Algebraic Calculation of Graph and Sorting Algorithms[*]

Bernhard Möller

Report Nr. 286 1993

[*]To appear in: V.K. Sabelfeld (ed.): Proc. International Conference on Formal Methods in Programming and Their Applications, Novosibirsk, Russia, June 29 - July 2, 1993. Lecture Notes in Computer Science. Berlin: Springer

Algebraic Calculation of Graph and Sorting 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 sorting 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 (see e.g. [3, 7, 1, 13]. There one starts from a (possibly non-executable) specification and transforms it into a (hopefully efficient) program using semantics-preserving rules. Many specifications and derivations, however, suffer from the use of lengthy, obscure and unstructured expressions involving formulas 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. [10, 2]) 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 sorting problems within a suitable algebra of formal languages [12]. This differs from the approach of [10, 2] 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.

2.1 Words, Languages and Relations

We denote by $A^{(*)}$ the set of all finite words over an alphabet A. A **(formal) language** V over A is a subset $V \subseteq A^{(*)}$. As is customary in formal language theory, to save parentheses a singleton language is identified with its only word and a word consisting just of one letter is identified with that letter. By ε we denote the empty word over A.

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

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

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 $\subseteq$:

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

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. [8]).

2.3 Concatenation and Auxiliaries

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

$$u \bullet (v \bullet w) = (u \bullet v) \bullet w \ ,$$
$$\varepsilon \bullet u \ = \ u \ = \ u \bullet \varepsilon \ .$$

Since associativity of concatenation and neutrality of ε are expressed by bilinear equational laws, they also hold for the pointwise extension of concatenation to languages:

$$U \bullet (V \bullet W) = (U \bullet V) \bullet W \ ,$$
$$\varepsilon \bullet U \ = \ U \ = \ U \bullet \varepsilon \ .$$

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

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

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 Shuffle

An operation on languages that is well-known from the trace theory of parallel processes is the **shuffle**, viz. the set of arbitrary interleavings of words from these languages. It is defined inductively ($a, b \in A, s, t \in A^{(*)}$):

$$\begin{aligned} \varepsilon \,|||\, t &\stackrel{\text{def}}{=} t \ , \\ s \,|||\, \varepsilon &\stackrel{\text{def}}{=} s \ , \\ (a \bullet s) \,|||\, (b \bullet t) &\stackrel{\text{def}}{=} a \bullet (s \,|||\, (b \bullet t)) \ \cup \ b \bullet ((a \bullet s) \,|||\, t) \ . \end{aligned}$$

As usual, the operation is extended pointwise to sets. It is commutative and associative and satisfies

$$\mathsf{set}\,(S \,|||\, T) \ = \ \mathsf{set}\,S \ \cup \ \mathsf{set}\,T \ . \tag{1}$$

2.5 Permutations

An important requirement for sorting is that the number of occurrences of each element in the word to be sorted must not be changed by the sorting process. An equivalent requirement is that the result word must be a permutation of the input word. To capture that, we give a definition of permutation in terms of our operators. For a word s we denote the set of its permutations by $\mathit{permw}(s)$. Formally,

$$\begin{aligned} \mathit{permw}(\varepsilon) &\overset{\text{def}}{=} \varepsilon \ , \\ \mathit{permw}(a) &\overset{\text{def}}{=} a \ , \\ \mathit{permw}(s \bullet t) &\overset{\text{def}}{=} \mathit{permw}(s) \,|||\, \mathit{permw}(t) \ . \end{aligned} \tag{2}$$

A useful property is

$$a \,|||\, \mathit{permw}(s) \ = \bigcup_{u \bullet v \in \mathit{permw}(s)} \mathit{permw}(u) \bullet a \bullet \mathit{permw}(v) \ . \tag{3}$$

We shall also need the set of all permutations of a finite set of letters. This is given by

$$\begin{aligned} \mathit{perms}(\emptyset) &\overset{\text{def}}{=} \varepsilon \ , \\ \mathit{perms}(a \cup T) &\overset{\text{def}}{=} a \,|||\, \mathit{perms}(T \backslash a) \ , \end{aligned} \tag{4}$$

for $a \in A$ and finite $T \subseteq A$. Note that in general $\mathit{permw}(s) \neq \mathit{perms}(\mathsf{set}\, s)$, since multiple occurrences of a letter are preserved by $\mathit{permw}(s)$ but not by $\mathit{perms}(\mathsf{set}\, s)$. Hence each element of $\mathit{perms}(T)$ is repetition-free.

2.6 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

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

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. On binary relations, composition coincides with usual relational composition (see e.g. [19]). 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 node without making that node 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 nodes.

The operations associate nicely with each other and with concatenation:

$$\left.\begin{array}{rll}
U \bullet (V \bullet W) = (U \bullet V) \bullet W & , & \\
U \bowtie (V \bowtie W) = (U \bowtie V) \bowtie W & , & \\
U\,;(V\,;W) = (U\,;V)\,;W & \Leftarrow V \cap A = \emptyset & , \\
U\,;(V \bowtie W) = (U\,;V) \bowtie W & \Leftarrow V \cap A = \emptyset & , \\
(U \bowtie V)\,;W = U \bowtie (V\,;W) & \Leftarrow V \cap A = \emptyset & , \\
U \bullet (V \bowtie W) = (U \bullet V) \bowtie W & \Leftarrow V \cap \varepsilon = \emptyset & , \\
U \bowtie (V \bullet W) = (U \bowtie V) \bullet W & \Leftarrow V \cap \varepsilon = \emptyset & , \\
U \bullet (V\,;W) = (U \bullet V)\,;W & \Leftarrow V \cap \varepsilon = \emptyset & , \\
U\,;(V \bullet W) = (U\,;V) \bullet W & \Leftarrow V \cap \varepsilon = \emptyset & .
\end{array}\right\} \qquad (5)$$

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

$$S \bowtie R \bowtie T = S \bullet T \cap R\ , \qquad S\,;R\,;T = \begin{cases} \varepsilon & \text{if } S \bullet T \cap R \neq \emptyset\ , \\ \emptyset & \text{otherwise}\ . \end{cases} \qquad (6)$$

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

$$R \bowtie S = R \cap S\ , \qquad R\,;S = \begin{cases} \varepsilon & \text{if } R \cap S \neq \emptyset\ , \\ \emptyset & \text{if } R \cap S = \emptyset\ . \end{cases} \qquad (7)$$

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

$$P \bowtie V = V = V \bowtie Q\ , \qquad I_P\,;V = V = V\,;I_Q\ . \qquad (8)$$

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

$$P \bowtie R = I_P \,;R \ , \qquad R \bowtie Q = R\,; I_Q \ , \tag{9}$$

$$P\,;(Q \bowtie R) = (P \bowtie Q)\,;R \ , \qquad (R \bowtie P)\,;Q = R\,;(P \bowtie Q) \ . \tag{10}$$

2.7 Assertions, Guards and Conditional

As we have seen in (6) and (7), 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

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

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

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

in [11] and can be used for propagating assertions through recursions.

Using assertions we can also define a guarded expression and a conditional by

$$\mathsf{if}\ B_1\,\mathsf{then}\,U_1 \ [] \ \cdots \ []\ B_n\,\mathsf{then}\,U_n\ \mathsf{fi} \ \overset{\text{def}}{=} \ \bigcup_{i=1}^{n} B_i \bullet U_i \ ,$$

$$\mathsf{if}\ B\,\mathsf{then}\,U\,\mathsf{else}\,V\ \mathsf{fi} \ \overset{\text{def}}{=} \ \mathsf{if}\ B\,\mathsf{then}\,U \ []\ \overline{B}\,\mathsf{then}\,V\ \mathsf{fi} \ ,$$

for assertions B, B_i and languages U, U_i, V. Although the conditional 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. Note that the guarded expression has angelic semantics: whenever one of the branches is $\neq \emptyset$, so is the whole expression.

3 Closure Operations

We now study in more detail iterated join and composition of a binary relation with itself, which were already seen to be important for path problems.

3.1 Closures

Consider a binary relation $R \subseteq A \bullet A$. We define the **(reflexive and transitive) closure** R^* and the **transitive closure** R^+ of R by

$$R^* \overset{\text{def}}{=} \bigcup_{i \in \mathbb{N}} R^i \ , \qquad R^+ \overset{\text{def}}{=} \bigcup_{i \in \mathbb{N}\setminus 0} R^i \ ,$$

where, as usual, $R^0 \overset{\text{def}}{=} I_A$ and $R^{i+1} \overset{\text{def}}{=} R\,;R^i$. It is well-known that R^* is the least fixpoint of the recursion equations

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

This is important since it allows proofs about R^* using fixpoint induction (see e.g. [9]).

Let G be the directed graph associated with R. We have

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

For $S \subseteq A$, the set $S\,;R^*$ gives all nodes in A reachable from nodes in S via paths in G, whereas $R^*;S$ gives all nodes in A from which some node in S can be reached.

Analogously to R^* we introduce the **path closure** $R^{\bowtie}$ and the **proper path closure** $R^{\Rightarrow}$ of R by

$$R^{\bowtie} \overset{\text{def}}{=} \bigcup_{i \in \mathbb{N}} {}^{i}R \ , \qquad R^{\Rightarrow} \overset{\text{def}}{=} \bigcup_{i \in \mathbb{N}\setminus 0} {}^{i}R \ ,$$

where ${}^{0}R \overset{\text{def}}{=} A$ and ${}^{i+1}R \overset{\text{def}}{=} R \bowtie {}^{i}R$. $R^{\bowtie}$ is the least fixpoint of the recursion equations

$$R^{\bowtie} = A \cup R \bowtie R^{\bowtie} = A \cup R^{\bowtie} \bowtie R \ .$$

It consists of all finite paths in G including the trivial ones with just one node, whereas $R^{\Rightarrow}$ consists of paths with at least one edge. Hence

$$a \bowtie R^{\bowtie} \bowtie b$$

is the language of all paths between a and b in G.

A uniform treatment of these closure operations within the framework of Kleene algebras (see [5]) can be found in [15].

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 $\leq$.

To close this section we present a property that allows localising graph traversals in computing the nodes reachable from some set $T \subseteq A$. Suppose $R \subseteq A \bullet A$. By the fixpoint property of R^* and distributivity we have, for $R \subseteq A \bullet A$,

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

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

Lemma 1. *Assume $R \subseteq A \bullet A$ and $T \subseteq A$. Then, with $\overline{T} \overset{\text{def}}{=} A \backslash T$,*

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

The proof uses (a variant of) fixpoint induction and can be found in [15].

3.2 Applications to Sorting

We have already briefly mentioned the relation between the path closure and ordered sequences. However, the path closure consists of nonempty words only, whereas sorting algorithms usually also work for the empty word. Therefore we define, for binary relation $\prec \subseteq A \bullet A$, the **improper path closure** $\prec^{\rightsquigarrow}$ of $\prec$ by

$$\prec^{\rightsquigarrow} \overset{\text{def}}{=} \varepsilon \cup \prec^{\bowtie} \ .$$

In the sequel s and t range over $A^{(*)}$ and a over A. Moreover, we abbreviate the assertion $S \bullet T \subseteq \prec$ for $S, T \subseteq A$ by $S \prec T$.

Intersection with the improper path closure distributes through join:

$$(s \bowtie t) \cap \prec^{\rightsquigarrow} = (s \cap \prec^{\rightsquigarrow}) \bowtie (t \cap \prec^{\rightsquigarrow}) \ .$$

This implies

$$s \bullet a \bullet t \cap \prec^{\rightsquigarrow} = (s \bullet a \cap \prec^{\rightsquigarrow}) \bowtie (a \bullet t \cap \prec^{\rightsquigarrow}) \ . \tag{11}$$

Moreover, we have the following important property:

$$(s \bullet t) \cap \prec^{\rightsquigarrow} = (s \cap \prec^{\rightsquigarrow}) \bullet (\mathsf{lst}\, s \prec \mathsf{fst}\, t) \bullet (t \cap \prec^{\rightsquigarrow}) \ . \tag{12}$$

In particular (using also that assertions commute with all languages w.r.t. $\bullet$),

$$(a \bullet t) \cap \prec^{\rightsquigarrow} = (a \prec \mathsf{fst}\, t) \bullet a \bullet (t \cap \prec^{\rightsquigarrow}) \ .$$

From this we obtain

Corollary 2. *If $\prec$ is transitive then*

$$(a \bullet t) \cap \prec^{\rightsquigarrow} = (a \prec \mathsf{set}\, t) \bullet a \bullet (t \cap \prec^{\rightsquigarrow}) \ .$$

Moreover, if for all $t \in T \subseteq A$ we have $\mathsf{set}\, t = U$ then

$$(a \bullet T) \cap \prec^{\rightsquigarrow} = (a \prec U) \bullet a \bullet (T \cap \prec^{\rightsquigarrow}) \ .$$

The additional condition in the extension to languages is necessary, since the property on words is not bilinear.

For transitive relations we have a weak distributivity property w.r.t. shuffle, more precisely:

Lemma 3 (Merging Lemma). *$\prec$ is transitive iff*

$$\forall\, S \,.\, \forall\, T \,.\, (S \,|||\, T) \cap \prec^{\rightsquigarrow} = ((S \cap \prec^{\rightsquigarrow}) \,|||\, (T \cap \prec^{\rightsquigarrow})) \cap \prec^{\rightsquigarrow} \ .$$

Proof. ($\Leftarrow$) Assume $a \prec b$ and $b \prec c$. Then

$$
\begin{array}{ll}
 & \emptyset \\
\neq & \{\![\ \text{by } a \bullet b \bullet c \in \prec^{\leadsto}\ ]\!\} \\
 & (b \bullet a \bullet c \ \cup\ a \bullet b \bullet c \ \cup\ a \bullet c \bullet b) \cap \prec^{\leadsto} \\
= & \{\![\ \text{definition}\ ]\!\} \\
 & ((a \bullet c) \,|||\, b) \cap \prec^{\leadsto} \\
= & \{\![\ \text{assumption}\ ]\!\} \\
 & (((a \bullet c) \cap \prec^{\leadsto}) \,|||\, (b \cap \prec^{\leadsto})) \cap \prec^{\leadsto} \\
= & \{\![\ \text{definition of } \prec^{\leadsto}\ ]\!\} \\
 & (((a \bullet c) \cap \prec) \,|||\, b) \cap \prec^{\leadsto} \quad .
\end{array}
$$

Now strictness shows $(a \bullet c) \cap \prec \neq \emptyset$, i.e. $a \prec c$.

($\Rightarrow$) We show the property for single words; by bilinearity it propagates to languages. The property is immediate if $s \in \prec^{\leadsto}$ and $t \in \prec^{\leadsto}$. So assume now $s \notin \prec^{\leadsto}$. By strictness the right hand side reduces to $\emptyset$. Moreover, by (12) there are u, a, b, w such that $s = u \bullet a \bullet b \bullet v$ and $a \not\prec b$. Consider now some $w \in s \,|||\, t$. It has the form $w = p \bullet a \bullet q \bullet b \bullet r$ for certain $p, q, r \in A^{(*)}$, since shuffling preserves the relative order of the elements of each argument word. Now the assumption $w \in \prec^{\leadsto}$ by transitivity implies $a \prec b$, a contradiction. $\square$

The formula in the above lemma states that a word is ordered iff all its scattered subwords are. This property will be crucial for the derivation of sorting algorithms.

4 Graph Algorithms

We now want to use our framework to derive three simple graph algorithms, viz. a reachability algorithm, cycle detection and topological sorting. As further applications, [16] calculates an algorithm computing the length of a shortest connecting path between two graph nodes, whereas [18] deals with an algorithm for finding Hamiltonian cycles and relates it to the selection sort algorithm.

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 nodes, and a subset $S \subseteq A$, compute the set of nodes reachable by paths starting in S.

Hence we define

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

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 Lemma 1:

$$S \,;\, R^* = S \ \cup\ S \,;\, R \,;\, (\overline{S} \bowtie R)^* \quad ,$$

where $\overline{S} \overset{\text{def}}{=} A\backslash S$. However, since on the right hand side we have $(\overline{S} \bowtie R)^*$ 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. [17]): we introduce a second parameter T for the set that restricts R by defining

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

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

$$\begin{array}{cl}
 & re(S,T) \\
= & \{\!\![\ \text{definition of } re\ ]\!\!\} \\
 & S\,;(\overline{T} \bowtie R)^* \\
= & \{\!\![\ \text{by Lemma 1}\ ]\!\!\} \\
 & S\ \cup\ S\,;(\overline{T} \bowtie R)\,;(\overline{S} \bowtie \overline{T} \bowtie R)^* \\
= & \{\!\![\ \text{by (10)}\ ]\!\!\} \\
 & S\ \cup\ (S \bowtie \overline{T})\,;R\,;(\overline{S} \bowtie \overline{T} \bowtie R)^* \\
= & \{\!\![\ \text{by (7) and Boolean algebra}\ ]\!\!\} \\
 & S\ \cup\ (S\backslash T)\,;R\,;(\overline{S \cup T} \bowtie R)^* \\
= & \{\!\![\ \text{definition of } re\ ]\!\!\} \\
 & S\ \cup\ re((S\backslash T)\,;R,\ S \cup T)\ .
\end{array}$$

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 (7) the test $S = \emptyset$ can be expressed by the assertion $S\,;S$. We see that T keeps track of the nodes "already visited", while S is the set of nodes 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. [17]) 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 nodes 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 $R^{\Rightarrow}$. The set of all proper paths that begin and end in the same node is

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

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.

Lemma 4. *The following statements are equivalent:*
(a) $cyc(R) \neq \emptyset$.
(b) $R^{+} \cap I_A \neq \emptyset$.
(c) $R^{|A|} \neq \emptyset$.
(d) $R^{|A|} ; A \neq \emptyset$.
(e) $A ; R^{|A|} \neq \emptyset$.

For the proof see [15]. Among these equivalent formulations, (d) and (e) seem computationally most promising, since they deal with unary relations which in general are much smaller objects than binary ones. We choose (e) as our starting point and specify our problem as

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

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

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

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 5. *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 [16]. A similar theorem has been stated in [4].

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

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

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 6. $A_{|A|} = A_\infty$.

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 $\mathit{inf}(A) \neq \emptyset$. For our special case the recursion for *inf* reads (omitting the trivial part $S \subseteq A$)

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

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

$$\mathit{src}(S,R) \overset{\text{def}}{=} S\backslash(S\,;R)\ .$$

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

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

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

This is an improvement in that $\mathit{src}(S,R)$ usually will be small compared with S. Moreover, the computation of $\mathit{src}(S,R)$ can be facilitated by a suitable representation of R. Plugging *inf* into our original problem of cycle recognition we obtain

$$\begin{aligned}\mathit{hascycle} &= \mathit{hcy}(A)\ ,\\ \mathit{hcy}(S) &= (A_\infty \subseteq S \wedge S\,;R \subseteq S) \bullet\\ &\qquad \mathsf{if}\ \mathit{src}(S,R) = \emptyset\ \mathsf{then}\ S \neq \emptyset\ \mathsf{else}\ \mathit{hcy}(S\backslash \mathit{src}(S,R))\ \mathsf{fi}\ ,\end{aligned} \tag{13}$$

which is one of the classical algorithms and works by successive removal of sources. Note that Lemma 4(d) 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 [16] a more efficient one, in which the source sets are computed from an incrementally adjusted vector of in-degrees of the graph nodes. The transition from the tail-recursive functional versions to imperative ones with loops and variables which administer the data structures in place is standard transformational knowledge [17].

4.3 Topological Sorting

The problem of topological sorting can be given as follows:

Given an acyclic directed graph, find a total strict-order $<$ on the nodes such that if there is an edge from node a to node b then $a < b$ holds as well.

A finite total order on the nodes can be conveniently described by a repetition-free word comprising all the nodes, taking $<$ as the relation "occurs before". We can give an inductive definition of this relation as

$$\begin{aligned} \mathit{bef}(\varepsilon) &\overset{\text{def}}{=} \emptyset \ , \\ \mathit{bef}(a \bullet s) &\overset{\text{def}}{=} a \bullet \mathsf{set}\, s \ \cup \ \mathit{bef}(s) \ . \end{aligned}$$

Now we can specify the topological sortings of a directed graph with node set A and acyclic edge relation $R \subseteq A \bullet A$ by

$$\mathit{topsort}(R) \overset{\text{def}}{=} \{s : s \in \mathit{perms}(A) \wedge R \subseteq \mathit{bef}(s)\} \ .$$

Since we require a permutation of the set A of all nodes, also isolated nodes are covered.

One possibility to obtain a recursive solution is to exhaust in some fashion the set A of nodes. We therefore generalise the problem to deal with arbitrary subsets $S \subseteq A$:

$$\mathit{tops}(S, R) \overset{\text{def}}{=} (R \subseteq S \bullet S) \bullet \{s : s \in \mathit{perms}(S) \wedge R \subseteq \mathit{bef}(s)\} \ ,$$

with the embedding $\mathit{topsort}(R) = \mathit{tops}(A, R)$.

Suppose now $R \subseteq S \bullet S$. If $S = \emptyset$, then $\mathit{tops}(\emptyset, R) = \varepsilon$. If $S \neq \emptyset$, choose an arbitrary $s \in \mathit{tops}(S, R)$. Since $s \in \mathit{perms}(S)$ we have $s \neq \varepsilon$, and $s = a \bullet t$ for some $a \in S$ and $t \in \mathit{perms}(S \backslash a)$, so that $\mathsf{set}\, t = S \backslash a$. We want to characterise a and t. First, from (7) it follows that

$$a \bowtie \mathsf{set}\, t = \emptyset = \mathsf{set}\, t \bowtie a \ \wedge \ \overline{a} \bowtie \mathsf{set}\, t = \mathsf{set}\, t = \mathsf{set}\, t \bowtie \overline{a} \ . \tag{14}$$

Let us now investigate the relation between a and R. An easy induction shows $\mathit{bef}(t) \subseteq \mathsf{set}\, t \bullet \mathsf{set}\, t$, which implies, by the definition of bef, that $R \subseteq a \bullet \mathsf{set}\, t \ \cup$ $\mathsf{set}\, t \bullet \mathsf{set}\, t$. Using monotonicity, distributivity and (7, 14) we obtain from this

$$a \bowtie R \subseteq a \bullet \mathsf{set}\, t \ \wedge \ \overline{a} \bowtie R \subseteq \mathsf{set}\, t \bullet \mathsf{set}\, t \ \wedge \ R \bowtie a \subseteq \emptyset \tag{15}$$

with $\overline{a} \overset{\text{def}}{=} A\backslash a$. This, in turn, implies

$$R \bowtie \overline{a} = R \ . \tag{16}$$

Hence

$$\begin{array}{ll}
 & R \subseteq \mathit{bef}(a \bullet t) \\
\Leftrightarrow & \{\!\![\text{ definition of } \mathit{bef} \ ]\!\!\} \\
 & R \subseteq a \bullet \mathsf{set}\, t \ \cup \ \mathit{bef}(t) \\
\Leftrightarrow & \{\!\![\text{ monotonicity, (14, 15) and Boolean algebra }]\!\!\} \\
 & a \bowtie R \subseteq a \bullet \mathsf{set}\, t \wedge \overline{a} \bowtie R \subseteq \mathit{bef}(t) \\
\Leftrightarrow & \{\!\![\text{ by (15) }]\!\!\} \\
 & \overline{a} \bowtie R \subseteq \mathit{bef}(t) \ .
\end{array}$$

Moreover, by the definition of *perms* we have $a \bullet t \in \mathit{perms}(S) \Leftrightarrow t \in \mathit{perms}(S\backslash a)$; also, it is easily checked that $\overline{a} \bowtie R \subseteq (S\backslash a) \bullet (S\backslash a)$. Altogether,

$$a \bullet t \in \mathit{tops}(S, R) \ \Rightarrow \ R \bowtie a = \emptyset \ \wedge \ t \in \mathit{tops}(S\backslash a, \overline{a} \bowtie R) \ .$$

Hence we are close to a recursion for *topsort*. We only need to check whether the necessary condition also is sufficient. So suppose now $R \bowtie a = \emptyset$ and $t \in \mathit{tops}(S\backslash a, \overline{a} \bowtie R)$. We need to show $R \subseteq \mathit{bef}(a \bullet t)$.

$$\begin{array}{ll}
 & R \\
= & \{\!\![\text{ Boolean algebra and distributivity }]\!\!\} \\
 & a \bowtie R \ \cup \ \overline{a} \bowtie R \\
= & \{\!\![\text{ by (16) }]\!\!\} \\
 & a \bowtie R \bowtie \overline{a} \ \cup \ \overline{a} \bowtie R \\
\subseteq & \{\!\![\text{ by } R \subseteq S \bullet S \text{ and monotonicity }]\!\!\} \\
 & a \bowtie S \bullet S \bowtie \overline{a} \ \cup \ \overline{a} \bowtie R \\
= & \{\!\![\text{ by } a \in S, \text{ (7) and Boolean algebra }]\!\!\} \\
 & a \bullet (S\backslash a) \ \cup \ \overline{a} \bowtie R \\
\subseteq & \{\!\![\text{ by } t \in \mathit{tops}(S\backslash a, \overline{a} \bowtie R) \ ]\!\!\} \\
 & a \bullet \mathsf{set}\, t \ \cup \ \mathit{bef}(t) \ .
\end{array}$$

Summing up, we have shown the recursion relation

$$a \bullet t \in \mathit{tops}(S, R) \ \Leftrightarrow \ a \in S \wedge R \bowtie a = \emptyset \wedge t \in \mathit{tops}(S\backslash a, \overline{a} \bowtie R) \ .$$

To relate this to the previous section, we observe that

$$a \in S \wedge R \bowtie a = \emptyset \ \Leftrightarrow \ a \in S\backslash(S \,;R) \ \Leftrightarrow \ a \in \mathit{src}(S, R) \ .$$

Since R is assumed to be acyclic, we know from (13) that $S \neq \emptyset \Rightarrow src(S,R) \neq \emptyset$. Moreover, Lemma 4(b) and monotonicity show that the assumption of cyclefreeness also holds for $\overline{a} \bowtie R \subseteq R$. Altogether we have the (obviously terminating) recursion

$$tops(S,R) = \mathsf{if}\ S = \emptyset\ \mathsf{then}\ \varepsilon\ \mathsf{else} \bigcup_{a \in src(S,R)} a \bullet tops(S \backslash a,\ \overline{a} \bowtie R)\ \mathsf{fi}\ .$$

Again, the repeated *src*-computations are inefficient. The remedy, as in the previous algorithm, lies in carrying along an incrementally updated vector of in-degrees.

5 Sorting Algorithms

We now want to derive several sorting algorithms from a common specification.

5.1 Formal Specification of the Sorting Problem

The task of sorting can be formulated as follows:

Given an alphabet A with a total order $\leq$ on it and a word $s \in A^{(*)}$, find a permutation of s which is sorted in ascending order with respect to $\leq$.

Using our operations, we can formalize this by defining

$$sort(s) \overset{\text{def}}{=} permw(s) \cap \leq^{\rightsquigarrow}\ .$$

In other words, $sort(s)$ is specified as the set of all ordered permutations of s.

5.2 Mergesort

We use the definition of the permutation set to calculate $sort(\varepsilon) = \varepsilon$, $sort(a) = a$ and

$$\begin{array}{ll}
 & sort(s \bullet t) \\
= & \{\!\![\ \text{definition}\ ]\!\!\} \\
 & permw(s \bullet t) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\ \text{by (2)}\ ]\!\!\} \\
 & (permw(s) \,|||\, permw(t)) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\ \text{by Lemma 3}\ ]\!\!\} \\
 & ((permw(s) \cap \leq^{\rightsquigarrow}) \,|||\, (permw(t) \cap \leq^{\rightsquigarrow})) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\ \text{definition}\ ]\!\!\} \\
 & (sort(s) \,|||\, sort(t)) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\ \text{abbreviation}\ ]\!\!\} \\
 & merge(sort(s), sort(t))\ ,
\end{array}$$

where, for $s, t \in \leq^{\rightsquigarrow}$,

$$merge(s,t) \overset{\text{def}}{=} (s \cup t \subseteq \leq^{\rightsquigarrow}) \bullet ((s \,|||\, t) \cap \leq^{\rightsquigarrow}) \ .$$

From the definition it is immediate that *merge* is commutative and associative with neutral element ε. This gives us the base cases for the recursion. Moreover, we calculate, assuming $a \bullet s, b \bullet t \in \leq^{\rightsquigarrow}$,

$$\begin{array}{cl}
& merge(a \bullet s, b \bullet t) \\
= & \{\!\![\text{ definition }]\!\!\} \\
& ((a \bullet s) \,|||\, (b \bullet t)) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\text{ definition }]\!\!\} \\
& (a \bullet (s \,|||\, (b \bullet t)) \ \cup \ b \bullet ((a \bullet s) \,|||\, t)) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\text{ distributivity }]\!\!\} \\
& (a \bullet (s \,|||\, (b \bullet t)) \cap \leq^{\rightsquigarrow}) \ \cup \ (b \bullet ((a \bullet s) \,|||\, t) \cap \leq^{\rightsquigarrow}) \ .
\end{array}$$

Now we treat the first summand, the second one being symmetric. Again we abbreviate $S \bullet T \subseteq \leq$ by $S \leq T$ (for $S, T \subseteq A$).

$$\begin{array}{cl}
& a \bullet (s \,|||\, (b \bullet t)) \cap \leq^{\rightsquigarrow} \\
= & \{\!\![\text{ by Corollary 2 }]\!\!\} \\
& (a \leq \mathsf{set}\,(s \,|||\, (b \bullet t))) \bullet a \bullet ((s \,|||\, (b \bullet t)) \cap \leq^{\rightsquigarrow}) \\
= & \{\!\![\text{ definition of } merge \ ]\!\!\} \\
& (a \leq \mathsf{set}\,(s \,|||\, (b \bullet t))) \bullet a \bullet merge(s, b \bullet t) \\
= & \{\!\![\text{ by (1) and distributivity }]\!\!\} \\
& (a \leq \mathsf{set}\, s \ \wedge \ a \leq \mathsf{set}\,(b \bullet t)) \bullet a \bullet merge(s, b \bullet t) \\
= & \{\!\![\text{ by } a \bullet s, b \bullet t \in \leq^{\rightsquigarrow} \text{ and transitivity }]\!\!\} \\
& (a \leq b) \bullet a \bullet merge(s, b \bullet t) \\
= & \{\!\![\text{ definition }]\!\!\} \\
& (a \leq b) \bullet a \bullet merge(s, b \bullet t) \ .
\end{array}$$

Combining the two summands we therefore obtain

$$\begin{array}{rl}
merge(a \bullet s, b \bullet t) = & \mathsf{if} \ a \leq b \,\mathsf{then}\, a \bullet merge(s, b \bullet t) \\
& [\,]\ b \leq a \,\mathsf{then}\, b \bullet merge(a \bullet s, t) \ \mathsf{fi} \ .
\end{array}$$

5.3 Quicksort

This time we use a different way of splitting nonempty words and calculate

$$
\begin{array}{ll}
 & \mathit{sort}(s \bullet a \bullet t) \\
= & \{\![\text{ definition }]\!\} \\
 & \mathit{permw}(s \bullet a \bullet t) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ by (2) }]\!\} \\
 & (\mathit{permw}(s) \mathbin{|||} a \mathbin{|||} \mathit{permw}(t)) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ commutativity }]\!\} \\
 & (a \mathbin{|||} \mathit{permw}(s) \mathbin{|||} \mathit{permw}(t)) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ by (2) }]\!\} \\
 & (a \mathbin{|||} \mathit{permw}(s \bullet t)) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ abbreviation }]\!\} \\
 & \mathit{insert}(a, s \bullet t)\ ,
\end{array}
$$

where

$$\mathit{insert}(a, s) \overset{\text{def}}{=} (a \mathbin{|||} \mathit{permw}(s)) \cap \leq^{\rightsquigarrow} \quad .$$

Now we obtain

$$
\begin{array}{ll}
 & \mathit{insert}(a, s) \\
= & \{\![\text{ definition }]\!\} \\
 & (a \mathbin{|||} \mathit{permw}(s)) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ by (3) }]\!\} \\
 & (\bigcup\limits_{u \bullet v \in \mathit{permw}(s)} \mathit{permw}(u) \bullet a \bullet \mathit{permw}(v)) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ distributivity }]\!\} \\
 & \bigcup\limits_{u \bullet v \in \mathit{permw}(s)} \mathit{permw}(u) \bullet a \bullet \mathit{permw}(v) \cap \leq^{\rightsquigarrow} \\
= & \{\![\text{ by (11) }]\!\} \\
 & \bigcup\limits_{u \bullet v \in \mathit{permw}(s)} (\mathit{permw}(u) \bullet a \cap \leq^{\rightsquigarrow}) \bowtie (a \bullet \mathit{permw}(v) \cap \leq^{\rightsquigarrow}) \\
= & \{\![\text{ by Corollary 2 and its dual }]\!\} \\
 & \bigcup\limits_{u \bullet v \in \mathit{permw}(s)} ((\mathit{permw}(u) \cap \leq^{\rightsquigarrow}) \bullet a \bullet (\mathsf{set}\, u \leq a)) \bowtie \\
 & \qquad ((a \leq \mathsf{set}\, v) \bullet a \bullet (\mathit{permw}(v) \cap \leq^{\rightsquigarrow})) \\
= & \{\![\text{ definition of } \mathit{sort}\text{, shifting assertions }]\!\} \\
 & \bigcup\limits_{u \bullet v \in \mathit{permw}(s)} (\mathsf{set}\, u \leq a) \bullet (a \leq \mathsf{set}\, v) \bullet ((\mathit{sort}(u) \bullet a) \bowtie (a \bullet \mathit{sort}(v))) \\
= & \{\![\text{ definition of join }]\!\}
\end{array}
$$

$$\bigcup_{u \bullet v \in permw(s)} (\mathsf{set}\, u \le a) \bullet (a \le \mathsf{set}\, v) \bullet (sort(u) \bullet a \bullet sort(v)) \ .$$

Altogether, defining

$$split(s, a) \overset{\text{def}}{=} \bigcup_{u \bullet v \in permw(s)} (\mathsf{set}\, u \le a) \bullet (a \le \mathsf{set}\, v) \bullet \langle u, v \rangle \ ,$$

where $\langle u, v \rangle$ denotes the pair consisting of u and v, we have

$$\begin{aligned} sort(\varepsilon) &= \varepsilon \ , \\ sort(s \bullet a \bullet t) &= \bigcup_{\langle u,v \rangle \in split(s \bullet t, a)} sort(u) \bullet a \bullet sort(v) \ . \end{aligned} \tag{17}$$

This is the quicksort algorithm.

5.4 Treesort

We now also consider binary trees. The empty binary tree is denoted by $\Box$ and the composition of two binary trees l, r and a node value a by the triple $\langle l, a, r \rangle$. The inorder traversal of a binary tree is defined by

$$\begin{aligned} inord(\Box) &\overset{\text{def}}{=} \varepsilon \\ inord(\langle l, a, r \rangle) &\overset{\text{def}}{=} inord(l) \bullet a \bullet inord(r) \ . \end{aligned}$$

Now we specify the sorted trees representing a word s as

$$trees(s) \overset{\text{def}}{=} \{b : inord(b) \in sort(s)\} \ .$$

From this it is easily calculated that $trees(\varepsilon) = \Box$ and

$$s \ne \varepsilon \Rightarrow \Box \notin trees(s) \ . \tag{18}$$

Moreover,

$$\begin{aligned} & trees(s \bullet a \bullet t) \\ = \quad & \{\!\![\text{ definition, (18), (17) }]\!\!\} \\ & \{\langle l, b, r \rangle : inord(\langle l, b, r \rangle) \in \bigcup_{\langle u,v \rangle \in split(s \bullet t, a)} sort(u) \bullet a \bullet sort(v)\} \\ = \quad & \{\!\![\text{ commuting quantifiers, definition }]\!\!\} \\ & \bigcup_{\langle u,v \rangle \in split(s \bullet t, a)} \{\langle l, b, r \rangle : inord(l) \bullet b \bullet inord(r) \in sort(u) \bullet a \bullet sort(v)\} \\ \supseteq \quad & \{\!\![\text{ specializing } b \text{ to } a \ ]\!\!\} \\ & \bigcup_{\langle u,v \rangle \in split(s \bullet t, a)} \{\langle l, a, r \rangle : inord(l) \bullet a \bullet inord(r) \in sort(u) \bullet a \bullet sort(v)\} \\ \supseteq \quad & \{\!\![\text{ specializing } l \text{ and } r \ ]\!\!\} \end{aligned}$$

$$\bigcup_{\langle u,v\rangle \in split(s\bullet t,a)} \{\langle l,a,r\rangle : inord(l) \in sort(u) \wedge inord(r) \in sort(v)\}$$

$$= \quad \{\![\text{ folding }]\!\}$$

$$\bigcup_{\langle u,v\rangle \in split(s\bullet t,a)} \langle trees(u), a, trees(v)\rangle \ .$$

Note that proper descendant steps are involved. Altogether,

$$\begin{aligned} trees(\varepsilon) &= \square \ , \\ trees(s \bullet a \bullet t) &\supseteq \bigcup_{\langle u,v\rangle \in split(s\bullet t,a)} \langle trees(u), a, trees(v)\rangle \ . \end{aligned}$$

In a similar manner algorithms for inserting into and deleting from sorted trees can be derived. This is, on a less algebraic basis, discussed in [6].

6 Conclusion

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

Different sets of operators have been used to derive algorithms on pointer structures (see [15, 14]) and binary search trees (see [6]). A long term goal is the construction of a database of such 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, 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)
2. 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
3. R.M. Burstall, J. Darlington: A transformation system for developing recursive programs. J. ACM **24**, 44–67 (1977)

4. J. Cai, R. Paige: Program derivation by fixed point computation. Science of Computer Programming **11**, 197–261 (1989)
5. J.H. Conway: Regular algebra and finite machines. London: Chapman and Hall 1971
6. W. Dosch, B. Möller: Calculating a module for binary search trees. GI-Jahrestagung 1993 (to appear)
7. M.S. Feather: A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens (ed.): Proc. IFIP TC2 Working Conference on Program Specification and Transformation, Bad Tölz, April 14–17, 1986. Amsterdam: North-Holland1987, 165–195
8. P. Lescanne: Modèles non déterministes de types abstraits. R.A.I.R.O. Informatique théorique **16**, 225–244 (1982)
9. Z. Manna: Mathematical theory of computation. New York: McGraw-Hill 1974
10. 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
11. 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
12. B. Möller: Relations as a program development language. In [13], 373–397
13. B. Möller (ed.): Constructing programs from specifications. Proc. IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13–16 May 1991. Amsterdam: North-Holland 1991, 373–397
14. B. Möller: Towards pointer algebra. Institut für Mathematik der Universität Augsburg, Report No. 279, 1993. Also to appear in Science of Computer Programming
15. B. Möller: Derivation of graph and pointer algorithms. Institut für Mathematik der Universität Augsburg, Report No. 280, 1993. Also to appear in B. Möller, H.A. Partsch, S.A. Schuman (eds.): Formal program development. Proc. of an IFIP TC2/WG 2.1 State of the Art Seminar. Lecture Notes in Computer Science. Berlin: Springer (to appear)
16. B. Möller, M. Russling: Shorter paths to graph algorithms. Proc. 1992 International Conference on Mathematics of Program Construction (to appear). Extended version: Institut für Mathematik der Universität Augsburg, Report Nr. 272, 1992. Also to appear in Science of Computer Programming
17. H.A. Partsch: Specification and transformation of programs — A formal approach to software development. Berlin: Springer 1990
18. M. Russling: Hamiltonian sorting. Institut für Mathematik der Universität Augsburg, Report Nr. 270, 1992
19. G. Schmidt, T. Ströhlein: Relations and graphs. Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Berlin: Springer 1993