stackrel{\text{def}}{=}\; W \backslash (W ; R) \;\; .$$

Since $W ; R$ is the set of successors of W under R, this is the set of **sources** of W, i.e., the set of nodes in W which do not have a predecessor in W.

Now, assuming $W ; R \subseteq W$, we have $W = W ; R \Leftrightarrow src(W) = \emptyset$ and $W ; R = W \backslash src(W)$, so that we can rewrite *inf* into

$$\begin{aligned} inf(W) = (A_\infty \subseteq W \wedge W ; R \subseteq W) \;\bullet\; \\ \mathsf{if}\; src(W) = \emptyset \;\mathsf{then}\; W \;\mathsf{else}\; inf(W \backslash src(W)) \;\mathsf{fi}\;\; . \end{aligned}$$

This is an improvement in that $src(W)$ usually will be small compared to W; moreover, the computation of $src(W)$ can be facilitated by a suitable representation of R.

Plugging this into our original problem of cycle recognition we obtain

$$hascycle \;=\; hcy(A) \;\; ,$$

where

$$\begin{aligned} hcy(W) \;=\; (A_\infty \subseteq W \wedge W ; R \subseteq W) \;\bullet\; \\ \mathsf{if}\; src(W) = \emptyset \;\mathsf{then}\; W \neq \emptyset \;\mathsf{else}\; hcy(W \backslash src(W)) \;\mathsf{fi}\;\; , \end{aligned}$$

which is one of the classical algorithms and works by successive removal of sources (see e.g. [4]). Note that Lemma 7(c) suggests a dual specification to the one we have used; replaying our development for it would lead to an algorithm that works by successive removal of sinks. From the algorithm above we derive in [32] a more efficient one, in which the source sets are computed from an incrementally adjusted vector of in-degrees of the graph vertices.

5 Pointer Algorithms

5.1 Introduction

It is well-known that algorithms involving pointers are both difficult to write and to verify. One reason is that, due to the 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 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. In this section 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.

5.2 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 modeled by a binary **access relation** $R \subseteq A \bullet A$ where our alphabet A is now taken as the set of records (represented, say, by their initial addresses). A is not restricted to be finite; this way also (conceptually) unbounded storage can be modeled.

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 $\mathit{left}, \mathit{right} : A \to A$ and set $R \stackrel{\text{def}}{=} \mathit{left} \cup \mathit{right}$.

Given a set $T \subseteq A$ of entry records, we can follow the pointers to other records. This is modeled by considering $T \bowtie R^{\bowtie}$ or its ϕ-abstraction $T\,;R^*$. The part of the pointer structure reachable from T is

$$\mathit{from}(T,R) \stackrel{\text{def}}{=} T\,;R^* \bowtie R \ ,$$

i.e., the restriction of R to the records reachable from T. Plugging in our reachability algorithm from Section 4.1 we obtain

$$\mathit{from}(T,R) = \mathit{fr}(T,\emptyset) \ ,$$

where

$$\begin{aligned}\mathit{fr}(T,U) = \ &\mathsf{if}\ T = \emptyset\ \mathsf{then}\ \emptyset \\ &\mathsf{else}\ T \bowtie R \ \cup\ \mathit{fr}((T \backslash U)\,;R,\ T \cup U)\ \mathsf{fi}\ .\end{aligned}$$

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 $\Box \in A$ be a distinguished element, called the **anchor**. The elements of $A \backslash \Box$ are called **proper** records. A **state** is a finite relation $R \subseteq A \bullet A$ such that $\Box \notin \mathsf{fst}\,R$. 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.

5.3 Algebraic Properties of Overwriting

An essential operation on pointer structures is their selective updating. To model this we define the operation of **overwriting** one relation with another one (see e.g. [16, 17, 18, 34] for the special case of maps). Let A now be arbitrary. Then, given binary relations $R, S \subseteq A \bullet A$, we define

$$R \mid S \stackrel{\text{def}}{=} R \cup \overline{\mathsf{fst}\, R} \rhd S \ .$$

Hence, a pair $a \bullet b$ is in $R \mid S$ iff it is in R or a is not in the domain of R and $a \bullet b$ is in S. In other words, $R \mid S$ results from S by changing the values associated with the "arguments" from A according to the prescription of R (if any). For example, if S is a map then $(a \bullet b) \mid S$ "updates" S to make b the value corresponding to a. This operation will be our main tool for describing selective updating of storage structures. We use the convention that $\mid$ binds stronger than all set-theoretic operations.

We call relations R and S **compatible** if one of the following equivalent properties holds:

$$\begin{gathered} R \mid S = R \cup S = S \mid R \ , \\ R \mid S = S \mid R \ , \\ (\mathsf{fst}\, S) \rhd R = (\mathsf{fst}\, R) \rhd S \ , \\ (\mathsf{fst}\, S) \rhd R = R \cap S = (\mathsf{fst}\, R) \rhd S \ . \end{gathered}$$

In the sequel we list a number of useful properties of overwriting; for proofs and further properties see [30].

1. *Monoid properties:*
$$\begin{gathered} \emptyset \mid R = R = R \mid \emptyset \ , \\ (R \mid S) \mid T = R \mid (S \mid T) \ . \end{gathered}$$
2. *Localisation:*
$$\mathsf{fst}\, R \cap \mathsf{fst}\, S = \emptyset \Rightarrow R \mid (S \cup T) = S \cup R \mid T \ , \tag{25}$$
$$\mathsf{fst}\, R \cap \mathsf{fst}\, T = \emptyset \Rightarrow (R \cup S) \mid T = R \cup S \mid T \ . \tag{26}$$
These properties allow localising side effects to that part of a pointer structure really affected.
3. *Sequentialisation:*
$$(R \cup S) \mid T = R \mid (S \mid T) \ \text{ if } R \text{ and } S \text{ are compatible.} \tag{27}$$
4. *Idempotence:*
$$S \mid T = S \mid R \mid T \ \text{ if } R \subseteq S \text{ and } R \text{ and } S \text{ are compatible.}$$
This is an immediate consequence of sequentialisation. When viewed operationally, the direction from left to right allows early overwriting of parts of a map.
5. *Annihilation:*
$$S \mid T = T \Rightarrow (R \cup S) \mid T = R \mid T \ \text{ if } R \text{ and } S \text{ are compatible.}$$

Now we consider the special case of maps. A binary relation $R \subseteq A \bullet A$ is a **(partial) map** if $a \bullet b \in R \wedge a \bullet c \in R \Rightarrow b = c$ or, more succinctly, if $R^{-1} ; R \subseteq I_A$. We write $R : A \rightarrow A$ to indicate that R is a map. The following properties are important for maps:

1. *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.
2. *Submap:*
   Let S be a map and $R \subseteq S$. Then R is a map as well and R and S are compatible.
3. *Idempotence:*
   Let S be a map and T a relation. Then
   $$R \subseteq S \Rightarrow S \mid T = S \mid R \mid T \ . \tag{28}$$
4. *Annihilation:*
   Let R, S, T be maps such that $S \subseteq T$ and R and S are compatible. Then
   $$(R \cup S) \mid T = R \mid T \ . \tag{29}$$

5.4 Overwriting and Paths

Using our general path properties we can show that overwriting outside a certain substructure does not affect that substructure:

Lemma 10. *Assume $R, S \subseteq A \bullet A$ and $T \subseteq A$ such that $T ; R^* \cap \mathsf{fst}\, S = \emptyset$. Then*

$$T \bowtie (S \mid R)^{\bowtie} = T \bowtie R^{\bowtie} \ .$$

Proof. We calculate

$$\begin{array}{ll} & T \bowtie (S \mid R)^{\bowtie} \\ = & \quad \{\![\text{ definition of } \mid \ ]\!\} \\ & T \bowtie (S \ \cup \ \overline{\mathsf{fst}\, S} \bowtie R)^{\bowtie} \\ = & \quad \{\![\text{ by Lemma 2 and assumption, since } T ; (\overline{\mathsf{fst}\, S} \bowtie R)^* \subseteq T ; R^* \\ & \quad \text{by monotonicity }]\!\} \\ & T \bowtie (\overline{\mathsf{fst}\, S} \bowtie R)^{\bowtie} \\ = & \quad \{\![\text{ by Lemma 2, since } T ; R^* \cap \mathsf{fst}\, S \bowtie R = \emptyset \text{ by (5, 1) and assumption }]\!\} \\ & T \bowtie ((\mathsf{fst}\, S) \bowtie R \ \cup \ \overline{\mathsf{fst}\, S} \bowtie R)^{\bowtie} \\ = & \quad \{\![\text{ distributivity, Boolean algebra }]\!\} \\ & T \bowtie (A \bowtie R)^{\bowtie} \\ = & \quad \{\![\text{ neutrality of } A \ ]\!\} \\ & T \bowtie R^{\bowtie} \ . \end{array}$$

□

Corollary 11. *Assume* $R, S \subseteq A \bullet A$ *and* $T \subseteq A$ *such that* $T ; R^* \cap \mathsf{fst}\, S = \emptyset$. *Then*

$$T ; (S \mid R)^* = T ; R^* \ .$$

Proof. We use the homomorphism ϕ of (19) to calculate

$$\begin{array}{ll} & T ; (S \mid R)^* \\ = & \quad \{\![\text{ by (24) }]\!\} \\ & T ; \phi(T \bowtie (S \mid R)^{\bowtie}) \\ = & \quad \{\![\text{ by the previous lemma }]\!\} \\ & T ; \phi(T \bowtie R^{\bowtie}) \\ = & \quad \{\![\text{ by (24) }]\!\} \\ & T ; R^* \ . \end{array}$$

□

5.5 Chains

We return now to pointer structures and deal first with the special case of **chains**, i.e., with (finite) cycle-free singly linked lists. Since every record has at most one link, the associated access relation will actually be a map $next : A \to A$. 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 is the head record, followed by the others in the order of traversal. Since there is no cycle, the sequence is repetition-free.

More formally, we call a record a a **(chain) head** in a state $next : A \to A$ if the set $a \bowtie next^{\bowtie}$ of paths starting from a in $next$ is finite. This condition characterises the situation that no cycle can be reached from a in $next$. Note that $\Box$ always is a head, since $\Box \bowtie next^{\bowtie} = \Box$.

We call a record a **anchored** in a state $next : A \to A$ if $\Box \in a ; next^*$. Since $next$ is a map this implies that a is a head. Note that $\Box$ is anchored in every state.

An Abstraction Function. We define an operator $\triangleleft$ which for anchored a constructs the sequence $a \triangleleft next$ of proper records reachable from a in state $next : A \to A$:

$$a \triangleleft next \stackrel{\text{def}}{=} (a \bowtie next^{\bowtie}) ; \Box \ .$$

This selects all paths starting in a and ending in $\Box$ and removes the trailing $\Box$. Hence for non-anchored a we have $a \triangleleft next = \emptyset$. We calculate:

$$\begin{array}{ll} & a \triangleleft next \\ = & \quad \{\![\text{ definition of } \triangleleft \ ]\!\} \\ & (a \bowtie next^{\bowtie}) ; \Box \end{array}$$

$$
\begin{aligned}
&= \quad \{\![\text{ by fixpoint property (13) of } next^{\bowtie} \text{ and distributivity }]\!\} \\
&\quad (a \bowtie A)\,;\Box \;\cup\; (a \bowtie next \bowtie next^{\bowtie})\,;\Box \\
&= \quad \{\![\text{ by neutrality of } A \ ]\!\} \\
&\quad a\,;\Box \;\cup\; (a \bowtie next \bowtie next^{\bowtie})\,;\Box \\
&= \quad \{\![\text{ by (8) }]\!\} \\
&\quad a\,;\Box \;\cup\; (a \bullet a\,; next \bowtie next^{\bowtie})\,;\Box \\
&= \quad \{\![\text{ by (1) }]\!\} \\
&\quad a\,;\Box \;\cup\; a \bullet ((a\,; next \bowtie next^{\bowtie})\,;\Box) \\
&= \quad \{\![\text{ definition of } \lhd \ ]\!\} \\
&\quad a\,;\Box \;\cup\; a \bullet ((a\,; next) \lhd next) \ .
\end{aligned}
$$

The first summand suggests a case analysis according to whether $a = \Box$. If so, $a; next = \emptyset$, since *next* is a state, and hence the second summand vanishes by strictness. Otherwise the first summand disappears. Thus we have the following recursion relation for $\lhd$:

$$a \lhd next \;=\; \mathsf{if}\ a \;=\; \Box\ \mathsf{then}\ \varepsilon\ \mathsf{else}\ a \bullet ((a\,; next) \lhd next)\ \mathsf{fi}\ . \tag{30}$$

Let us now study the behaviour of $\lhd$ under overwriting. From our general properties of overwriting we can derive that overwriting outside of a chain does not change the chain:

Corollary 12. $a \lhd next \;=\; a \lhd (u \bullet v) \mid next$ *provided* $u \notin a\,; next^*$.

Proof. We calculate

$$
\begin{aligned}
&\quad a \lhd (u \bullet v) \mid next \\
&= \quad \{\![\text{ definition }]\!\} \\
&\quad (a \bowtie ((u \bullet v) \mid next)^{\bowtie})\,;\Box \\
&= \quad \{\![\text{ by Lemma 10 }]\!\} \\
&\quad (a \bowtie next^{\bowtie})\,;\Box \\
&= \quad \{\![\text{ definition }]\!\} \\
&\quad a \lhd next \ .
\end{aligned}
$$

□

Moreover, we can specialise our general induction principles to chains. We first show

Lemma 13. *Let P be a continuous predicate on $A^{(*)}$ and next a state such that $P[\varepsilon]$, $P[next\,;\Box]$ and $\forall\ S \subseteq A^{(*)} : P[S] \;\Rightarrow\; P[next \bowtie S]$ hold. Then $P[next^{\bowtie}\,;\Box]$ holds as well.*

Proof. A straightforward induction shows $P[(\overset{n}{\underset{i=1}{\bowtie}}\ next)\,;\Box]$ for all $n \in \mathbb{N}$. Now the claim is immediate from Kleene's approximation (14) and continuity of P. □

Call a predicate P on $A^{(*)}$ **additive** if for all sets $\mathcal{S} \subseteq \mathcal{P}(A^{(*)})$ we have

$$P[\cup\mathcal{S}] \Leftrightarrow \bigwedge_{S\in\mathcal{S}} P[S] \ .$$

Any additive predicate is also continuous. Note that for additive P

$$S \subseteq T \Rightarrow (P[T] \Rightarrow P[S]) \tag{31}$$

and $P[\emptyset]$ hold.

Corollary 14 (Chain Induction). *Let $next : A \to A$ be a state and P an additive predicate on $A^{(*)}$ such that $P[\varepsilon]$, $P[next\,;\Box]$ and $\forall\, S \subseteq A^{(*)} : P[S] \Rightarrow P[next \bowtie S]$ hold. Then $\forall\, a \in A : P[a \lhd next]$ holds as well.*

Proof. By monotonicity and the definition of $\lhd$ we have $a \lhd next \subseteq next^{\bowtie}\,;\Box$. Now the claim follows from the previous Lemma 13 and (31). □

A Representation Function. The operator $\lhd$ is an abstraction function from states to record sequences. We now deal with a corresponding representation function. Given a sequence s of records we define its chaining $chain(s) \subseteq A \bullet A$ by

$$\begin{aligned} chain(\varepsilon) &\stackrel{\text{def}}{=} \emptyset \ , \\ chain(a \bullet s) &\stackrel{\text{def}}{=} a \bullet first(s) \ \cup \ chain(s) \ , \end{aligned} \tag{32}$$

where, for $s \in (A\backslash\Box)^{(*)}$,

$$first(s) \stackrel{\text{def}}{=} \mathsf{if}\ s = \varepsilon\ \mathsf{then}\ \Box\ \mathsf{else}\ \mathsf{fst}\ s\ \mathsf{fi} \ .$$

If s is a repetition-free sequence of proper records, then $chain(s)$ is a map. From the definition it is obvious that

$$\mathsf{fst}\ chain(s) \ = \ \mathsf{set}\ s \tag{33}$$

(see Section 2.3 for the definition of set). Moreover, for $a \in A\backslash\Box$ we have

$$chain(a) \ = \ a \bullet \Box \ ,$$

meaning that chaining a single proper record yields a state consisting just of that record with the anchor $\Box$ as its "successor".

We now investigate the behaviour of *chain* with respect to concatenation. First we note that for sequences u, v of proper records such that $u \bullet v$ is repetition-free we have by (33)

$$\mathsf{fst}\ chain(u) \cap \mathsf{fst}\ chain(v) \ = \ \emptyset \ , \tag{34}$$

so that $chain(u)$ and $chain(v)$ are compatible. Now we can show

Lemma 15. *Assume $s, t \in (A\backslash\Box)^{(*)}$ and $a \in A\backslash\Box$ such that $s \bullet a \bullet t$ is repetition-free. Then*

$$chain(s \bullet a \bullet t) \ = \ chain(a \bullet t) \mid chain(s \bullet a) \ .$$

Proof. We use induction on s. For the case $s = \varepsilon$ we calculate

$$
\begin{array}{ll}
& \mathit{chain}(s \bullet a \bullet t) \\
= & \quad \{\![\text{ neutrality }]\!\} \\
& \mathit{chain}(a \bullet t) \\
= & \quad \{\![\text{ by (32) }]\!\} \\
& a \bullet \mathit{first}(t) \ \cup \ \mathit{chain}(t) \\
= & \quad \{\![\text{ definition of } | \ ]\!\} \\
& a \bullet \mathit{first}(t) \mid (a \bullet \Box) \ \cup \ \mathit{chain}(t) \\
= & \quad \{\![\text{ localisation (26), since } a \notin \mathsf{set}\, t = \mathsf{fst}\, \mathit{chain}(t) \text{ by (34) }]\!\} \\
& (a \bullet \mathit{first}(t) \ \cup \ \mathit{chain}(t)) \mid (a \bullet \Box) \\
= & \quad \{\![\text{ definition of } \mathit{chain} \ ]\!\} \\
& \mathit{chain}(a \bullet t) \mid \mathit{chain}(a) \\
= & \quad \{\![\text{ neutrality }]\!\} \\
& \mathit{chain}(a \bullet t) \mid \mathit{chain}(s \bullet a) \ .
\end{array}
$$

Assume now $s = b \bullet u$ and that the claim holds for $u \bullet a \bullet t$. We calculate

$$
\begin{array}{ll}
& \mathit{chain}(s \bullet a \bullet t) \\
= & \quad \{\![\text{ definition of } \mathit{chain} \ ]\!\} \\
& b \bullet \mathit{first}(u \bullet a \bullet t) \ \cup \ \mathit{chain}(u \bullet a \bullet t) \\
= & \quad \{\![\text{ definition of } \mathit{first}, \text{ assumption }]\!\} \\
& b \bullet \mathit{first}(u \bullet a) \ \cup \ \mathit{chain}(a \bullet t) \mid \mathit{chain}(u \bullet a) \\
= & \quad \{\![\text{ localisation (25), since } b \notin \mathsf{set}\, a \bullet t \text{ by (34) }]\!\} \\
& \mathit{chain}(a \bullet t) \mid (b \bullet \mathit{first}(u \bullet a) \ \cup \ \mathit{chain}(u \bullet a)) \\
= & \quad \{\![\text{ by (32) }]\!\} \\
& \mathit{chain}(a \bullet t) \mid \mathit{chain}(s \bullet a) \ .
\end{array}
$$

□

From this we obtain

Corollary 16. *For a, s, t as above we have*

$$\mathit{chain}(s \bullet a \bullet t) \ \supseteq \ a \bullet \mathit{first}(t) \ .$$

Proof. Immediate from the previous lemma, (32) and $R \subseteq R \mid S$. □

Moreover we have

Lemma 17. *Let a be anchored in next. Then*
(a) $\mathit{first}(a \lhd \mathit{next}) = a$.
(b) $\mathit{chain}(a \lhd \mathit{next}) \subseteq \mathit{next}$.
(c) $\mathit{first}(s) \lhd \mathit{chain}(s) = s$.

Proof. (a) follows from (30) by distinguishing the cases $a = \Box$ and $a \neq \Box$.
(b) is a straightforward chain induction (Corollary 14) with the additive predicate

$$P[S] \stackrel{\text{def}}{\Leftrightarrow} \bigcup_{s \in S} \mathit{chain}(s) \subseteq \mathit{next} \ .$$

(c) is a straightforward induction on the length of s.

□

5.6 Concatenation of Chains "in Situ"

Specification and First Explicit Solution. We now want to specify and develop an algorithm for concatenating two non-overlapping anchored chains "in situ". The problem is as follows:

Given two anchored heads a, b in a state next, we want to form a new state in which the concatenation of the chains of a and b is overwritten onto the same *set of proper records; moreover, the order of traversal within each chain should be preserved, and all records in the chain of a should precede all records in the chain of b.*

An expression achieving this is

$$\mathit{chain}((a \lhd \mathit{next}) \bullet (b \lhd \mathit{next})) \mid \mathit{next} \ ,$$

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

To make the application of chain sensible we need the precondition that $(a \lhd \mathit{next}) \bullet (b \lhd \mathit{next})$ is repetition-free provided $(a \lhd \mathit{next})$ and $(b \lhd \mathit{next})$ are. We define

$$\mathit{disjoint}(a, b, \mathit{next}) \stackrel{\text{def}}{=} ((a \,;\, \mathit{next}^*) \cap (b \,;\, \mathit{next}^*) = \Box) \ .$$

This states that a and b are anchored heads in *next* and that the chains starting from a and b are disjoint except for the anchor, which also implies that $(a \lhd \mathit{next}) \bullet (b \lhd \mathit{next})$ is repetition-free.

Thus our formal specification of the concatenation function is now

$$\mathit{conc}(a, b, \mathit{next}) \stackrel{\text{def}}{=} \mathit{disjoint}(a, b, \mathit{next}) \bullet \mathit{chain}((a \lhd \mathit{next}) \bullet (b \lhd \mathit{next})) \mid \mathit{next} \ .$$

From this specification we now want to develop a direct recursion without the "detour" through sequences. We try to obtain it again by the unfold/fold technique. Guided by the recursion relation (30) for $\lhd$ we perform a case analysis on a.

Case 1: $a = \Box$. Then

$$
\begin{array}{ll}
 & \mathit{chain}((a \lhd \mathit{next}) \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= & \quad \{\![\text{ by (30) and neutrality }]\!\} \\
 & \mathit{chain}(b \lhd \mathit{next}) \mid \mathit{next} \\
= & \quad \{\![\text{ by annihilation (29), since } \mathit{chain}(b \lhd \mathit{next}) \subseteq \mathit{next} \text{ by Lemma 17(b) }]\!\} \\
 & \mathit{next} \ .
\end{array}
$$

Case 2: $a \neq \Box$. Then

$$
\begin{array}{ll}
 & \mathit{chain}((a \lhd \mathit{next}) \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= & \quad \{\![\text{ by (30) }]\!\} \\
 & \mathit{chain}(a \bullet ((a \,;\, \mathit{next}) \lhd \mathit{next}) \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= & \quad \{\![\text{ by (32) }]\!\} \\
 & (a \bullet \mathit{first}(((a \,;\, \mathit{next}) \lhd \mathit{next}) \bullet (b \lhd \mathit{next})) \ \cup \ \mathit{chain}(((a \,;\, \mathit{next}) \lhd \mathit{next}) \bullet (b \lhd \mathit{next}))) \mid \mathit{next} \\
= & \quad \{\![\text{ abbreviation }]\!\} \\
 & (*) \ .
\end{array}
$$

To simplify this expression we again use the recursion relation (30) on the first call to $\lhd$ and perform a case analysis on $a \,;\, \mathit{next}$.

Case 2.1: $a \,;\, \mathit{next} = \Box$. We calculate:

$$
\begin{array}{ll}
 & (*) \\
= & \quad \{\![\text{ by (30) and neutrality }]\!\} \\
 & (a \bullet \mathit{first}(b \lhd \mathit{next}) \ \cup \ \mathit{chain}(b \lhd \mathit{next})) \mid \mathit{next} \\
= & \quad \{\![\text{ by Lemma 17(a) }]\!\} \\
 & (a \bullet b \ \cup \ \mathit{chain}(b \lhd \mathit{next})) \mid \mathit{next} \\
= & \quad \{\![\text{ by annihilation (29), since } \mathit{chain}(b \lhd \mathit{next}) \subseteq \mathit{next} \text{ by Lemma 17(b) }]\!\} \\
 & (a \bullet b) \mid \mathit{next} \ .
\end{array}
$$

Case 2.2: $a \,;\, \mathit{next} \neq \Box$. Setting $s \stackrel{\text{def}}{=} a \,;\, \mathit{next} \bullet ((a \,;\, \mathit{next} \,;\, \mathit{next}) \lhd \mathit{next})$ we calculate

$$
\begin{array}{ll}
 & (*) \\
= & \quad \{\![\text{ by (30) }]\!\} \\
 & \mathit{chain}(a \bullet s \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= & \quad \{\![\text{ by (32) }]\!\} \\
 & ((a \bullet a \,;\, \mathit{next}) \ \cup \ \mathit{chain}(s \bullet (b \lhd \mathit{next}))) \mid \mathit{next} \\
= & \quad \{\![\text{ by annihilation (29), since } a \bullet a \,;\, \mathit{next} \subseteq \mathit{next} \text{ }]\!\} \\
 & \mathit{chain}(s \bullet (b \lhd \mathit{next})) \mid \mathit{next}
\end{array}
$$

$$
\begin{array}{ll}
= & \{\![\text{ by definition of } s \text{ and (30) }]\!\} \\
& \mathit{chain}(((a\,;\mathit{next}) \triangleleft \mathit{next}) \bullet (b \triangleleft \mathit{next})) \mid \mathit{next} \\
= & \{\![\text{ fold } \mathit{conc} \ ]\!\} \\
& \mathit{conc}(a\,;\mathit{next},\, b\,, \mathit{next}) \ .
\end{array}
$$

For the correctness of the folding step we also need to check the validity of the assertion of *conc* for the recursive call. We calculate, assuming $a \triangleleft \mathit{next} \neq \varepsilon$, which implies $a \neq \Box$:

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

Thus, $\mathit{disjoint}(a, b, \mathit{next})$ implies $\mathit{disjoint}(a\,;\mathit{next},\, b,\, \mathit{next})$ as required.

Altogether we obtain

$$
\begin{array}{l}
\mathit{conc}(a, b, \mathit{next}) \ = \ \mathsf{if}\ a = \Box\ \mathsf{then}\ \mathit{next} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathsf{if}\ a\,;\mathit{next} = \Box\ \mathsf{then}\ (a \bullet b) \mid \mathit{next} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ \mathit{conc}(a\,;\mathit{next}, b, \mathit{next})\ \mathsf{fi}\ \mathsf{fi} \ .
\end{array}
$$

Termination follows from the fact that a is anchored in *next*.

Introducing Selective Updating. Since we have even obtained a tail-recursive version, we are already very close to an imperative program. By standard transformation techniques one shows that the assignment

$$\mathit{next} := \mathit{conc}(\mathit{next}, a, b)$$

is equivalent to

$$
\begin{array}{l}
\mathsf{if}\ a = \Box\ \mathsf{then}\ \mathit{next} := \mathit{next} \\
\qquad \mathsf{else}\ \ \mathit{va} := a\ ; \\
\qquad\quad \mathsf{while}\ (\mathit{va}\,;\mathit{next}) \neq \Box\ \ \mathsf{do}\ \ \mathit{va} := (\mathit{va}\,;\mathit{next})\ \ \mathsf{od}; \\
\qquad\quad \mathit{next} := (\mathit{va} \bullet b) \mid \mathit{next} \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{fi} \quad .
\end{array}
$$

Since *next* is modified by the assignment anyway, we can operate on it directly. Also b never changes, and so only for a an auxiliary variable is needed. If we write the assignment

$$next := (va \bullet b) \mid next$$

in a Pascal-like way as

$$va\uparrow.next := b \ ,$$

we see that we actually have derived a version with selective updating.

In the derivation we have not made use of any assumptions about absence of sharing. Indeed, if in *next* there are pointers from other data structures to (parts of) the lists headed by a and b, 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.

A Variation. We briefly discuss an alternative derivation for the same problem. In Cases 2 and 3 above we know by (30) that $a \lhd next \neq \varepsilon$ and thus

$$a \lhd next \ = \ s \bullet c \tag{35}$$

for some $s \in (A \backslash \Box)^{(*)}$ and $c \in A \backslash \Box$. Hence we calculate

$$\begin{array}{ll}
 & chain((a \lhd next) \bullet (b \lhd next)) \mid next \\
= & \quad \{\![\text{ by (35) }]\!\} \\
 & chain(s \bullet c \bullet (b \lhd next)) \mid next \\
= & \quad \{\![\text{ by Lemma 15 and associativity }]\!\} \\
 & chain(c \bullet (b \lhd next)) \mid chain(s \bullet c) \mid next \\
= & \quad \{\![\text{ by (35) }]\!\} \\
 & chain(c \bullet (b \lhd next)) \mid chain(a \lhd next) \mid next \\
= & \quad \{\![\text{ by annihilation, since } chain(a \lhd next) \subseteq next \text{ by Lemma 17(b) }]\!\} \\
 & chain(c \bullet (b \lhd next)) \mid next \\
= & \quad \{\![\text{ by (32) }]\!\} \\
 & (c \bullet first(b \lhd next) \ \cup \ chain(b \lhd next)) \mid next \\
= & \quad \{\![\text{ by Lemma 17(a) }]\!\} \\
 & (c \bullet b \ \cup \ chain(b \lhd next)) \mid next \\
= & \quad \{\![\text{ by annihilation, since } chain(b \lhd next) \subseteq next \ ]\!\} \\
 & (c \bullet b) \mid next \ .
\end{array}$$

Since $c = \mathsf{lst}\,(a \lhd next)$, we specify an auxiliary function *neconc* by

$$neconc(a, b, next) \ = \ (disjoint(a, b, next) \ \wedge \ a \in \mathsf{fst}\ next) \bullet (\mathsf{lst}\,(a \lhd next) \bullet b) \mid next \ .$$

Now one can use the standard recursion for lst and the definition of $\lhd$ to show, for $a \in \mathsf{fst}\, next$,

$$\mathsf{lst}\,(a \lhd next) \;=\; \mathsf{if}\;\; a\,;\, next \in \mathsf{fst}\, next\, \mathsf{then}\, \mathsf{lst}\,((a\,;\, next) \lhd next)\, \mathsf{else}\, a\;\; \mathsf{fi}\;\; .$$

A straightforward unfold/fold transformation for *neconc* then yields

$$neconc(a, b, next) \;=\; \mathsf{if}\;\; a\,;\, next \in \mathsf{fst}\, next\, \mathsf{then}\, neconc(a\,;\, next, b, next)\, \mathsf{else}\, (a \bullet b) \mid next\;\; \mathsf{fi}\;\; .$$

The transformation to the imperative level now gives the same result as before. This second derivation invests more lemmata, whereas the first one is more direct.

5.7 Chain Reversal

Specification and First Explicit Solution. Next we want to derive a procedure for reversing a non-empty chain "in situ". The problem is as follows:

Given a head a in a state $next$, we want to form a new state which contains the proper records of the chain of a in reverse order of traversal.

We can express this formally as follows:

$$reverse(a, next) \stackrel{\text{def}}{=} chain((a \lhd next)^{-1}) \mid next\;\; .$$

Let us now derive a recursion for *reverse*. The basic idea for the development is to adapt the standard technique for deriving a tail recursion for the reversal operation. There one defines a generalised function *rev* with an additional parameter that accumulates the intermediate results:

$$rev(s, t) \stackrel{\text{def}}{=} s^{-1} \bullet t\;\; .$$

Reversal is embedded into *rev* by $s^{-1} = rev(s, \varepsilon)$. A straightforward unfold/fold derivation using associativity of $\bullet$ leads to the tail-recursion

$$\begin{aligned} rev(\varepsilon, t) &= t\;\; , \\ rev(a \bullet s, t) &= rev(s, a \bullet t)\;\; . \end{aligned}$$

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

$$reve(a, b, next) \stackrel{\text{def}}{=} disjoint(a, b, next) \bullet chain((a \lhd next)^{-1} \bullet (b \lhd next)) \mid next\;\; .$$

An appropriate embedding is $reverse(a, next) = reve(a, \square, next)$, since $\square \lhd next = \varepsilon$. As before, we now perform a case analysis.

Case 1: $a = \square$. Then $a \lhd next = \varepsilon$, and hence $(a \lhd next)^{-1} = \varepsilon$. Thus

$$\begin{aligned} & chain((a \lhd next)^{-1} \bullet (b \lhd next)) \mid next \\ = \;\; & \{\!\![\ \text{neutrality}\ ]\!\!\} \\ & chain(b \lhd next) \mid next \\ = \;\; & \{\!\![\ \text{since } chain(b \lhd next) \subseteq next \text{ by Lemma 17(b)}\ ]\!\!\} \\ & next\;\; . \end{aligned}$$

Case 2: $a \neq \square$. Then

$$
\begin{aligned}
& \mathit{chain}((a \lhd \mathit{next})^{-1} \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= \quad & \{\![\text{ by (30) }]\!\} \\
& \mathit{chain}((a \bullet (a \,;\, \mathit{next} \lhd \mathit{next}))^{-1} \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= \quad & \{\![\text{ reversal }]\!\} \\
& \mathit{chain}((a \,;\, \mathit{next} \lhd \mathit{next})^{-1} \bullet a \bullet (b \lhd \mathit{next})) \mid \mathit{next} \\
= \quad & \{\![\text{ by Corollary 16, Lemma 17(a) and idempotence (28) }]\!\} \\
& \mathit{chain}((a \,;\, \mathit{next} \lhd \mathit{next})^{-1} \bullet a \bullet (b \lhd \mathit{next})) \mid (a \bullet b) \mid \mathit{next} \\
= \quad & \{\![\text{ by Corollary 12, since } \mathit{disjoint}(a, b, \mathit{next}) \text{ implies} \\
& \quad a \notin \mathsf{set}\,(b \lhd \mathit{next}) \cup \mathsf{set}\,((a \,;\, \mathit{next}) \lhd \mathit{next}) \;]\!\} \\
& \mathit{chain}(((a \,;\, \mathit{next}) \lhd ((a \bullet b) \mid \mathit{next}))^{-1} \bullet a \bullet (b \lhd (a \bullet b) \mid \mathit{next})) \mid (a \bullet b) \mid \mathit{next} \\
= \quad & \{\![\text{ definition of } \lhd \;]\!\} \\
& \mathit{chain}(((a \,;\, \mathit{next}) \lhd ((a \bullet b) \mid \mathit{next}))^{-1} \bullet (a \lhd ((a \bullet b) \mid \mathit{next}))) \mid (a \bullet b) \mid \mathit{next} \\
= \quad & \{\![\text{ fold } \mathit{reve} \;]\!\} \\
& \mathit{reve}(a \,;\, \mathit{next},\ a,\ (a \bullet b) \mid \mathit{next}) \ .
\end{aligned}
$$

Again, we have to check the validity of the assertion for the recursive call. Assuming that $a \neq \square \wedge \mathit{disjoint}(a, b, \mathit{next})$ holds, we calculate:

$$
\begin{aligned}
& \mathit{disjoint}(a \,;\, \mathit{next},\ a,\ (a \bullet b) | \mathit{next}) \\
= \quad & \{\![\text{ definition of } \mathit{disjoint} \;]\!\} \\
& (a \,;\, \mathit{next} \,;\, ((a \bullet b) | \mathit{next})^*) \cap (a \,;\, ((a \bullet b) | \mathit{next})^*) = \square \\
= \quad & \{\![\text{ by Corollary 12 }]\!\} \\
& (a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap (a \,;\, ((a \bullet b) | \mathit{next})^*) = \square \\
= \quad & \{\![\text{ definition of } ((a \bullet b) | \mathit{next})^* \text{ and } |, \text{ distributivity }]\!\} \\
& (a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap (a \ \cup \ b \,;\, ((a \bullet b) | \mathit{next})^*) = \square \\
= \quad & \{\![\text{ by Corollary 12 and } \mathit{disjoint}(a, b, \mathit{next}) \;]\!\} \\
& (a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap (a \ \cup \ b \,;\, \mathit{next}^*) = \square \\
= \quad & \{\![\text{ distributivity }]\!\} \\
& ((a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap a) \ \cup \ ((a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap (b \,;\, \mathit{next}^*)) = \square \\
= \quad & \{\![\text{ since } (a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap a \subseteq a \neq \square \;]\!\} \\
& (a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap a = \emptyset \wedge (a \,;\, \mathit{next} \,;\, \mathit{next}^*) \cap (b \,;\, \mathit{next}^*) = \square \ .
\end{aligned}
$$

Both conjuncts are implied by $\mathit{disjoint}(a, b, \mathit{next})$, since this also implies $\square \in a \,;\, \mathit{next}^*$.

Altogether, we have

$$
\mathit{reve}(a, b, \mathit{next}) = \mathsf{if}\ a = \square\ \mathsf{then}\ \mathit{next}\ \mathsf{else}\ \mathit{reve}(a \,;\, \mathit{next},\ a,\ (a \bullet b) \mid \mathit{next})\ \mathsf{fi}\ .
$$

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

A Version With Selective Updating. Continuing to the imperative level, again by standard transformation techniqes one sees that the assignment $next := reverse(a, next)$ is equivalent to

$$(va, vb) := (a, \Box) \ ;$$
$$\mathsf{while}\ va \neq \Box\ \mathsf{do}\ (va, vb, next) := ((va\,;\, next),\ va,\ (va \bullet vb) \mid next)\ \mathsf{od}\ .$$

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. [8, 24]), 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.

5.8 Copying Pointer Structures

We now return to the case of general pointer structures. A frequent task is that of copying a structure to some other part of the memory. We want to formalise that notion and develop one particular copying algorithm.

Copies. Let R, S be states. A **relocation** from R to S is a total bijective map

$$K : \mathsf{set}\, R \ \cup\ \Box\ \to\ \mathsf{set}\, S\ \cup\ \Box$$

such that $K(\Box) = \Box$ and the following diagram commutes:

$$\begin{array}{ccc} \mathsf{fst}\, R & \overset{R}{\leftrightarrow} & \mathsf{lst}\, R \\ \Big\downarrow K & & \Big\downarrow K \\ \mathsf{fst}\, S & \overset{S}{\leftrightarrow} & \mathsf{lst}\, S \end{array}$$

We call S the **copy** of R by K. The problem now is:

Given an access relation R and a relocation K, construct the copy of R by K.

Commutation of the above diagram means that $R;K = K;S$ and, since K is bijective, that $S = K^{-1}\,;R\,;K$. Hence, given K, we can compute S from R in two passes: first we form $R\,;K$ which means that the links as given by R are redirected to the corresponding records of the copy ("pointer relocation pass"); then we compose with K^{-1} which means the actual transfer of the new links to the new records ("copying pass").

Copying Pass. Given $P = R;K$, the copying pass is easily performed. First, by totality of K, we have $\mathsf{fst}\, P = \mathsf{fst}\, R\ (\subseteq \mathsf{fst}\, K)$. Moreover, (10) and the defining laws for $.^{-1}$ imply that

$$L^{-1} = \bigcup_{a \in \mathsf{fst}\, L} (a\,;L) \bullet a \ . \tag{36}$$

Now

$$\begin{aligned}
& K^{-1}\,;P \\
= \quad & \{\![\text{ by (36) }]\!\} \\
& (\bigcup_{a \in \mathsf{fst}\, K} (a\,;K) \bullet a)\,;P \\
= \quad & \{\![\text{ distributivity }]\!\} \\
& \bigcup_{a \in \mathsf{fst}\, K} ((a\,;K) \bullet a)\,;P \\
= \quad & \{\![\text{ associativity }]\!\} \\
& \bigcup_{a \in \mathsf{fst}\, K} (a\,;K) \bullet (a\,;P) \\
= \quad & \bigcup_{a \in \mathsf{fst}\, P} (a\,;K) \bullet (a\,;P) \ ,
\end{aligned}$$

since $(a\,;K) \bullet (a\,;P) = \emptyset$ for $a \in \mathsf{fst}\, K \backslash \mathsf{fst}\, P = \mathsf{fst}\, K \backslash \mathsf{fst}\, R$. We set

$$\mathit{copass}(P,K) \stackrel{\text{def}}{=} (\mathsf{fst}\, P \subseteq \mathsf{fst}\, K) \bullet \bigcup_{a \in \mathsf{fst}\, P} (a\,;K) \bullet (a\,;P) \ .$$

Considering union as a loop leads to a straigtforward iterative algorithm.

Pointer Relocation. The more difficult subtask consists in computing the composition $R;K$ efficiently. For this we recall the domain and range oriented representations (10,11) of relations. According to these properties (and using distributivity) there are essentially two ways of forming $R\,;K$:

1. domain-oriented:
$$R\,;K = \bigcup_{a \in \mathsf{fst}\, R} a \bullet ((a\,;R)\,;K) \ .$$
If we view the union again as a loop, this way of forming $R\,;K$ needs an explicit representation of K, since the same value of K may be needed repeatedly at irregular intervals.
2. range-oriented:
$$R\,;K = \bigcup_{c \in \mathsf{lst}\, R} (R\,;c) \bullet (c\,;K) \ .$$
For evaluating this by a loop we only need one K-value, viz. $c\,;K$, at a time to process the whole subset $R\,;c$ of $\mathsf{fst}\, R$. Hence we can avoid explicit representation of the complete K. Moreover, the repeated lookups are avoided and thus also time-efficiency is improved. Of course, this latter aspect is interesting only if R is highly non-injective so that the inverse images $R\,;c$ are large.

We follow now the range-oriented variant. We need a way of representing the component maps $(R\,;c)\bullet b$ with $b \stackrel{\text{def}}{=} z\,;K$ suitably. For this we use an idea that is presented e.g. in [12]: all elements of $R;c$ are chained into a linked list; then $(R;c)\bullet b$ can be formed following the chain as $\bigcup_{a\in R;c} a\bullet b$, again viewing union as a loop.

Chained Representation of States. We say that a pair (a,R) **represents** a set S of proper records if $(a\,;R^*)\backslash\Box = S$. Let now R be a state. From the range-oriented decomposition $R = \bigcup_{c\in\mathsf{lst}\,R} R\,;c\bullet c$ we obtain the partition $\mathsf{fst}\,R = \bigcup_{c\in\mathsf{lst}\,R} R\,;c$. We represent R by a union of chains each of which represents one of the sets $R\,;c$ and prefix c as a header record to the respective chain. To avoid confusion between these chains we require that

$$\mathit{ischainable}(R) \stackrel{\text{def}}{\Leftrightarrow} \mathsf{fst}\,R \cap \mathsf{lst}\,R = \emptyset$$

holds; otherwise there would be a link from one chain to another and the partition would be lost. In concrete realisations of pointer structures this requirement can be met by considering the actual allocation of records in cells of the von Neumann store. Each record can be viewed as a micro-structure in which the cells are the nodes linked by the successor map *next* (on addresses) which is considered partial on the terminal cell of each record. From these cells emanate the links proper to the starting cells of other records; this is described by a map *link*. If we now assume that each record has a starting cell c with the pseudo-link $c\bullet\Box\subseteq \mathit{link}$ and define

$$\begin{aligned}\mathit{heads} &\stackrel{\text{def}}{=} \mathit{link}\,;\Box\ ,\\ \mathit{arcs} &\stackrel{\text{def}}{=} \overline{\mathit{heads}}\bowtie \mathit{link}\ ,\end{aligned}$$

we have $\mathit{ischainable}(\mathit{arcs})$. We set $R \stackrel{\text{def}}{=} \mathit{link}\cup\mathit{next}$; the access relation between starting cells of records then is $\mathit{next}^+\,;\mathit{link}$.

Suppose now that R is a state satisfying $\mathit{ischainable}(R)$. Since the elements of $R;c$ 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 binary relation $P\subseteq A\bullet A$,

$$\mathit{src}(P) \stackrel{\text{def}}{=} \mathsf{fst}\,P\backslash\mathsf{lst}\,P\ ;$$

for a pointer structure P this is the set of records to which no pointer exists in P. With the help of this notion we can characterise chainings of maps by the following predicate *ischaining*:

$$\mathit{ischaining}(P,R) \stackrel{\text{def}}{\Leftrightarrow} \mathit{src}(P) = \mathsf{lst}\,R \wedge \forall\, c\in\mathit{src}(P) : R\,;c = \mathsf{set}\,((z\,;P)\triangleleft P)\ .$$

In this case

$$R = \mathit{unchain}(P) \stackrel{\text{def}}{=} \bigcup_{c\in\mathit{src}(P)} \mathsf{set}\,((z\,;P)\triangleleft P)\bullet c\ .$$

To compute $R\,;K$ one now first constructs a chained representation P of R and then forms

$$\mathit{relocate}(P,K) \stackrel{\text{def}}{=} \mathit{unchain}(P)\,;K\ ,$$

which, by distributivity, leads to the loop

$$relocate(P, K) \stackrel{\text{def}}{=} \bigcup_{c \in src(P)} \mathsf{set}\,((z\,;P) \triangleleft P)\ \bullet\ c\,;K\ .$$

In [3] we develop a suitable incrementation function for computing a chaining of $R \cup (a \bullet b)$ from a chaining of R, which allows the stepwise construction of chainings.

5.9 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 [3]. For earlier attempts at similar problems cf. [7, 13, 14]. Differing from all of these, [3] 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.

Formal Problem Specification. Garbage collection becomes necessary when the store is exhausted, i.e., when 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 store which is given by the values of the currently active variables of the program that operates on the store. Only the substate consisting of the records reachable through chains of references from the entry pointers actually needs to be saved; all other records are inaccessible and thus the corresponding storage can be reclaimed.

However, for a state the restriction to a substate usually leads to gaps in the storage. 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.

To be able to talk about compactification we now assume that the set A of records is denumerable and linearly ordered by some ordering $\leq \subseteq A \bullet A$ in which the anchor $\Box$ is the least element. Without loss of generality we assume A to be the set $\mathbb{N}$ of natural numbers under the usual ordering $\leq$; then $\Box = 0$.

Now a state R is called **compressed** if $\mathsf{fst}\,R = (\leq\,;\mathsf{fst}\,R)\backslash\Box$, i.e., if its domain is an initial interval of $A\backslash\Box$, which implies that there are no gaps in $\mathsf{fst}\,R$. Now the garbage collection problem is as follows:

Given a state R and a set $S \subseteq \mathsf{fst}\,R$, compute the reachable substate $R_S \stackrel{\text{def}}{=} from(S, R)$ (see Section 5.2) together with a relocation K of R_S such that the corresponding copy $R_c \stackrel{\text{def}}{=} K^{-1}\,;R\,;K$ is compressed.

In fact, ultimately we are interested in an algorithm that computes R_c directly from R. In the following sections we present some essential parts of the development of such an algorithm.

Compressing Chained Graph States. To actually form a compressed copy of the reachable substate we can use the copying algorithm from the previous section.

Since there is an ordering on the records, one possible decision is to use an **order-preserving** relocation, i.e., a relocation K such that $\leq \ \subseteq \ K\,;\leq\,;K^{-1}$. It can be shown [3] that for a state R there is exactly one order-preserving relocation K of R such that the copy of R by K is compressed; K is given by $K(\Box) = \Box$ and

$$a\,;K \stackrel{\text{def}}{=} |(\leq\,;a) \cap \mathsf{set}\,R|$$

for $a \in \mathsf{set}\,R\backslash\Box$.

We now assume a partition

$$R \;=\; \mathit{link} \;\cup\; \mathit{next}$$

as in the previous section. Then we have, by the results of Section 5.8, that

$$\begin{aligned} \mathit{arcs}_c \;&=\; \mathit{copass}(\mathit{relocate}(\mathit{arcs},K),K) \;\;, \\ \mathit{link}_c \;&=\; \mathit{heads}\,;K \bullet \Box \;\cup\; \mathit{arcs}_c \;\;. \end{aligned}$$

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

The relocation pass thus returns a union LP of the state $P \stackrel{\text{def}}{=} \mathit{relocate}(\mathit{arcs},K)$ and a chain L with $\mathsf{fst}\,L = \mathit{heads}$. Now we can construct link_c from this union based on the values $\mathit{size}(c)$ for $c \in \mathit{heads}$, where

$$\mathit{size}(c) \stackrel{\text{def}}{=} |c\,;\mathit{next}^+|$$

for $c \in \mathit{heads}$ gives the number of successor cells of c in the same record. It is easily calculated that

$$\mathit{link}_c \;=\; \bigcup_{c\in\mathit{heads}} (c\,;K \bullet \Box \;\cup\; \bigcup_{i=1}^{\mathit{size}(c)} ((c\,;K)+i) \bullet (c\,;\mathit{next}^i\,;P)) \;\;,$$

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

$$b\,;\mathit{succ}\,;K \;=\; (b\,;K)+\mathit{size}(b)$$

for $b \in \mathit{heads}\backslash\mathit{max}(\mathit{heads})$ and

$$\mathit{succ}(b) \stackrel{\text{def}}{=} \mathit{min}(\mathit{heads}\backslash(\leq\,;b)) \;\;.$$

Hence we can eliminate K and use its values on the single records instead. So no extra space for K is necessary.

Merging Reachability and Chaining. Above 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 from Section 5.2 so that it does not use a separate parameter for the set T of cells already visited, since this would occupy a lot of space. Everything has to be done using just a few auxiliary records. This is achieved by a specialisation of the reachability algorithm above: the set S is represented by a sequence w, and the choice of an arbitrary element of S is implemented by $\mathit{first}(w)$. In this way we obtain a depth-first traversal of the reachable substructure with w as the stack. This stack is then compressed considerably by representing the set of cells belonging to a record by its leading cell only.

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 [3] is linear in the size of the store in which garbage collection takes place.

6 Conclusion

We have shown with several examples how to derive graph and pointer algorithms from formal specifications using standard transformation techniques in connection with a powerful algebra for that particular problem domain.

In the case of graph algorithms the concept of Kleene algebras and their induction principles turned out to be most fruitful. Also, the different degrees of abstraction provided by the path closure and the conventional reflexive transitive closure as well as their homomorphic connection have proved to be flexible and economic in that results can easily be transferred between these levels.

In the case of pointer algorithms, 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. [26, 20]) 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 [5]. Moreover, the operations of that algebra are expressive enough that we did not need to explain anything with the help of pictures. Even when developing the intricate garbage collection algorithm described above we quite soon stopped drawing pictures 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 [37].

We are convinced that a similar approach can be taken for many other problem domains, the main task being to find the appropriate algebraic structures for these. A long term goal is the construction of a database of the corresponding operators, enhanced by indexing and external representation using informal language referring to the intuitive meaning of the operators. Such a component would serve as a "specifier's workbench" as a front end to a formal development tool.

Acknowledgements

Many individuals have helped me with their advice. I gratefully acknowledge helpful remarks from F.L. Bauer, R. Berghammer, R. Bird, J. Desharnais, W. Dosch, H. Ehler, M. Lichtmannegger, O. de Moor, H. Partsch, P. Pepper, M. Russling, G. Schmidt and M. Sintzoff.

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 (Hrsg.): Methodik des Programmierens. Fakultät für Mathematik und Informatik, Universität Passau, MIP-8915, 1989, 1–52. Also in: M. Broy, M. Wirsing (eds.): Methods of programming. Lecture Notes in Computer Science **544**. Berlin: Springer 1991, 137–192
4. R. Berghammer: A transformational development of several algorithms for testing the existence of cycles in a directed graph. Institut für Informatik der TU München, TUM-I8615, 1986
5. A. Bijlsma: Calculating with pointers. Science of Computer Programming **12**, 191–205 (1988)
6. R. Bird: Lectures on constructive functional programming. In: M. Broy (ed.): Constructive methods in computing science. NATO ASI Series. Series F: Computer and systems sciences **55**. Berlin: Springer 1989, 151–216
7. M. Broy, P. Pepper: Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite-Algorithm. ACM TOPLAS **4**, 362–381 (1982)
8. 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

9. R.M. Burstall, J. Darlington: A transformation system for developing recursive programs. J. ACM **24**, 44–67 (1977)
10. J. Cai, R. Paige: Program derivation by fixed point computation. Science of Computer Programming **11**, 197–261 (1989)
11. J.H. Conway: Regular algebra and finite machines. London: Chapman and Hall 1971
12. R. Dewar, A. McCann: MACRO SPITBOL — a SNOBOL4 compiler. Software — Practice and Experience **7**, 95–113 (1977)
13. R. Dewar, M. Sharir, E. Weixelbaum: Transformational derivation of a garbage collection algorithm. ACM TOPLAS **4**, 650–667 (1982)
14. N. van Diepen, W. de Roever: Program derivation through transformations: The evolution of list-copying algorithms. Science of Computer Programming **6**, 213–272 (1986)
15. S. Ginsburg: The mathematical theory of context-free languages. New York: McGraw-Hill 1966
16. R. C. Hehner: A practical theory of programming. Berlin: Springer 1993
17. A. Horsch: Functional programming with partially applicable operators. Fakultät für Mathematik und Informatik der TU München, Dissertation, 1989
18. C.B. Jones: Software development: A rigorous approach. Englewood Cliffs: Prentice-Hall 1980
19. P. Kanellakis: Elements of relational database theory. In: J. van Leeuven (ed.): Handbook of Theoretical Computer Science. Vol. B: Formal models and semantics. Amsterdam: North Holland 1990, 576–583
20. A. Kausche: Modale Logiken von geflechtartigen Datenstrukturen und ihre Kombination mit temporaler Programmlogik. Fakultät für Mathematik und Informatik der TU München, Dissertation, 1989
21. S.C. Kleene: Introduction to metamathematics. New York: van Nostrand 1952
22. B. Knaster: Un théorème sur les fonctions d'ensembles. Ann. Soc. Pol. Math. **6**, 133–134 (1928)
23. P. Lescanne: Modèles non déterministes de types abstraits. R.A.I.R.O. Informatique théorique **16**, 225–244 (1982)
24. M. Levy: Verification of programs with data referencing. Proc. 3me Colloque sur la Programmation 1978, 413–426
25. Z. Manna: Mathematical theory of computation. New York: McGraw-Hill 1974
26. I. Mason: Verification of programs that destructively manipulate data. Science of Computer Programming **10**, 177–210 (1988)
27. L.G.L.T. Meertens: Algorithmics — Towards programming as a mathematical activity. In: J. W. de Bakker et al. (eds.): Proc. CWI Symposium on Mathematics and Computer Science. CWI Monographs Vol **1**. Amsterdam: North-Holland 1986, 289–334
28. B. Möller: Applicative assertions. In: J.L.A. van de Snepscheut (ed.): Mathematics of Program Construction. Lecture Notes in Computer Science **375**. Berlin: Springer 1989, 348–362
29. 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
30. B. Möller: Towards pointer algebra. Institut für Mathematik der Universität Augsburg, Report No. 279, 1993. Also to appear in Science of Computer Programming
31. B. Möller: Algebraic calculation of graph and sorting algorithms. Invited Paper at the International Conference on Formal methods in Programming and their Applications, Nowosibirsk, 28.6.–3.7. 1993. Lecture Notes in Computer Science. Berlin: Springer 1993 (to appear)

32. B. Möller, M. Russling: Shorter paths to graph algorithms. In: R.S. Bird, C.C. Morgan, J.C.P. Woodcock (eds.): Mathematics of Program Construction. Lecture Notes in Computer Science **669**. Berlin: Springer 1993, 250–268. Extended version: Institut für Mathematik der Universität Augsburg, Report Nr. 272, 1992. Also to appear in Science of Computer Programming
33. H.A. Partsch: Specification and transformation of programs — A formal approach to software development. Berlin: Springer 1990
34. P. Pepper, B. Möller: Programming with (finite) mappings. In: M. Broy (ed.): Informatik und Mathematik. Berlin: Springer 1991, 381–405
35. M. Russling: Hamiltonian sorting. Institut für Mathematik der Universität Augsburg, Report Nr. 270, 1992
36. G. Schmidt, T. Ströhlein: Relations and graphs. Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Berlin: Springer 1993
37. N. Suzuki: Analysis of pointer rotation. Conf. Record 7th POPL, 1980, 1–11. Revised version: Commun. ACM **25**, 330–335 (1982)
38. A. Tarski: On the calculus of relations. J. Symbolic Logic **6**, 73–89 (1941)
39. A. Tarski: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**, 285–309 (1955)