Reprinted from: M. Broy (ed.): Calculational system design. International Summer School Mark- toberdorf, July 28 - August 9, 1998. NATO Advanced Science Institutes Series. Subseries F: Computer and Systems Sciences. Amsterdam: IOS Press, with permission from IOS Press. The publication is available at IOS Press.

Algebraic Structures for Program Calculation

Bernhard MÖLLER
Institut für Informatik, Universität Augsburg, D-86135 Augsburg, Germany

Abstract. In calculational program design one derives implementations from specifications using semantics-preserving deduction rules. The aim of modern algebraic approaches is to make both specification and calculation more concise and perspicuous by compacting logic into algebra as much as possible. We present a collection of algebraic calculi that are useful in this activity. They are shown at work in a number of examples that range from graph algorithms over stream processing to hardware description.

1 Introduction

In calculational program design one derives implementations from specifications using semantics-preserving deduction rules. The aim of modern algebraic approaches is to make both specification and calculation more concise and perspicuous by compacting logic into algebra as much as possible. The essential aims are:

- To package frequently occurring shapes of formulae, notably larger aggregations of quantifiers, into algebraic operators and to prove strong equational or inequational laws for them. Such a law usually compacts a series of inference steps in pure predicate calculus into a single one.
- To make the formulae involved more concise and less repetitious. Then writing them will be less error-prone, in particular, since copying mistakes are reduced, and reading and understanding them will be easier and quicker.
- To raise the level of discourse in formal specification and derivation close to that of informal reasoning, to achieve formality and understandability at the same time.

Of course, rather than developing an algebraic theory anew for each application, one seeks to re-use existing and well-studied ones. Important examples of algebraic structures for calculational program design are the following:

- The *Relational Calculus* provides the algebra of binary relations.
- *Kleene Algebras* capture notions around finite iteration.
- *Network Algebra* is the algebraic theory of directed graphs, such as circuit diagrams or data flow graphs. In particular, it contains an algebra of feedback.
- *Fixpoint Calculus* is a set of rules for transforming recursive definitions.
- *Galois Connections* allow concise definitions of many notions in partially ordered structures and give several useful calculation properties for free.

This paper illustrates various algebraic notions with derivations of graph algorithms, stream-based systems and sequential hardware. For further examples in this area and omitted proofs see [46, 55, 54]. It turns out that the algebraic approach can also be extended to the field of pointer algorithms (see [47, 57, 9]).

It should be noted that the main focus of the present paper is not on the particular examples treated here but on the "algebraic spirit". Also, we do not present a single calculus, but a *collection* of various useful ones. Still we aim at notational uniformity as much as possible. Next, we use a combination of point-free and pointwise reasoning, since each of these two approaches has its own advantages and disadvantages in specific circumstances. Our main message may be stated, with tongue in cheek, as: "There is more algebra in heaven and earth than is dreamt of in your philosophy." — Just dare to discover and use it!

2 Formal Languages, Relations and Graphs

2.1 Groupoids, Monoids and Languages

Consider a non-empty set M. An **inner operation** on M is a (total) function $\circ : M \times M \to M$. The pair $(M, \circ)$ is called a **groupoid**. An element $x \in M$ is called **neutral** if $\forall\, y \in M : x \circ y = y = y \circ x$. It is called **absorbing** iff $\forall\, y \in M : x \circ y = x = y \circ x$. Sometimes neutral or absorbing elements are called **ones** or **zeros**, resp. From the definition it is clear that for every operation there is at most one neutral and at most one absorbing element.

A **semigroup** is a groupoid $(M, \circ)$ in which the operation $\circ$ is associative. A **monoid** is a triple $(M, \circ, e)$ such that $(M, \circ)$ is a semigroup and e is neutral w.r.t. $\circ$. For a sequence $(x_i)_{i\in[0,n-1]}$ of monoid elements $x_i \in M$ we define the iterated monoid operation by

$$\bigcirc_{i=0}^{n-1} x_i \stackrel{\text{def}}{=} \begin{cases} e & \text{if } n = 0\ , \\ (\bigcirc_{i=0}^{n-2} x_i) \circ x_{n-1} & \text{otherwise}\ . \end{cases}$$

Because of associativity one has then $\bigcirc_{i=0}^{n-1} x_i = x_0 \circ \cdots \circ x_{n-1}$. If for some $i_0 \in [0, n-1]$ the element x_{i_0} is absorbing, then $\bigcirc_{i=0}^{n-1} x_i = x_{i_0}$. The powers x^n for $x \in M$ and $n \in \mathbb{N}$ are defined as $x^n \stackrel{\text{def}}{=} \bigcirc_{i=0}^{n-1} x$.

Consider now 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 $A^+ \stackrel{\text{def}}{=} A^* \backslash \varepsilon$. By $\bullet$ we denote the concatenation operation A^*. Then $(A^*, \bullet, \varepsilon)$ is a monoid.

2.2 Pointwise Extension

Many of our operations on sets (e.g. on languages, see below) are first defined for single elements and then extended pointwise to sets. We explain this mechanism for a unary operation on a set M; the extension to multiary ones is straightforward. Since we also need partial operations, we choose $\mathcal{P}(M)$, the power set of M, as the codomain for such

an operation. The operation will then return a singleton set consisting of the result, if this is defined, and the empty set $\emptyset$ otherwise. Thus $\emptyset$ plays the role of the error "value" $\perp$ in denotational semantics. Consider now such an operation $f : M \to \mathcal{P}(M)$. Then the **pointwise extension** of f is denoted again by f, has the functionality $f : \mathcal{P}(M) \to \mathcal{P}(M)$ and is defined by

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

for $N \subseteq M$. By this definition, the extended operation distributes through union:

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

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

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

Moreover, taking $\mathcal{N} = \{N, P\}$ and using the equivalence $N \subseteq P \ \Leftrightarrow \ N \cup P = P$, we also obtain monotonicity with respect to $\subseteq$:

$$N \subseteq P \ \Rightarrow \ f(N) \subseteq f(P) \ .$$

Monotonicity is crucial for least fixpoint semantics of recursion (see Section 3.2).

Because of $\subseteq$-monotonicity and $N \cap P \subseteq N \wedge N \cap P \subseteq P$, all pointwise extended functions are **subdistributive** over intersection:

$$f(N \cap P) \subseteq f(N) \cap f(P) \ .$$

Pointwise extensions inherit **linear laws** (see e.g. [41, 22]). These are laws of the following forms:

- Equational laws in which all variables occur exactly once on both sides of the equality sign. Examples are the laws of neutrality, associativity and commutativity.
- Implications with element relations as atoms in which all variables occur exactly once on both sides of the implication sign. In the inherited form the variables for elements turn into variables for non-empty sets and the element relations turn into inclusions. An example is

  $$s \bullet t \in \varepsilon \Rightarrow s \in \varepsilon \wedge t \in \varepsilon$$

  which lifts to

  $$S \neq \emptyset \wedge T \neq \emptyset \wedge S \bullet T \subseteq \varepsilon \ \Rightarrow \ S \subseteq \varepsilon \wedge T \subseteq \varepsilon \ .$$

Consider now a monoid $(M, \circ, e)$. If we extend $\circ$ pointwise to $\mathcal{P}(M)$ then $(\mathcal{P}(M), \circ, \{e\})$ is again a monoid with absorbing element $\emptyset$. The powers are then also used on subsets $N \subseteq M$, ie. $N^k = \underbrace{N \circ \cdots \circ N}_{k}$.

2.3 Relations

2.3.1 Basic Notions

A **relation of arity** n is a language $R \subseteq A^n$. In particular, the empty language $\emptyset$ is a relation of any arity. There are only two 0-ary relations, viz. $\emptyset$ and ε. The **negation** of relation $R \subseteq A^n$ is

$$\neg R \stackrel{\text{def}}{=} A^n \backslash R \ .$$

Of particular interest are **binary relations**, ie. relations of arity 2. Instead of $a \bullet b \in R$ one also writes $a\,R\,b$. The **universal relation** $A \bullet A$ is also denoted $\top$. 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 **reverse** $u^{\smile}$ of a word $u \in A^*$ is defined inductively by

$$\begin{array}{rcll} \varepsilon^{\smile} & = & \varepsilon \ , & \\ a^{\smile} & = & a & (a \in A) \ , \\ (u \bullet v)^{\smile} & = & v^{\smile} \bullet u^{\smile} \ . & \end{array}$$

From this it follows that

$$(u^{\smile})^{\smile} = u \ .$$

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

The reversal of a binary relation $R \subseteq A \bullet A$ is also called the **converse** of R. One has $I_B^{\smile} = I_B$ and $\top^{\smile} = \top$.

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

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

and, again, extended pointwise to languages.

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

$$\begin{array}{rcl} \mathsf{dom}\,\varepsilon = \emptyset = \mathsf{cod}\,\varepsilon \ , & & \\ \mathsf{dom}\,(a \bullet u) = a & & \mathsf{cod}\,(v \bullet b) = b \end{array}$$

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: $\mathsf{dom}\,R$ is the domain of R, whereas $\mathsf{cod}\,R$ is the codomain of R. Moreover, for binary R we have $\mathsf{set}\,R = \mathsf{dom}\,R \cup \mathsf{cod}\,R$.

As unary operators, dom, cod and set bind strongest.

To end this section, we briefly show how to represent graphs by relations, since this will be used in examples in further sections. We will describe the node set by the alphabet A we use. Then paths may be modeled by words from A^* and edges by words in $A \bullet A$. Path sets are given by formal languages $P \subseteq A^*$, whereas the graph itself is determined by its set of edges, ie. a binary relation $R \subseteq A \bullet A$.

2.3.2 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 "gluing" 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. [33]). On binary relations, composition coincides with usual relational composition (see e.g. [74, 66]). 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

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

Thus, the relation $R \bowtie R$ consists of exactly those paths $a \bullet b \bullet c$ which result from gluing 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 nodes.

The operations associate nicely with each other and with concatenation:

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

The conditions in the case of qualified laws are due to the fact that composition "loses information" in that it throws the connecting node away. It is a useful exercise to find counterexamples for the cases where the conditions do not hold. 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

$$\begin{aligned} R \bowtie S &= \{a \bullet u : a \in R \wedge a \bullet u \in S\} \ , \\ R \,;\, S &= \{u : \exists\, a \in R : a \bullet u \in S\} \ . \end{aligned}$$

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

$$\mathsf{dom}\, R \;=\; R \,;\, A \qquad \mathsf{cod}\, R \;=\; A \,;\, R \ . \tag{2}$$

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 $R \neq \emptyset \Rightarrow \top \,;\, R \,;\, \top = \top$ (see [74, 66]).
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 $R, S \subseteq A \bullet A$,

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

We also have neutral elements for join and composition. Assume $A \supseteq P \supseteq \mathsf{dom}\, V$ and $A \supseteq Q \supseteq \mathsf{cod}\, V$ 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

$$P \,;\, (Q \bowtie R) \;=\; (P \bowtie Q) \,;\, R \ , \qquad (R \bowtie P) \,;\, Q \;=\; R \,;\, (P \bowtie Q) \ . \tag{8}$$

Join and composition are contravariant w.r.t. reversal:

$$(R \,;\, S)^{\smile} = S^{\smile} ; R^{\smile} \ , \qquad (R \bowtie S)^{\smile} = S^{\smile} \bowtie R^{\smile} \ . \tag{9}$$

As pointwise extensions, they distribute through arbitrary unions, and hence are monotonic and strict, and are subdistributive over intersection.

2.3.3 Particular Relations

We now introduce some particular properties of relations. A relation $R \subseteq A \bullet A$ is called

–**reflexive**	iff	$I_A \subseteq R$,
–**coreflexive**	iff	$R \subseteq I_A$,
–**irreflexive**	iff	$R \subseteq \neg I_A$,
–**transitive**	iff	$R ; R \subseteq R$,
–**dense**	iff	$R \subseteq R ; R$,
–**symmetric**	iff	$R \subseteq R^\smile$,
–**antisymmetric**	iff	$R \cap R^\smile \subseteq I_A$,
–**asymmetric**	iff	$R \cap R^\smile = \emptyset$.

For a symmetric relation it follows that even $R = R^\smile$.

A relation $\leq \subseteq A \bullet A$ is called a **pre-order**, iff it is reflexive and transitive. A pre-order is called a **(partial) order** iff it is antisymmetric, and an **equivalence relation** iff it is symmetric. A pre-order $\leq$ is called **linear** or **total** iff any two elements are comparable, ie. iff $\top \subseteq \leq \cup \leq^\smile$. Frequently also the pair $(A, \leq)$ is called a pre-order or order. Finally, a relation $<$ is called a **strict-order** iff it is irreflexive and transitive. Note that a strict-order is asymmetric.

Further properties of relations concern their connection to functions or mappings. Note that

– $x\,(R ; R^\smile)\,y$ iff x and y have a common image under R,
– $x\,(R^\smile ; R)\,y$ iff x and y have a common inverse image under R.

Hence R is called

– **total** iff $I_A \subseteq R ; R^\smile$,
– **surjective** iff $I_A \subseteq R^\smile ; R$,
– **injective** iff $R ; R^\smile \subseteq I_A$,
– **deterministic** or a **partial map** if $R^\smile ; R \subseteq I_A$.

2.4 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 expressions yielding nullary relations **assertions**. In our use of assertions we shall be quite liberal and allow also expressions like $R \subseteq S$ as assertions although they may not easily be formulated in the algebra of formal languages and relations.

Note that in this view "false" and "undefined" both are represented by $\emptyset$. Negation is defined as in Section 2.3.1, amounting to

$$\neg\emptyset = \varepsilon \ , \qquad \neg\varepsilon = \emptyset \ .$$

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

From this it should be clear that 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 $B \cup C$. Using assertions we can also define a conditional by

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

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.

2.5 Example: Topological Sorting

We shall now use the concepts introduced so far to derive a simple algorithm for sorting a graph topologically. The problem can be stated as follows:

Given an acyclic directed graph, find a linear 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 linear order on the nodes can be conveniently described by a repetition-free word s comprising all the nodes, taking for $<$ the relation $bef(s)$ with $a\, bef(s)\, b$ iff a occurs before b in s. We can give an inductive definition of this relation by

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

Repetition-free words are also known as **permutations** of the set of letters they contain. An inductive definition of the set of all permutations of a finite set of letters is

$$\begin{aligned} perms(\emptyset) &\stackrel{\text{def}}{=} \varepsilon\ , \\ perms(S) &\stackrel{\text{def}}{=} \bigcup_{a \in S} a \bullet perms(S \backslash a)\ , \end{aligned} \tag{10}$$

if $S \subseteq A$ is finite and $S \neq \emptyset$.

It is an instructive exercise to use relational calculation for showing that, for $s \in perms(S)$, the relation $bef(s)$ is irreflexive and asymmetric, ie. a strict-order.

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

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

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. For this we employ the **principle of generalisation** (see e.g. [62]): Replace a constant by a parameter or "invent" suitable additional parameters. The generalisation gives an additional degree of freedom which can be exploited to obtain a recursion when the original problem was too specialised. The original problem is then retrieved via an **embedding** as a special case of the generalised problem.

Here we generalise the problem to the case of topologically sorting arbitrary subsets $S \subseteq A$, not just A itself. Then only the restriction of R to S need be considered:

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

Our original function is retrieved as a special case by the embedding $topsort(R) = tops(A, R)$.

If $S = \emptyset$, then $tops(S, R) = \varepsilon$.

If $S \neq \emptyset$, choose an arbitrary $s \in tops(S, R)$. Since $s \in perms(S)$ we have $s \neq \varepsilon$, and $s = a \bullet t$ for some $a \in S$ and $t \in perms(S \backslash a)$, so that $\mathsf{set}\, t = S \backslash a$ and

$$bef(t) \subseteq (S \backslash a) \bullet (S \backslash a) \; . \tag{11}$$

In particular,

$$bef(a \bullet t) \cap S \bullet a = \emptyset \; . \tag{12}$$

We want to find a characterisation of a. We calculate

$$\begin{array}{ll} & R \cap S \bullet S \subseteq bef(a \bullet t) \\ \Leftrightarrow & \quad \{\![\text{ using } S = a \cup (S \backslash a) \text{ and distributivity }]\!\} \\ & (R \cap S \bullet a) \cup (R \cap S \bullet (S \backslash a)) \subseteq bef(a \bullet t) \\ \Leftrightarrow & \quad \{\![\text{ by Boolean algebra and (12) }]\!\} \\ & R \cap S \bullet a = \emptyset \wedge \\ & R \cap S \bullet (S \backslash a) \subseteq bef(a \bullet t) \; . \end{array}$$

The second conjunct can be transformed as follows:

$$\begin{array}{ll} & R \cap S \bullet (S \backslash a) \subseteq bef(a \bullet t) \\ \Leftrightarrow & \quad \{\![\text{ using } S = (S \backslash a) \cup a, \text{ distributivity and Boolean algebra }]\!\} \\ & R \cap (S \backslash a) \bullet (S \backslash a) \subseteq bef(a \bullet t) \wedge \\ & R \cap a \bullet (S \backslash a) \subseteq bef(a \bullet t) \\ \Leftrightarrow & \quad \{\![\text{ since } a \bullet (S \backslash a) \subseteq bef(a \bullet t) \text{ by definition of } bef \;]\!\} \\ & R \cap (S \backslash a) \bullet (S \backslash a) \subseteq bef(a \bullet t) \\ \Leftrightarrow & \quad \{\![\text{ definition of } bef \;]\!\} \\ & R \cap (S \backslash a) \bullet (S \backslash a) \subseteq a \bullet (S \backslash a) \cup bef(t) \\ \Leftrightarrow & \quad \{\![\text{ by } B \subseteq C \Leftrightarrow B \cap C = B \text{ and } (S \backslash a) \bullet (S \backslash a) \cap a \bullet (S \backslash a) = \emptyset \;]\!\} \\ & R \cap (S \backslash a) \bullet (S \backslash a) \subseteq bef(t) \\ \Leftrightarrow & \quad \{\![\text{ definition of } tops \;]\!\} \\ & t \in tops(S \backslash a, R) \; . \end{array}$$

Let us now define the set $src(S, R)$ by

$$a \in src(S, R) \Leftrightarrow a \in S \wedge R \cap S \bullet a = \emptyset \; .$$

This is the set of **relative sources** of S, ie. the set of nodes of S that do not have an immediate R-predecessor in S. An equivalent definition is

$$src(S, R) \stackrel{\text{def}}{=} S \backslash (S \,;\, R) \; . \tag{13}$$

Then, altogether, we have shown the recursion relation

$$a \bullet t \in tops(S, R) \;\; \Leftrightarrow \;\; a \in src(S, R) \wedge t \in tops(S \backslash a, R) \; .$$

This means that the following (obviously terminating) recursive function correctly solves our task:

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

So far we have not used acyclicity of R. We have not even given a formal definition of that property. This will be done in Section 4.4. Let us just mention here that R is acyclic iff for all $S \subseteq A$ with $S \neq \emptyset$ also $src(S,R) \neq \emptyset$. So the algorithm is even correct for cyclic R: in that case eventually the set of relative sources becomes empty, and by strictness the empty set results overall.

The efficiency of the algorithm can be improved by computing the source sets from an incrementally adjusted vector of in-degrees of the graph nodes. For details about the formal treatment of such vectors see [58].

3 Orders, Fixpoints and Galois Connections

3.1 Lattices and Completeness

Assume a pre-order $(A, \leq)$ and a subset $B \subseteq A$.

– We define the sets $\mathsf{lwb}\, B$ of **lower bounds** and $\mathsf{upb}\, B$ of **upper bounds** of B by

$$\begin{array}{lll} x \in \mathsf{lwb}\, B & \Leftrightarrow & \forall\, y \in B : x \leq y \quad \Leftrightarrow\ x \bullet B \subseteq \leq\ , \\ x \in \mathsf{upb}\, B & \Leftrightarrow & \forall\, y \in B : y \leq x \quad \Leftrightarrow\ B \bullet x \subseteq \leq\ . \end{array}$$

– The sets $\mathsf{lst}\, B$ of **least** and $\mathsf{gst}\, B$ of **greatest** elements of B are given by

$$\mathsf{lst}\, B \stackrel{\mathrm{def}}{=} B \cap \mathsf{lwb}\, B\ , \qquad \mathsf{gst}\, B \stackrel{\mathrm{def}}{=} B \cap \mathsf{upb}\, B\ .$$

– The sets $\mathsf{glb}\, B$ of **infima** or **greatest lower bounds** and $\mathsf{lub}\, B$ the **suprema** or **least upper bounds** of B are

$$\mathsf{glb}\, B \stackrel{\mathrm{def}}{=} \mathsf{gst}\, \mathsf{lwb}\, B\ , \qquad \mathsf{lub}\, B \stackrel{\mathrm{def}}{=} \mathsf{lst}\, \mathsf{upb}\, B\ .$$

In the case of an order $\leq$ these sets of suprema and infima are at most singletons; in this case we again do not distinguish them from their only elements. An important observation is

$$x \leq y \;\Leftrightarrow\; y \;=\; \mathsf{lub}\,(x \cup y) \;\Leftrightarrow\; x \;=\; \mathsf{glb}\,(x \cup y)\ . \tag{14}$$

Of particular importance are orders in which certain infima and suprema always exist. An order $(A, \leq)$ is called a

– **upper semilattice** iff every two-element subset $\{x, y\}$ has a supremum $x \sqcup y$ (an equivalent requirement is that every non-empty finite subset has a supremum),
– **lower semilattice** iff every two-element subset $\{x, y\}$ has an infimum $x \sqcap y$ (an equivalent requirement is that every non-empty finite subset has an infimum),
– **lattice** iff it is both an upper and a lower semilattice,
– **complete lattice** iff every subset has an infimum (and hence also a supremum; the proof is an exercise).

A complete lattice has a least element $\bot \stackrel{\text{def}}{=} \mathsf{glb}\, A = \mathsf{lub}\, \emptyset$ and a greatest element $\top \stackrel{\text{def}}{=} \mathsf{lub}\, A = \mathsf{glb}\, \emptyset$. Of course,every complete lattice is a lattice, but not vice versa.

Example 3.1 Let $| \subseteq \mathbb{N} \bullet \mathbb{N}$ be the divisibility relation. Then $(\mathbb{N}, |)$ is a lattice in which the infimum is the greatest common divisor and the supremum is the lowest common multiple. This lattice is not complete, since no infinite subset of $\mathbb{N}$ has a supremum. □

As an example of a complete lattice we mention

Example 3.2 For arbitrary set N the order $(\mathcal{P}(N), \subseteq)$ is a complete lattice in which for $\mathcal{P} \subseteq \mathcal{P}(N)$ one has

$$\mathsf{lub}\, \mathcal{P} = \bigcup_{P \in \mathcal{P}} P\ , \qquad \mathsf{glb}\, \mathcal{P} = \bigcap_{P \in \mathcal{P}} P\ .$$

□

Let now $(A_1, \leq_1)$ and $(A_2, \leq_2)$ be lattices. A function $f : A_1 \to A_2$ is **disjunctive** iff for all $x, y \in A_1$ we have $f(x \sqcup y) = f(x) \sqcup f(y)$. If $(A_1, \leq_1)$ and $(A_2, \leq_2)$ are even complete lattices, then f is **universally disjunctive** iff for every subset $P \subseteq A_1$ we have

$$f(\mathsf{lub}\, P) = \mathsf{lub}\, f(P)\ .$$

Every universally disjunctive function is disjunctive, but not vice versa. Again, the notion of **universal conjunctivity** is defined dually.

Let $(A, \leq)$ be an order. A subset $B \subseteq A$ is a **chain** iff it is non-empty and ordered linearly by $\leq$, ie. iff any two of its elements are comparable.

A function $f : A_1 \to A_2$ between orders $(A_1, \leq_1)$ and $(A_2, \leq_2)$ is called **monotonic** if

$$\forall\, x, y \in A_1 : x \leq_1 y \Rightarrow f(x) \leq_2 f(y)\ .$$

By (14) it is immediate that every disjunctive function f is monotonic. In particular, the f-image of a chain is again a chain.

3.2 Fixpoint Theorems and Fixpoint Calculus

We now come to a central theorem for computing science:

Theorem 3.3 (Fixpoint Theorem by Knaster and Tarski) Let $(A, \leq)$ be a complete lattice and let function $f : A \to A$ be monotonic. Then the set $\mathsf{fix}\, f \stackrel{\text{def}}{=} \{x : f(x) = x\}$ of fixpoints of f has a least and a greatest element with

$$\begin{aligned} \mathsf{lst}\, \mathsf{fix}\, f &= \mathsf{glb}\, \{x : f(x) \leq x\}\ , \\ \mathsf{gst}\, \mathsf{fix}\, f &= \mathsf{lub}\, \{x : x \leq f(x)\}\ . \end{aligned}$$

Proof: See e.g. [10]. □

In fact, one can show the even stronger property (see [75]) that $\mathsf{fix}\, f$ itself forms a complete lattice under the induced order. Examples of fixpoint constructions will be given in Sections 4.1 and 5.10.

Instead of $\mathsf{lst\,fix}\, f$ and $\mathsf{gst\,fix}\, f$ one also writes μf and νf, resp. If f is given in the form $f : x \mapsto E$ with some expression E that may depend on x, one writes also $\mu x.E$ and $\nu x.E$ instead of μf and νf, resp.

From the Knaster/Tarski fixpoint theorem we obtain the following inference rules:

$$\frac{f(x) \leq x}{\mu f \leq x} \qquad \frac{x \leq f(x)}{x \leq \nu f}$$

A simple consequence of these rules is that the fixpoint operators are monotonic:

Corollary 3.4 *Let $(A, \leq)$ be a complete lattice and $f, g : A \to A$ two monotonic functions such that for all $x \in A$ one has $f(x) \leq g(x)$. Then $\mu f \leq \mu g$.*

For the formulation of another important proof principle we need further auxiliary notions. Let $(A, \leq)$ be an order. A subset $B \subseteq A$ is **directed** iff every finite subset of B has an upper bound in B. Equivalently, B is directed iff $B \neq \emptyset$ and any two elements in B have a common upper bound in B. Clearly, every chain is directed.

Using chains and directed sets we can formulate a weaker notion of completeness. A **complete partial order (cpo)** is an order $(A, \leq)$ with a least element $\bot$ in which every *directed* set has a supremum. One can show (see e.g. [42]) that $(A, \leq)$ is a cpo iff every chain has a supremum. Of course, every complete lattice is a cpo.

Let now $(A_1, \leq_1)$ and $(A_2, \leq_2)$ be cpos. A function $f : A_1 \to A_2$ is called **continuous** iff for every directed set $P \subseteq A_1$ the supremum $\mathsf{lub\,f(P)}$ exists and

$$f(\mathsf{lub}\, P) \;=\; \mathsf{lub}\, f(P)\ .$$

Note that, contrary to universal disjunctivity, only *directed* subsets are considered here. Function f is called **strict** iff $f(\bot_1) = \bot_2$. Every universally disjunctive function on a complete lattice is continuous and strict.

Example 3.5 Every pointwise extended operation, such as the composition operation on relations, is universally disjunctive and hence continuous and strict. Furthermore, constant functions and the identity function are continuous. □

Again it follows from (14) that every continuous function f is monotonic and the f-image of a directed set is directed again.

Consider now a predicate P on A. We say for $B \subseteq A$ that $P(B)$ **holds** iff $P(x)$ holds for all $x \in B$. Now P is called **continuous** (in the literature also **admissible**) iff for all directed sets $B \subseteq A$ from $P(B)$ we can conclude $P(\mathsf{lub}\, B)$ as well.

Lemma 3.6 *Consider cpos $(A, \leq)$ and $(B, \leq)$ and functions $f, g : A \to B$.*

1. *If f is continuous and g is monotonic then the predicate $P(x) \Leftrightarrow f(x) \leq g(x)$ is continuous.*
2. *If f and g are continuous then also the predicate $R(x) \Leftrightarrow f(x) = g(x)$ is continuous.*

Moreover, arbitrary conjunctions and finite disjunctions of continuous predicates are continuous again.

Now one can show the following more general fixpoint theorem (see e.g. [42]) together with an important proof principle:

Theorem 3.7 (Fixpoint Induction and Fixpoint Theorem by Kleene) Let $(A, \leq)$ be a cpo and $f : A \to A$ a monotonic function.

1. f has a least fixpoint μf.
2. If P is a continuous predicate on A then the following inference rule is valid:

$$\frac{P(\bot) \qquad P(y) \Rightarrow P(f(y))}{P(\mu f)}$$

   Moreover, $K_f \stackrel{\text{def}}{=} \{f^i(\bot) : i \in \mathbb{N}\}$ is a chain and $P(K_f)$ holds.
3. If f is continuous, then $\mu_f \equiv \mathsf{lub}\, K_f$.

Property 3. is due to [34]. Examples will again be found in Section 4.1.

Using fixpoint induction on the continuous predicate $P(y) \stackrel{\text{def}}{=} y \leq x$ one sees immediately that the inference rule

$$(*) \quad \frac{f(x) \leq x}{\mu f \leq x}$$

remains valid.

We have the following fusion rules for least fixpoints. Consider monotonic functions $f, g, h : A \to A$, where $(A, \leq)$ is a cpo. Function composition is denoted by $\circ$, ie.

$$(f \circ g)(x) \;=\; f(g(x)) \;.$$

Moreover, we use the pointwise lifting of the order $\leq$ to functions:

$$f \leq g \;\Leftrightarrow\; \forall\, x \in A : f(x) \leq g(x) \;.$$

Then the following inference rules are valid:

$$\text{(Right Subfusion)} \quad \frac{f \circ g \leq g \circ h}{\mu f \leq g(\mu h)}$$

$$\text{(Left Subfusion)} \quad \frac{f \text{ continuous and strict} \qquad f \circ g \leq h \circ f}{f(\mu g) \leq \mu h}$$

$$\text{(Fusion)} \quad \frac{f \text{ continuous and strict} \qquad f \circ g = h \circ f}{f(\mu g) = \mu h}$$

The right subfusion rule is immediate from $(*)$. Contrarily, the proof of the left subfusion rule needs fixpoint induction. For the case of $(A, \leq)$ being a complete lattice, a different derivation using the theory of Galois connections (see Section 3.3) is given in [44]; then f has to be universally disjunctive. The fusion rule, finally, is immediate from the two subfusion rules.

Further rules about fixpoints can be found in [44].

Consider now again an arbitrary order $(A, \leq)$. A function $f : A \to A$ is called a **closure operator** iff it is monotonic, extensive and idempotent, the latter two properties meaning $x \leq f(x) \wedge f(f(x)) = f(x)$. Dually, f is called a **kernel operator** iff it is monotonic, contracting and idempotent, where contracting means $f(x) \leq x$. By idempotence, the range of a closure or kernel operator coincides with the set of its fixpoints.

3.3 Galois Connections

Galois connections provide a very concise way of defining certain operations in terms of others. Moreover, a number of useful calculational laws can be proved once and for all for such operations.

3.3.1 Definition and Basic Properties

Consider two pre-orders $(A, \leq_A)$ and $(B, \leq_B)$ and total functions $F : A \to B$ and $G : B \to A$. Then the pair (F, G) is called a **Galois connection (GC)** between A and B iff

$$\forall\, x \in A : \forall\, y \in B : F(x) \leq_B y \;\Leftrightarrow\; x \leq_A G(y)\ .$$

Then F is called the **lower**, G the **upper adjoint** of the GC.

This definition was given by J. Schmidt in 1953. From this it is immediate that if (F, G) is a GC between $(A, \leq_A)$ and $(B, \leq_B)$, then (G, F) is a GC between $(B, \leq_B^{\smile})$ and $(A, \leq_A^{\smile})$. By this observation we frequently only need to reason about one of the adjoints.

As a special case we may choose as $\leq_A$ and $\leq_B$ the identity relations on A and B. Then (F, G) form a GC iff they are inverses of each other.

In the sequel we shall suppress the indices of the pre-orders involved in a Galois connection.

3.3.2 Examples

Many of the examples in this section are taken directly or generalised from [4].

Floor and Ceiling. Consider the functions $\lceil\_\rceil, \lfloor\_\rfloor : \mathbb{R} \to \mathbb{Z}$ and the injection mapping $\iota : \mathbb{Z} \to \mathbb{R}$. We have

$$\begin{aligned}\forall\, x \in \mathbb{R} : \forall\, n \in \mathbb{Z} :&\\ \lceil x \rceil \leq n \;&\Leftrightarrow\; x \leq n\ ,\\ n \leq \lfloor x \rfloor \;&\Leftrightarrow\; n \leq x\ .\end{aligned}$$

Hence $(\lceil\_\rceil, \iota)$ is a GC between $(\mathbb{R}, \leq)$ and $(\mathbb{Z}, \leq)$, whereas $(\lfloor\_\rfloor, \iota)$ is a GC between $(\mathbb{R}, \geq)$ and $(\mathbb{Z}, \geq)$.

Bounds, Suprema and Infima, Maxima and Minima. Consider an arbitrary pre-order $(M, \leq)$. The sets of upper and lower bounds of subsets $X, Y \subseteq M$ are Galois-connected by

$$\mathsf{upb}\, X \supseteq Y \;\Leftrightarrow\; X \subseteq \mathsf{lwb}\, Y\ .$$

So $(\mathsf{upb}, \mathsf{lwb})$ is a GC between $(\mathcal{P}(M), \supseteq)$ and $(\mathcal{P}(M), \subseteq)$.

Let now $(M, \leq)$ be an upper semilattice. Then

$$x \sqcup y \leq z \;\Leftrightarrow\; x \leq z \wedge y \leq z \;\Leftrightarrow\; (x, y) \leq (z, z)\ ,$$

where $\leq$ on the right-hand side denotes the componentwise extension of the ordering to pairs:

$$(x_1, x_2) \leq (y_1, y_2) \;\Leftrightarrow\; x_1 \leq y_1 \wedge x_2 \leq y_2\ .$$

So if we define the doubling function $\Delta : M \to M \times M$ by $\Delta(x) \stackrel{\text{def}}{=} (x, x)$ we find that $(\sqcup, \Delta)$ is a GC between $(M \times M, \leq)$ and $(M, \leq)$. An analogous remark holds for the infimum operation in a lower semilattice.

Let now $(M, \leq)$ be linearly ordered. Then $(M, \leq)$ is a lattice with $x \sqcup y = \max(x, y)$ and $x \sqcap y = \min(x, y)$. So now $(\max, \Delta)$ is a GC between $M \times M$ and M and a similar observation applies to the minimum function.

Boolean Lattices and Predicates. A **Boolean lattice** consists of a distributive lattice $(M, \leq)$ together with a complement operation $\neg : M \to M$ that satisfies the neutrality laws

$$x \sqcup (y \sqcap \neg y) = x \ ,$$
$$x \sqcap (y \sqcup \neg y) = x \ ,$$

and

$$\neg x \leq y \ \Leftrightarrow \ x \geq \neg y \ ,$$
$$\neg x \geq y \ \Leftrightarrow \ x \leq \neg y \ .$$

These latter properties mean that $(\neg \, , \neg \,)$ is required to form a GC between $(M, \leq)$ and $(M, \geq)$ and another one between $(M, \geq)$ and $(M, \leq)$, where $\geq$ is the converse of $\leq$.

In a Boolean lattice one can define an implication operation $\to$ by

$$x \to y \stackrel{\text{def}}{=} \neg x \sqcup y \ .$$

Then one has the further GC

$$x \sqcap y \leq z \ \Leftrightarrow \ x \leq (y \to z)$$

which is an algebraic form of the so-called deduction theorem in classical logic.

A particular Boolean lattice is the space of predicates over a set. Its ordering is the implication ordering

$$p \leq q \ \Leftrightarrow \ p \vee q = q \ \Leftrightarrow \ p \wedge q = p \ .$$

If $p \leq q$ then p is called **weaker** than q while q is called **stronger** than p.

There is a Galois connection between the predicate transformers that yield the weakest liberal precondition and the strongest liberal postcondition, but for reasons of space we do not want to elaborate on that.

3.3.3 More Direct Proofs by Indirect Reasoning

A useful tool for working with GCs are the rules of **indirect inequality**:

$$x \leq y \ \Leftrightarrow \ (\forall \ z : z \leq x \ \Rightarrow \ z \leq y) \ ,$$
$$x \leq y \ \Leftrightarrow \ (\forall \ z : y \leq z \ \Rightarrow \ x \leq z) \ ,$$

The direction $(\Rightarrow)$ needs transitivity of $\leq$, whereas $(\Leftarrow)$ needs reflexivity.

In the case of an antisymmetric and reflexive relation $\leq$ we have the rule of **indirect equality**:

$$x = y \ \Leftrightarrow \ (\forall \ z : z \leq x \ \Leftrightarrow \ z \leq y) \ .$$

As an example of the use of indirect equality we want to derive a simpler representation of the expression $\lfloor \sqrt{\lfloor x \rfloor} \rfloor$ for non-negative x. We calculate

$$\begin{array}{ll} & n \leq \lfloor \sqrt{\lfloor x \rfloor} \rfloor \\ \Leftrightarrow & \quad \{\!\![\text{ GC }]\!\!\} \\ & n \leq \sqrt{\lfloor x \rfloor} \end{array}$$

$$
\begin{array}{ll}
\Leftrightarrow & \{\![\text{ monotonicity }]\!\} \\
& n^2 \leq \lfloor x \rfloor \\
\Leftrightarrow & \{\![\text{ GC, since also } n^2 \in \mathbb{Z} \text{ if } n \in \mathbb{Z} \ ]\!\} \\
& n^2 \leq x \\
\Leftrightarrow & \{\![\text{ monotonicity }]\!\} \\
& n \leq \sqrt{x} \\
\Leftrightarrow & \{\![\text{ GC }]\!\} \\
& n \leq \lfloor \sqrt{x} \rfloor \ .
\end{array}
$$

Hence, by indirect equality, $\lfloor \sqrt{\lfloor x \rfloor} \rfloor = \lfloor \sqrt{x} \rfloor$.

3.3.4 Properties of Galois Connections

We first note that the functions in a GC are quasi-inverses of each other:

Lemma 3.8 (Quasi-Inverses (QI)) *Assume that $F : A \to B$ and $G : B \to A$ form a GC between A and B. Then*

$$
\begin{array}{l}
\forall\ x \in A : x \leq G(F(x)) \ , \\
\forall\ y \in B : F(G(y)) \leq y \ .
\end{array}
$$

Proof: By the GC,

$$x \leq G(F(x)) \ \Leftrightarrow \ F(x) \leq F(x) \ ,$$

which holds by reflexivity of $\leq$. □

For an example consider a Boolean lattice $(M, \leq)$. Then QI for the Galois connection $(\neg\ , \neg\ )$ between $(M, \leq)$ and $(M, \geq)$ means

$$\neg\neg x \leq x \ ,$$

ie. one half of the involution property. The other half follows by QI for the Galois connection $(\neg\ , \neg\ )$ between $(M, \geq)$ and $(M, \leq)$. Together we obtain the **involution** property

$$\neg\neg x \ = \ x \ .$$

From QI we obtain

Corollary 3.9 (Monotonicity) *The adjoints of a GC are monotonic.*

Proof:

$$
\begin{array}{ll}
& x \leq z \\
\Rightarrow & \{\![\text{ by transitivity, since by QI } z \leq G(F(z)) \ ]\!\} \\
& x \leq G(F(z)) \\
\Leftrightarrow & \{\![\text{ GC }]\!\} \\
& F(x) \leq F(z) \ .
\end{array}
$$

□

On the other hand, monotonicity and QI imply that we have a GC:

Lemma 3.10 (O. Ore) *(F, G) form a GC iff F and G are monotonic and quasi-inverses of each other.*

In the literature frequently one of the functions is required to be antitonic rather than monotonic. By passing to the converse of the respective pre-order one obtains a monotonic function again.

Lemma 3.11 *Assume that $F : A \to B$ and $G : B \to A$ form a GC between A and B and let $f \stackrel{\text{def}}{=} F|G(B)$ and $g \stackrel{\text{def}}{=} G|F(A)$ be the restrictions of F and G to the respective image sets. Then*

$$\begin{array}{l} \forall\, x, y \in G(B) : f(x) \leq f(y) \;\Rightarrow\; x \leq y \quad \wedge \\ \forall\, u, v \in F(A) : g(u) \leq g(v) \;\Rightarrow\; u \leq v \;. \end{array}$$

Proof: Assume $x, y \in G(B)$, say $x = G(u), y = G(v)$. Then

$$\begin{array}{ll} & f(x) \leq f(y) \\ \Leftrightarrow & \quad \{\!\!\{ \text{ definition } \}\!\!\} \\ & F(G(u)) \leq F(G(v)) \\ \Leftrightarrow & \quad \{\!\!\{ \text{ GC } \}\!\!\} \\ & G(u) \leq G(F(G(v))) \\ \Rightarrow & \quad \{\!\!\{ \text{ since } F(G(v)) \leq v \text{ by QI and } G \text{ is monotonic } \}\!\!\} \\ & G(u) \leq G(v) \;. \end{array}$$

□

Corollary 3.12 *If both pre-orders are orders then f and g are injective, and hence $F(A)$ and $G(B)$ are order-isomorphic. Moreover, in this case we have*

$$\begin{array}{l} F \circ G \circ F \;=\; F \;, \\ G \circ F \circ G \;=\; G \;. \end{array}$$

Proof: We only need to show the second property. From QI we know $x \leq G(F(x))$. By monotonicity we obtain $F(x) \leq F(G(F(x)))$. On the other hand, QI gives us $F(G(F(x))) \leq F(x)$, so that the claim follows by antisymmetry. □

Corollary 3.13 *Under the above assumptions $F \circ G$ and $G \circ F$ are idempotent and hence a closure and a kernel operator, respectively. Moreover, one has*

$$\begin{array}{l} G(F(x)) \leq x \;\Leftrightarrow\; x \in G(B) \;, \\ y \leq F(G(x)) \;\Leftrightarrow\; y \in F(A) \;. \end{array}$$

From now on we assume orders rather than just pre-orders. First, one can show a uniqueness property:

Theorem 3.14 (Determination) Let $(F_i, G_i) \;\; (i = 1, 2)$ be GCs between A and B. Then

$$F_1 = F_2 \;\Leftrightarrow\; G_1 = G_2 \;,$$

ie. each adjoint determines the other one uniquely.

We now give characterisations of suprema and infima by formulas similar to the one for indirect inequality and equality. Assume that $(M, \leq)$ is ordered and that for $X, Y \subseteq M$ the supremum $\mathsf{lub}\, X$ and the infimum $\mathsf{glb}\, Y$ exist. Then

$$\begin{array}{l} \mathsf{lub}\, X \leq y \;\Leftrightarrow\; (\forall\, x \in X : x \leq y) \;\Leftrightarrow\; X \subseteq \mathsf{lwb}\, y \;, \\ z \leq \mathsf{glb}\, Y \;\Leftrightarrow\; (\forall\, y \in Y : z \leq y) \;\Leftrightarrow\; Y \subseteq \mathsf{upb}\, z \;. \end{array}$$

Lemma 3.15 *Let (F, G) form a GC. Then*

1. *F preserves all existing suprema.*
2. *G preserves all existing infima.*

Proof: We only show 1. Assume that $X \subseteq A$ is such that $\mathsf{lub}\, X$ exists.

$$\begin{array}{ll} & F(\mathsf{lub}\, X) \leq y \\ \Leftrightarrow & \quad \{\![\text{ GC }]\!\} \\ & \mathsf{lub}\, X \leq G(y) \\ \Leftrightarrow & \quad \{\![\text{ characterisation of suprema }]\!\} \\ & \forall\, x \in X : x \leq G(y) \\ \Leftrightarrow & \quad \{\![\text{ GC }]\!\} \\ & \forall\, x \in X : F(x) \leq y \\ \Leftrightarrow & \quad \{\![\text{ definition of the image set }]\!\} \\ & \forall\, z \in F(X) : z \leq y \\ \Leftrightarrow & \quad \{\![\text{ characterisation of suprema }]\!\} \\ & \mathsf{lub}\, F(X) \leq y \end{array}$$

Now we obtain $F(\mathsf{lub}\, X) = \mathsf{lub}\, F(X)$ by indirect equality. □

For an example consider a Boolean lattice $(M, \leq)$. Since $(\neg, \neg)$ is required to form a GC between $(M, \leq)$ and $(M, \geq)$, and the infimum in $(M, \leq)$ is the supremum in $(M, \geq)$ and vice versa, preservation of infima/suprema here amounts to de Morgan's laws

$$\neg(x \sqcup y) \;=\; \neg x \sqcap \neg y \quad \wedge \quad \neg(x \sqcap y) \;=\; \neg x \sqcup \neg y \;.$$

Another example is provided by the upb and lwb functions: we have

$$\mathsf{lwb} \bigcup_{X \in \mathcal{X}} \;=\; \bigcap_{X \in \mathcal{X}} \mathsf{lwb}\, X \quad \wedge \quad \mathsf{upb} \bigcup_{X \in \mathcal{X}} \;=\; \bigcap_{X \in \mathcal{X}} \mathsf{upb}\, X \;.$$

Corollary 3.16 *Let $(A \leq)$ and $(B, \leq)$ be complete lattices.*

1. *$F : A \to B$ has an upper adjoint iff F preserves all suprema.*
2. *$G : B \to A$ has a lower adjoint iff G preserves all infima.*

4 Finite Repetition: Kleene Algebras

4.1 Kleene Algebras and Closures

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.

A **Kleene algebra** (see [17]) is a tuple $(S, \Sigma, \cdot, 0, 1)$ consisting of a set S and operations $\Sigma : \mathcal{P}(S) \to S$ and $\cdot : S \times S \to S$ as well as elements $0, 1 \in S$ such that $(S, \cdot, 1)$ is a monoid and

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

where in this latter equation $\cdot$ is the pointwise extension of the monoid operation. Note that this implies that 0 is absorbing w.r.t. $\cdot$ or, in other words, that $\cdot$ is strict w.r.t. 0:

$$0 \cdot x \ = \ 0 \ = \ x \cdot 0 \ . \tag{15}$$

The binary version of Σ is

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

which makes $(S, +, 0)$ a commutative monoid. A bounded sum of powers is useful, too:

$$x^{\leq n} \stackrel{\text{def}}{=} \sum_{j \leq n} x^j \ .$$

In connection with graph algorithms one often considers the related structure of a **closed semiring** (see e.g. [1]). It differs from a Kleene algebra in that ΣK is only required to exist for *countable* K; moreover, idempotence of $+$ is not postulated. So every Kleene algebra is a closed semiring, but not vice versa.

A straightforward way of obtaining a Kleene algebra is to start from a monoid $(M, \cdot, 1)$, to let $S \stackrel{\text{def}}{=} \mathcal{P}(M)$, $\Sigma \stackrel{\text{def}}{=} \bigcup$, $0 \stackrel{\text{def}}{=} \emptyset$ and to use the pointwise extension of $\cdot$ 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

$$\begin{array}{l} REL \stackrel{\text{def}}{=} (\mathcal{P}(A \bullet A),\ \bigcup,\ ;,\ \emptyset,\ I_A) \ , \\ PAT \stackrel{\text{def}}{=} (\mathcal{P}(A^*),\ \bigcup,\ \bowtie,\ \emptyset,\ \varepsilon \cup A), \\ PAT^+ \stackrel{\text{def}}{=} (\mathcal{P}(A^+),\ \bigcup,\ \bowtie,\ A) \end{array}$$

form Kleene algebras. Of course, PAT^+ is a subalgebra of PAT.

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

$$x \leq y \ \Leftrightarrow \ x + y \ = \ y \ .$$

In our examples $\leq$ coincides with $\subseteq$. The pair $(S, \leq)$ forms a complete lattice with $\mathsf{lub}\,T \ = \ \Sigma\,T$. Moreover, $\cdot$ is universally disjunctive and hence continuous with respect

to $\leq$. Therefore, recursive definitions using $\cdot$ and $+$ can be given least fixpoint semantics using the Knaster-Tarski Fixpoint Theorem 3.3. In the case of LAN, systems of mutually recursive equations for languages give the power of context-free grammars (see e.g. [23]).

We use recursion to define the operators of **closure** $.^*$ and **proper closure** $.^+$ of an element $x \in S$ by

$$x^* \stackrel{\text{def}}{=} \mu\, y\,.\, (1 + y \cdot x) \ , \qquad x^+ \stackrel{\text{def}}{=} \mu\, y\,.\, (x + y \cdot x) \ , \tag{16}$$

where μ is the least fixpoint operator. Using continuity we can also represent the closures by Kleene's approximation sequence from Theorem 3.7.3 as

$$x^* \ = \ \Sigma\{x^j : j \in \mathbb{N}\} \ , \qquad x^+ \ = \ \Sigma\{x^j : j \in \mathbb{N}\backslash 0\} \ , \tag{17}$$

where the x^j are the powers of x in the monoid $(S, \cdot, 1)$.

Remember the operations of iterated join and composition discussed in Section 2.3.2. Here we have embedded these operations into a more general algebraic setting. Note that both $\_^*$ and $\_^+$ are indeed closure operations in the sense of the previous section.

For the star operation one has a rich supply of laws. We list only the most important ones:

$$\begin{aligned}
(x \cdot y)^* \cdot x &= x \cdot (y \cdot x)^* \ , && (18)\\
(x + y)^* &= x^* \cdot (y \cdot x^*)^* \ , && (19)\\
(x + y)^* &= x^* + x^* \cdot y \cdot (x + y)^* \ . && (20)\\
0^* &= 1 \ , && (21)\\
(x^*)^* &= x^* \ , && (22)\\
x \cdot x^* &= x^* \cdot x \ . && (23)
\end{aligned}$$

(21) is immediate from the fixpoint definition of the star operation. (18), (19) and (22) are shown by fixpoint induction. (20) follows from (19), the fixpoint property of star and distributivity. (23) results from (18) by choosing $y = 1$.

4.2 Application to Paths and Transition Systems

For our particular Kleene algebras LAN, REL and PAT we denote the closure operations by $.^*$, $.^*$ and $.^{\rightsquigarrow}$ 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, ie. the graph with node set A and arcs between the nodes corresponding to the pairs in R. By (3) we have, in REL,

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

(Recall that, by our general definition for monoids, $R^i = \underbrace{R \,;\, \cdots \,;\, R}_{i}$ Hence,

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

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** $R^{\rightsquigarrow}$ of R in PAT consists of all finite paths in G (with $A \subseteq R^{\rightsquigarrow}$ 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), $a \bowtie R^{\rightsquigarrow} \bowtie b$ 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.

For the closure $.^*$ in LAN one has

$$\textbf{(Arden's Rule [3])}\ \frac{U \cap \varepsilon = \emptyset \qquad X = V \cup U \bullet X}{X = U^* \bullet V\ .} \tag{24}$$

A related fact can be shown for the iterated join:

Lemma 4.1 *$R^{\rightsquigarrow}$ is the* unique *solution of the recursion equations*

$$\begin{aligned} R^{\rightsquigarrow} &= \varepsilon \cup A \cup R \bowtie R^{\rightsquigarrow}\ , \\ R^{\rightsquigarrow} &= \varepsilon \cup A \cup R^{\rightsquigarrow} \bowtie R\ . \end{aligned}$$

More generally, we have the following proof rules which are analogous to Arden's rule:

$$\frac{X = V \cup R \bowtie X}{X = R^{\rightsquigarrow} \bowtie V} \qquad \frac{X = V \cup X \bowtie R}{X = V \bowtie R^{\rightsquigarrow}} \tag{25}$$

for $X, V \subseteq A^*$ and $R \subseteq A \bullet A$. They result from the above lemma by the fixpoint fusion rule. Again they frequently allow avoiding fixpoint induction.

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 $\leq^{\rightsquigarrow}$ is the language of all $\leq$-non-decreasing sequences. If $\leq$ is even a linear order, then $\leq^{\rightsquigarrow}$ is the language of all sequences which are sorted with respect to $\leq$. This is exploited in [48] 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

$$\phi(u) \stackrel{\text{def}}{=} \mathsf{dom}\, u \bullet \mathsf{cod}\, u \tag{26}$$

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),

$$\phi(S \bowtie R \bowtie T) = S \bullet T \cap R\ . \tag{27}$$

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

$$\phi(T) = I_T\ , \tag{28}$$

$$\phi(U \bowtie V) = \phi(U)\,;\phi(V)\ , \tag{29}$$

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

$$\phi(R^{\rightsquigarrow}) = R^* \ , \qquad \phi(R^{\Rightarrow}) = R^+ \ . \tag{30}$$

For $T \subseteq A$ and $U \subseteq A^+$ we calculate, using neutrality (7) and (28, 29),

$$T \,;\, \phi(U) = T \,;\, I_T \,;\, \phi(U) = T \,;\, \phi(T \bowtie U) \ . \tag{31}$$

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:

$$\begin{array}{ll} & T \bowtie (R \cup S)^{\rightsquigarrow} \\ = & \quad \{\![\text{ by (20) and distributivity }]\!\} \\ & T \bowtie R^{\rightsquigarrow} \cup T \bowtie R^{\rightsquigarrow} \bowtie S \bowtie (R \cup S)^{\rightsquigarrow} \ . \end{array}$$

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 $T \bowtie R^{\rightsquigarrow} \bowtie S = \emptyset$. 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

$$\begin{array}{ll} & T \bowtie R^{\rightsquigarrow} \bowtie S = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ since } \phi \text{ is a total mapping }]\!\} \\ & \phi(T \bowtie R^{\rightsquigarrow} \bowtie S) = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ by (28, 29, 30) and totality of } \phi \ ]\!\} \\ & I_T \,;\, R^* \,;\, S = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ by (6) }]\!\} \\ & \mathsf{cod}\,(I_T \,;\, R^*) \cap \mathsf{dom}\, S = \emptyset \\ \Leftrightarrow & \quad \{\![\text{ by (2) and definition of } I_T \ ]\!\} \\ & T \,;\, R^* \cap \mathsf{dom}\, S = \emptyset \ . \end{array}$$

So we have shown

Lemma 4.2 *Assume $R, S \subseteq A \bullet A$ and $T \subseteq A$ such that $T \,;\, R^* \cap \mathsf{dom}\, S = \emptyset$. Then*

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

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

Corollary 4.3 *Assume $R, S \subseteq A \bullet A$ and $T \subseteq A$ such that $T \,;\, R^* \cap \mathsf{dom}\, S = \emptyset$. Then*

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

Proof: We calculate

$$
\begin{array}{ll}
 & T\,;(R \cup S)^* \\
= & \quad \{\![\text{ by (30, 31) }]\!\} \\
 & T\,;\phi(T \bowtie (R \cup S)^{\rightsquigarrow}) \\
= & \quad \{\![\text{ by Lemma 4.2 }]\!\} \\
 & T\,;\phi(T \bowtie R^{\rightsquigarrow}) \\
= & \quad \{\![\text{ by (30, 31) }]\!\} \\
 & T\,;R^* \ .
\end{array}
$$

□

Since the path closure $R^{\rightsquigarrow}$ records all intermediate points of the paths, we need the strong precondition $T\,;R^* \cap \mathsf{dom}\,S = \emptyset$ 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 4.4 *Assume $R \subseteq A \bullet A$ and $T \subseteq A$. Then*

$$T\,;R^* \;\subseteq\; T \cup T\,;R\,;U^* \ ,$$

where $U \stackrel{\text{def}}{=} \neg T \bowtie R$ and $\neg T = A \backslash T$ (cf. Section 2.3.1).

Proof: Fixpoint induction on the continuous predicate

$$P[X] \;\Leftrightarrow\; T\,;X \;\subseteq\; T \cup T\,;R\,;U^* \ .$$

□

Corollary 4.5 *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. □

4.3 A Simple Reachability Algorithm

We now derive a simple algorithm that solves 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

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

The aim is to derive a recursive variant of *reach* from this specification. A termination case is given by $reach(\emptyset) = \emptyset\,;R^* = \emptyset$. Moreover, we can exploit Corollary 4.5:

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

where $U \stackrel{\text{def}}{=} \neg S \bowtie R$ and $\neg S \stackrel{\text{def}}{=} A\backslash S$. 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 again use the technique of generalisation and introduce a second parameter T for the set that restricts R by defining

$$re(S,T) = S\,;(\neg T \bowtie R)^* \ .$$

Then we have the embedding $reach(S) = re(S,\emptyset)$. As the termination case we get again $re(\emptyset,T) = \emptyset$. Moreover, we calculate:

$$\begin{array}{ll} & re(S,T) \\ = & \quad \{\![\text{ definition of } re \ ]\!\} \\ & S\,;(\neg T \bowtie R)^* \\ = & \quad \{\![\text{ by Corollary 4.5 }]\!\} \\ & S \cup S\,;(\neg T \bowtie R)\,;(\neg S \bowtie \neg T \bowtie R)^* \\ = & \quad \{\![\text{ by (8) }]\!\} \\ & S \cup (S \bowtie \neg T)\,;R\,;(\neg S \bowtie \neg T \bowtie R)^* \\ = & \quad \{\![\text{ by (5) and Boolean algebra }]\!\} \\ & S \cup (S\backslash T)\,;R\,;(\neg(S \cup T) \bowtie R)^* \\ = & \quad \{\![\text{ definition of } re \ ]\!\} \\ & S \cup re((S\backslash T)\,;R,\ S \cup T) \ . \end{array}$$

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 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. [62]) one can finally transform this into a tail recursion and from there into loop form.

4.4 A Characterisation of Cycles

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

Characterise when R contains a **cyclic path**, *ie. 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 (26) to calculate

$$\begin{array}{ll} & \phi(cyc(R)) \\ = & \quad \{\![\text{ by definition of } cyc \text{ and continuity of } \phi \ ]\!\} \\ & \bigcup_{a \in A} \phi(a \bowtie R^{\Rightarrow} \bowtie a) \\ = & \quad \{\![\text{ by (27, 30) }]\!\} \\ & \bigcup_{a \in A} a \bullet a \cap R^{+} \\ = & \quad \{\![\text{ distributivity }]\!\} \\ & (\bigcup_{a \in A} a \bullet a) \cap R^{+} \\ = & \quad \{\![\text{ definition of } I \ ]\!\} \\ & I_A \cap R^{+} \ . \end{array}$$

Since ϕ is a total mapping, we infer

Lemma 4.6 $cyc(R) \neq \emptyset \ \Leftrightarrow \ R^{+} \cap I_A \neq \emptyset$.

4.5 Layer-Oriented Graph Traversal

A number of graph problems use the set of all paths starting in a set S and ending in a set T of nodes. For some of these problems we give in this section a general specification, derive a basic algorithm and apply it to examples.

4.5.1 Specification

Consider a graph over node set A. We specify a general graph processing operation F by

$$F(f, g)(S, R, T) \stackrel{\text{def}}{=} g(f(S \bowtie R^{\rightsquigarrow} \bowtie T)) \ ,$$

where

- $f : A^{*} \to M$ is an *abstraction* function from paths to a "valuation" set M which might e.g. be the set of natural numbers if we are interested in counting edges in a path. f is extended pointwise to sets of words and hence is distributive, monotonic and strict.
- $g : \mathcal{P}(M) \to \mathcal{P}(M)$ is a *selection* function. It acts globally on $f(W)$, eg. selects the minimum in the case of sets of natural numbers, and hence *is not* defined as a pointwise extension. Rather we assume the properties

  (GEN1) $g(K) \subseteq K$,
  (GEN2) $g(K \cup L) = g(g(K) \cup g(L))$ (weak distributivity),

  for $K, L \subseteq M$.

Weak distributivity means that g may be applied to subsets without changing the overall result. Note that (GEN2) implies idempotence of g and (GEN2) is equivalent to

$$\text{(GEN2')} \quad g(K \cup L) = g(K \cup g(L)).$$

The difference in algebraic requirements for f and g explains why we use two separate functions rather than their combination into a single one.

We now abstract from the particular case of graphs. Let $(S, \Sigma, \cdot, 0, 1)$ be a Kleene algebra. We define the general operation F by

$$F(f,g)(a,b,c) \stackrel{\text{def}}{=} g(f(a \cdot b^* \cdot c)) \ ,$$

where $a, b, c \in S$ and $f : S \to \mathcal{P}(M)$ now is a disjunctive function:

$$f(x+y) \ = \ f(x) \cup f(y) \ .$$

The requirements on g are as before.

4.5.2 Derivation of the Basic Algorithm

We now want to find a recursion equation for F. We calculate

$$\begin{array}{ll} & F(f,g)(a,b,c) \\ = & \quad \{\![\text{ definition }]\!\} \\ & g(f(a \cdot b^* \cdot c)) \\ = & \quad \{\![\text{ idempotence of } + \ ]\!\} \\ & g(f(a \cdot b^* \cdot c + a \cdot b^* \cdot c)) \\ = & \quad \{\![\text{ recursion for } \_^* \ ]\!\} \\ & g(f(a \cdot (1 + b \cdot b^*) \cdot c + a \cdot b^* \cdot c))) \\ = & \quad \{\![\text{ distributivity and neutrality }]\!\} \\ & g(f(a \cdot c + a \cdot b \cdot b^* \cdot c + a \cdot b^* \cdot c)) \\ = & \quad \{\![\text{ commutativity and distributivity of } + \ ]\!\} \\ & g(f(a \cdot c + (a + a \cdot b) \cdot b^* \cdot c)) \\ = & \quad \{\![\text{ disjunctivity of } f \ ]\!\} \\ & g(f(a \cdot c) \ \cup \ f((a + a \cdot b) \cdot b^* \cdot c)) \\ = & \quad \{\![\text{ (GEN2') }]\!\} \\ & g(f(a \cdot c) \ \cup \ g(f((a + a \cdot b) \cdot b^* \cdot c))) \\ = & \quad \{\![\text{ definition }]\!\} \\ & g(f(a \cdot c) \ \cup \ F(f,g)(a + a \cdot b,\ b,\ c)) \ . \end{array}$$

4.5.3 Termination Cases

We prepare the introduction of termination cases by deriving another form of F:

$$\begin{array}{ll} & F(f,g)(a,b,c) \\ = & \quad \{\![\text{ definition }]\!\} \\ & g(f(a \cdot b^* \cdot c)) \end{array}$$

$$
\begin{aligned}
&= \quad \{\![\text{ the recursion for } \_^* \text{ implies } b^* = 1 + b^* \ ]\!\} \\
&\quad g(f(a \cdot (1 + b^*) \cdot c)) \\
&= \quad \{\![\text{ distributivity and neutrality }]\!\} \\
&\quad g(f(a \cdot c + a \cdot b^* \cdot c)) \\
&= \quad \{\![\text{ disjunctivity of } f \ ]\!\} \\
&\quad g(f(a \cdot c) \cup f(a \cdot b^* \cdot c)) \ . \qquad (32)
\end{aligned}
$$

Motivated by the graph theoretical applications we now postulate some conditions about f and g. They involve the domain and codomain operations which can be defined in Kleene algebras under mild additional conditions (see [15]) that hold, in particular, in LAN, REL and PAT. We require

$$
\begin{array}{lrcl}
\text{(LAY1)} & \mathsf{dom}\, c \ \leq \ \mathsf{cod}\, a & \Rightarrow & g(f(a \cdot c) \cup f(a \cdot u \cdot c)) = g(f(a \cdot c)) \ , \\
\text{(LAY2)} & \mathsf{cod}\,(a \cdot u) \ \leq \ \mathsf{cod}\, a & \Rightarrow & g(f(a \cdot c) \cup f(a \cdot u \cdot c)) = g(f(a \cdot c)) \ ,
\end{array}
$$

with $a, c, u \in S$. When dealing with graph problems, $\mathsf{dom}\, c \leq \mathsf{cod}\, a$ is the case where the set of starting nodes of c already is contained in the set of end nodes of a. The condition $\mathsf{cod}\,(a \cdot u) \leq \mathsf{cod}\, a$ means that by further traversal of the graph along u no new end nodes are reached. In both cases the second use of the abstraction f should give no new information and be ignored by g.

Case 1: $\mathsf{dom}\, c \leq \mathsf{cod}\, a$. Then we get by (LAY1) that

$$(32) \ = \ g(f(a \cdot c)) \ .$$

For the second case we use the following lemma that is easily shown by fixpoint induction:

Lemma 4.7 *For $a, b \in S$ one has* $\mathsf{cod}\,(a \cdot b) \ \leq \ \mathsf{cod}\, a \ \Rightarrow \ \mathsf{cod}\,(a \cdot b^*) \ \leq \ \mathsf{cod}\, a$.

Now we can proceed with

Case 2: $\mathsf{cod}\,(a \cdot b) \leq \mathsf{cod}\, a$. Then by Lemma 4.7 and (LAY2) we get again

$$(32) \ = \ g(f(a \cdot c)) \ .$$

In sum we have derived the following basic algorithm:

$$
\begin{aligned}
F(f,g)(a,b,c) = \ & \mathsf{if} \ \mathsf{dom}\, c \ \leq \ \mathsf{cod}\, a \ \vee \ \mathsf{cod}\,(a \cdot b) \ \leq \ \mathsf{cod}\, a \\
& \quad \mathsf{then}\, g(f(a \cdot c)) \\
& \quad \mathsf{else} \ g(f(a \cdot c) \ \cup \ F(f,g)(a + a \cdot b,\, b,\, c)) \ \mathsf{fi} \ .
\end{aligned}
$$

This terminates whenever the image set of cod in the underlying Kleene algebra is upward Noetherian, ie. has no infinite $\leq$-ascending chains. This is always the case in LAN, whereas in REL and PAT it holds iff A is finite. Then $\mathsf{cod}\, a$ is a suitable termination measure, since

$$\mathsf{cod}\,(a + a \cdot b) \ = \ \mathsf{cod}\, a + \mathsf{cod}\,(a \cdot b) \ \geq \ \mathsf{cod}\, a$$

and

$$\mathsf{cod}\,(a + a \cdot b) \ = \ \mathsf{cod}\, a \ \Leftrightarrow \ \mathsf{cod}\,(a \cdot b) \ \leq \ \mathsf{cod}\, a \ .$$

In [15] we derive from this basic algorithm a more efficient one that avoids the generally expensive computation of $a + a \cdot b$. In the graph-theoretic interpretation this algorithm separately maintains the set of already visited nodes and the set of nodes the successors of which may not have been visited yet. The derivation, however, again takes place purely in the general setting of Kleene algebras.

4.5.4 Application: Reachability

Our first application is, once again, a reachability problem. Given an alphabet A of nodes and an edge set $R \subseteq A \bullet A$, we want to compute the set of all nodes that are reachable from a set $S \subseteq A$ of nodes. We first give a solution in the Kleene algebra PAT^+ and choose $f \stackrel{\text{def}}{=} \mathsf{cod}$, $g \stackrel{\text{def}}{=} id$, $c \stackrel{\text{def}}{=} A$ and define, for $S \subseteq A^+$,

$$reach(S) \stackrel{\text{def}}{=} F(\mathsf{cod}, id)(S, R, A) \ .$$

It is easily checked that cod and id satisfy the required conditions.

By our definitions we can now solve the reachability problem recursively:

$$\begin{array}{ll}
 & reach(S) \\
= & \quad \{\![\text{ definition of } reach \ ]\!\} \\
 & F(\mathsf{cod}, id)(S, R, A) \\
= & \quad \{\![\text{ algorithm for } F \ ]\!\} \\
 & \mathsf{if}\ \mathsf{dom}\, A \subseteq \mathsf{cod}\, S \vee \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
 & \quad \mathsf{then}\ \mathsf{cod}\,(S \bowtie A) \\
 & \quad \mathsf{else}\ \mathsf{cod}\,(S \bowtie A) \cup reach(S \cup S \bowtie R)\ \mathsf{fi} \\
= & \quad \{\![\ (5), \text{ monotonicity of } \mathsf{cod} \ ]\!\} \\
 & \mathsf{if}\ A \subseteq \mathsf{cod}\, S \vee \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
 & \quad \mathsf{then}\ \mathsf{cod}\, S \\
 & \quad \mathsf{else}\ \mathsf{cod}\, S \cup reach(S \cup S \bowtie R)\ \mathsf{fi} \\
= & \quad \{\![\ A \subseteq \mathsf{cod}\, S \Rightarrow A = \mathsf{cod}\, S \Rightarrow \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \ ]\!\} \\
 & \mathsf{if}\ \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
 & \quad \mathsf{then}\ \mathsf{cod}\, S \\
 & \quad \mathsf{else}\ \mathsf{cod}\, S \cup reach(S \cup S \bowtie R)\ \mathsf{fi} \ .
\end{array}$$

Alternatively, we can solve the reachability problem in REL by setting, for $Q \subseteq A$

$$relreach(Q) \stackrel{\text{def}}{=} F(\mathsf{cod}, id)(I_Q, R, I_A) \ .$$

Again the required conditions are easily shown. The resulting algorithm is

$$\begin{array}{l}
relreach(Q) \ = \ rr(I_Q) \ , \\
rr(S) \ = \\
\quad \mathsf{if}\ \mathsf{cod}\,(S\,;R) \subseteq \mathsf{cod}\, S \\
\quad\quad \mathsf{then}\ \mathsf{cod}\, S \\
\quad\quad \mathsf{else}\ \mathsf{cod}\, S \cup rr(S \cup S\,;R)\ \mathsf{fi} \ .
\end{array}$$

4.5.5 Application: Shortest Connecting Path

Now we want to calculate the set of all shortest paths between two sets $S, T \subseteq A$, where we only count the number of edges (the simple relational model does not account for edge weights in an easy way). We define, in PAT^+,

$$shortestpaths(S, T) \stackrel{\text{def}}{=} F(id, minpaths)(S, R, T) \ ,$$

with

$$minpaths(U) \stackrel{\text{def}}{=} \mathsf{let}\ ml\ =\ min(\|U\|)\ \mathsf{in}\ \{s \in U : \|s\| = n\}\ .$$

Here we use the pointwise extension of $\|\_\|$ to languages. Hence *minpaths* selects from a set of words the ones with the least number of letters. Again the conditions (GEN) and (LAY1,LAY2) are satisfied. Therefore we have the following algorithm for computing the shortest path between a set S and the node y:

$$\begin{array}{ll}
& shortestpaths(S, y) \\
= & \quad \{\![\text{ definition }]\!\} \\
& F(id, minpaths)(S, R, y) \\
= & \quad \{\![\text{ recursion for } F \ ]\!\} \\
& \mathsf{if}\ \mathsf{dom}\, y \subseteq \mathsf{cod}\, S \vee \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
& \quad \mathsf{then}\ minpaths(S \bowtie y) \\
& \quad \mathsf{else}\ \ minpaths(S \bowtie y \cup shortestpaths(S \cup S \bowtie R, y))\ \mathsf{fi} \\
= & \quad \{\![\text{ set theory, logic }]\!\} \\
& \mathsf{if}\ y \in \mathsf{cod}\, S \\
& \quad \mathsf{then}\ minpaths(S \bowtie y) \\
& \quad \mathsf{else}\ \mathsf{if}\ \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
& \qquad \mathsf{then}\ minpaths(S \bowtie y) \\
& \qquad \mathsf{else}\ \ minpaths(S \bowtie y \cup shortestpaths(S \cup S \bowtie R, y))\ \mathsf{fi}\ \mathsf{fi} \\
= & \quad \{\![\ y \notin \mathsf{cod}\, S \ \Rightarrow\ S \bowtie y = \emptyset\ ]\!\} \\
& \mathsf{if}\ y \in \mathsf{cod}\, S \\
& \quad \mathsf{then}\ minpaths(S \bowtie y) \\
& \quad \mathsf{else}\ \mathsf{if}\ \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
& \qquad \mathsf{then}\ minpaths(\emptyset) \\
& \qquad \mathsf{else}\ \ minpaths(shortestpaths(S \cup S \bowtie R, y))\ \mathsf{fi}\ \mathsf{fi}\ .
\end{array}$$

By the definition of *minpaths* we have

$$minpaths(\emptyset)\ =\ \emptyset\ .$$

We now simplify the second else-branch.

$$\begin{array}{ll}
& minpaths(shortestpaths(S \cup S \bowtie R, y)) \\
= & \quad \{\![\text{ definition of } shortestpaths \ ]\!\} \\
& minpaths(F(id, minpaths)(S \cup S \bowtie R, R, y)) \\
= & \quad \{\![\text{ definition of } F \ ]\!\} \\
& minpaths(minpaths(id(S \cup S \bowtie R, y))) \\
= & \quad \{\![\text{ idempotence of } minpaths \ ]\!\} \\
& minpaths(id(S \cup S \bowtie R, y)) \\
= & \quad \{\![\text{ definition of } F \ ]\!\} \\
& F(id, minpaths)(S \cup S \bowtie R, R, y) \\
= & \quad \{\![\text{ definition of } shortestpaths \ ]\!\} \\
& shortestpaths(S \cup S \bowtie R, y)\ .
\end{array}$$

Altogether,

$$
\begin{array}{l}
\mathit{shortestpaths}(S, y) \;=\; \\
\quad \mathsf{if}\ y \in \mathsf{cod}\, S \\
\quad\quad \mathsf{then}\ \mathit{minpaths}(S \bowtie y) \\
\quad\quad \mathsf{else}\ \mathsf{if}\ \mathsf{cod}\,(S \bowtie R) \subseteq \mathsf{cod}\, S \\
\quad\quad\quad\quad \mathsf{then}\ \emptyset \\
\quad\quad\quad\quad \mathsf{else}\ \ \mathit{shortestpaths}(S \cup S \bowtie R, y)\ \mathsf{fi}\ \mathsf{fi}\ .
\end{array}
$$

We can simplify the algorithm further using the law $\mathsf{cod}\,(U \bowtie V) = \mathsf{cod}\,((\mathsf{cod}\, U) \bowtie V)$, since this shows that we only need to carry along $\mathsf{cod}\, S$ rather than all of S. Together with the general efficiency improvement in [15] this brings the algorithm down to a complexity of $O(|A| + |R|)$.

4.6 An Algebraic View of Invariants

We have already seen that the nullary relations ε and $\emptyset$ can play the roles of the truth values *true* and *false*, resp. On the other hand, they also act as 1 and 0 of the Kleene algebra LAN. So, more generally, we can use 1 and 0 as representations of *true* and *false* in arbitrary Kleene algebras.

Under certain circumstances one can then take another step and regard the set of subidentities, ie. $\{x : x \leq 1\}$, as representations of predicates and composition with them as restriction. For instance, as we have seen, in PAT for $S \subseteq A \cup \varepsilon$ the join $S \bowtie T$ restricts T to the subset of paths that start with a node in S while $T \bowtie S$ restricts T to the subset of paths that end with a node in S. In REL, for $S \subseteq I_A$, say $S = I_C$, the composition $S\,;T$ yields the domain restriction of T to C while $T\,;S$ yields the range restriction of T to C.

The latter property can now be used to give an algebraic view of invariants and an easy general proof of a rule for strengthening and weakening invariants for recursively defined entities.

Consider a Kleene algebra $(S, \Sigma, \cdot, 0, 1)$ and $a, b \in S$. We call a an **invariant** of b if

$$a \cdot b \;=\; a \cdot b \cdot a\ .$$

Let us interpret this in REL and choose a as a subidentity and view b as a transition relation between program states. Then we may rephrase the above equation as follows: for all states that satisfy assertion a, all successor states again satisfy a. So it is appropriate to call a an invariant.

We now use the fixpoint fusion rule of Section 3.2 to show that an idempotent invariant of b also is an invariant of b^*. Here an element $a \in S$ is **idempotent** if $a \cdot a = a$. Rules for introducing invariants into general recursive definitions are developed in [51].

First, the fusion rule tells us that for $a, b \in S$ we have

$$a \cdot b^* \;=\; \mu g_{a,b}\ , \tag{34}$$

where $g_{a,b}(y) \stackrel{\text{def}}{=} a + y \cdot b$. In particular, $b^* \stackrel{\text{def}}{=} \mu g_{1,b}$.

Next, we show, again by the fusion rule, that

$$a \cdot b^* \;=\; (a \cdot b)^* \cdot a\ . \tag{35}$$

Setting $f(y) \stackrel{\text{def}}{=} y \cdot a$ we obtain

$$(a \cdot b)^* \cdot a \;=\; f(\mu g_{1,a \cdot b})$$

and need to check that

$$f \cdot g_{1,a\cdot b} \;=\; g_{a,b} \cdot f \;.$$

We calculate

$$\begin{array}{ll} & (f \cdot g_{1,a\cdot b})(y) \\ = & \quad \{\![\text{ definitions }]\!\} \\ & (1 + y \cdot a \cdot b) \cdot a \\ = & \quad \{\![\cdot \text{ disjunctive, 1 neutral }]\!\} \\ & a + y \cdot a \cdot b \cdot a \\ = & \quad \{\![\ a \text{ invariant of } b \ ]\!\} \\ & a + y \cdot a \cdot b \\ = & \quad \{\![\text{ definitions }]\!\} \\ & (g_{a,b} \cdot f)(y) \;. \end{array}$$

Now,

$$\begin{array}{ll} & a \cdot b^* \\ = & \quad \{\![\text{ by (35) }]\!\} \\ & (a \cdot b)^* \cdot a \\ = & \quad \{\![\ a \text{ idempotent }]\!\} \\ & (a \cdot b)^* \cdot a \cdot a \\ = & \quad \{\![\text{ by (35) }]\!\} \\ & a \cdot b^* \cdot a \;. \end{array}$$

Let us interpret (35) in the path Kleene algebra PAT. For a set R of edges and a set S of nodes, viewed as singleton paths, it means that if all edges of R that start in S also end in S, then all paths using only edges in R that start in S will only pass through nodes in S and end in S. This is used in [65] for the simplification of another general scheme for layer-oriented graph traversal.

Moreover, (35) gives us a way to strengthen and weaken invariants. Suppose that a_1, a_2 are both idempotent invariants of b that commute under $\cdot$. Then also $a_1 \cdot a_2$ is an invariant of b and hence of b^*. Now, in LAN, REL and PAT subidentities commute under the respective interpretation of $\cdot$, since for them we have $a_1 \cdot a_2 \;=\; a_1 \sqcap a_2$. So the product of two subidentities that represent assertions corresponds to the conjunction of these assertions.

5 Streams

We have already hinted at the connection between transition relations and paths in the transition graph. We shall call finite or infinite paths in such a graph *streams*. Such a stream is characterised by the set of all its finite prefixes. We shall use this observation to apply our algebra of formal languages to the specification and manipulation of streams.

5.1 A Simple Soda Machine

To show the style of our approach and in order to better motivate the technicalities to come we first give a number of examples with informal explanations. The precise definitions will be given in later sections.

We start with the description of a simple soda machine. It accepts half dollars and quarters and emits a can of soda after having received a half dollar's worth in coins. Let h and q denote the events of receiving a half dollar and a quarter, respectively, and c the event of emitting a can of soda. Then the behaviour of that machine is described by the regular-like expression

$$((h \cup q \bullet q) \bullet c)^\omega \ ,$$

where $\_^\omega$ denotes infinite repetition. Each expression of this kind denotes a set of (finite or infinite) streams; in the case of the soda machine all these streams are infinite.

In the above expression, the iterated subexpression $(h \cup q \bullet q) \bullet c$ states the following safety properties: the customer must insert the correct amount of money and is not allowed to insert further money before delivery of the can. The infinite repetition $\_^\omega$ combines safety and liveness aspects: it expresses the correct order of insert/deliver cycles, a safety property, and expresses the temporal aspect of eventuality: it guarantees that after insertion of a sufficient amount of money eventually a can is delivered and the machine is ready to accept further orders.

We prefer to leave states implicit as long as possible, since frequently regular expressions are clearer and more concise than the corresponding descriptions by accepting automata (Büchi automata in the case of infinite repetition, see e.g. [63, 76, 77]).

5.2 Fairness

Other eventuality properties can already be expressed using the finite repetition operators $\_^*$ and $\_^+$. To exemplify this, we describe a scheduler for unboundedly fair merging of input from two channels. It is modeled as an infinite stream over the alphabet $\{0, 1\}$, where 0 denotes choice from the left and 1 choice from the right input channel of the merge module. The fact that at least once there is a choice from the left followed eventually by a choice from the right is expressed by the regular expression $0^+ \bullet 1$. By adding the symmetric requirement and, again infinite repetition to drive the single cycles, we get the following description of the set of streams that model the behaviour of a fair scheduler:

$$\mathcal{SCHED} \stackrel{\text{def}}{=} (0^+ \bullet 1 \cup 1^+ \bullet 0)^\omega \ .$$

The "local eventuality" is here expressed by the finiteness of $\_^+$, whereas the infinite repetition $\_^\omega$ again adds liveness and "global eventuality".

Arbitrary (and hence possibly non-fair) merge would be obtained by replacing this scheduler by $(0 \cup 1)^\omega$.

The reason why fairness does not cause problems in our approach is that fairness constraints are expressed using the star operation which has a simple recursive definition using least fixpoints w.r.t. the inclusion ordering on sets of streams, whereas there are continuity problems w.r.t. extensions of the prefix order to sets of streams. This is due to the fact that the prefix order has operational traits and unbounded fairness is operationally not feasible, whereas the inclusion ordering is purely descriptive and hence does not face this problem. It is adequate for proving properties of sets of streams; when it comes to implementation, of course operationally feasible descendants have to be used.

We prefer to state fairness assumptions explicitly, since this gives much greater flexibility than building them into the underlying semantic framework (such as e.g. in [16]).

5.3 Channels

Another aspect of fairness and eventuality is exhibited in the description of channels as used in many protocol specifications. The channels are faulty, but fair in the sense that after an unbounded but finite number of faulty transmissions they will at least once transmit correctly.

We will describe their behaviour using streams of functions that model individual transmissions. Let *id* be the identity function which models correct transmission, *fail* the function which transforms any message into an "error element", and *skip* the function transforming a message into empty output. Let in the sequel stand the sub/superscript b for $*$ (unbounded but finite repetition) or $\leq k$ for some $k \in \mathbb{N}$ (bounded repetition). Then the following specifications express unbounded and bounded fairness, respectively.

A possibly corrupting but fair channel is described by

$$cchan_b \stackrel{\text{def}}{=} (fail^b \bullet id)^\omega \ ,$$

a possibly lossy but fair channel by

$$lchan_b \stackrel{\text{def}}{=} (skip^b \bullet id)^\omega$$

and a possibly lossy and corrupting but fair channel by

$$lcchan_b \stackrel{\text{def}}{=} ((skip \cup fail)^b \bullet id)^\omega \ .$$

An unfair corrupting channel is

$$arbchan \stackrel{\text{def}}{=} (fail \cup id)^\omega \ .$$

This kind of channel descriptions has been used in [49] for a very concise algebraic correctness proof of the alternating bit protocol.

5.4 Two Stream-Based Models of Systems

5.4.1 Modules as Stream-Processing Functions

A stream-processing function (SPF) is a function from tuples of input streams to tuples of output streams (see e.g. [32, 14]). In the case of synchronous systems this may equivalently be replaced by a function from a stream of input tuples to a stream of output tuples.

In the SPF view each module is described as an SPF. The advantage of this model is that it allows easy definitions of various composition operations for modules and hence lends itself to a modular structuring of large systems.

The disadvantage in the description of asynchronous systems is that the separation between input and output streams loses causal information, viz. which input triggered which output. This gives rise to the (in)famous merge anomaly [11] which can be fixed by re-introducing time information into the streams. It has to be expressed which elements of a stream are considered to belong to the same time interval. This can be done using explicit time ticks [13] or streams of sequences where each sequence lists the elements belonging to one time interval [36].

5.4.2 Trace Models

In the trace view, the overall system is described by admissible seqentialisations of actions during system runs (interleaving semantics). If the structure of non-deterministic branching is preserved, one obtains tree-like semantic domains such as used in CCS [45], CSP [27] and process algebra [5]. In the simplest case, however, the trace structure is a *set* of streams (see also [31]) which we term a **behaviour**. In this view, a stream in A^∞ is a complete record of one possible system run with all system actions interleaved.

Eg. for a CSP-like view one uses the alphabet $A = C \times V$ of basic actions, where C is a set of channel names and V a set of values that are transmitted along the channels. Then the streams in A^∞ are complete records of system runs with all channel activities interleaved.

The advantage of this view is that it keeps track of the causality between input and output; hence the merge anomaly does not arise.

The disadvantage is the lack of immediate modularity, since the overall system is described. However, modularisation can be re-introduced by restricting attention to subsets of channels.

5.5 Streams as Formal Languages

We now make our notion of streams precise. A word u is a **prefix** of a word v, written $u \sqsubseteq v$, iff there is a word w such that $u \bullet w = v$. It is well-known that this defines a partial order on words which is even well-founded. Moreover, ε is the least element in this order. The corresponding strict-order is denoted by $\sqsubset$.

The **prefix closure** of a language $U \subseteq A^*$ is $(\sqsubseteq ; U)$. Then U is **prefix-closed** iff $(\sqsubseteq ; U) \subseteq U$. Note that every non-empty prefix-closed language contains ε.

A few properties we shall use are the following (where $x, y, u, v, w \in A^*$ and $U, V \subseteq A^*$):

$$v \sqsubseteq w \;\Leftrightarrow\; u \bullet v \sqsubseteq u \bullet w \;, \tag{36}$$

$$u \sqsubseteq w \wedge v \sqsubseteq w \;\Rightarrow\; u \sqsubseteq v \vee v \sqsubseteq u \;, \tag{37}$$

$$V \neq \emptyset \;\Rightarrow\; (\sqsubseteq ; (U \bullet V)) \;=\; (\sqsubseteq ; U) \cup U \bullet (\sqsubseteq ; V) \;. \tag{38}$$

Property (37) is also called **local linearity**.

Informally, a stream over A is a finite or infinite sequence of elements of A. The basis of our approach is the observation that such a stream is completely characterised by the set of its finite prefixes. This set is prefix-closed. Moreover, it is directed, since in the partial order $(A^*, \sqsubseteq)$ by local linearity the directed sets can be characterised in another way:

Lemma 5.1 *$D \subseteq A^*$ is directed w.r.t. $\sqsubseteq$ iff D is linearly ordered by $\sqsubseteq$, ie. iff for any two elements $u, v \in D$ we have $u \sqsubseteq v$ or $v \sqsubseteq u$.*

The set of finite prefixes of a stream is a set of words of increasing length "growing at the right end". This set may be finite or infinite. A simple example is, for $a \in A$, the infinite set

$$a^* = \{\varepsilon,\ a,\ a \bullet a,\ a \bullet a \bullet a,\ a \bullet a \bullet a \bullet a,\ \ldots\}$$

corresponding to the infinite constant stream consisting just of as.

We identify now a stream with the set of its finite prefixes. The set of streams is therefore

$$A^\infty \stackrel{\text{def}}{=} \{S \subseteq A^* : S \text{ prefix-closed and directed}\} .$$

We note here that prefix-closed directed sets are also known as **ideals**. So our view of streams corresponds to that used in denotational semantics, where all object of a semantic domain can be viewed as ideals of a partial order (the approximation order) on a set of finite approximations of potentially infinite objects. In fact, as shown in [50, 54], our treatment generalises to arbitrary domains via the so-called ideal completion (see e.g. [18]).

5.6 Behaviours and Trace Refinement

Our application of streams will be the description of systems. To model non-determinacy, we define a **behaviour** to be a set of streams.

It should be noted that using *sets* of streams as behaviours allows only "trace-like" semantics in which there is no distinction between internal and external non-determinacy. The algebraic reflection of this is that concatenation, our sequencing operation, distributes through union both from the left and from the right. In algebraic approaches to CCS-like systems (see e.g. [5]) only one of these distributivities holds. This results in models with tree-like objects that reflect the non-deterministic branching structure in time. This detailed record is lost by admitting both distributivities rather than just one.

As our refinement relation we choose inclusion, ie. behaviour $\mathcal{B}$ **refines** behaviour $\mathcal{C}$ iff $\mathcal{B} \subseteq \mathcal{C}$. To allow correct local refinements one therefore has to ensure monotonicity of all operations w.r.t. inclusion.

Example 5.2 We resume the example from Section 5.2 and show that bounded fairness refines unbounded fairness: since all operators involved are monotonic w.r.t. inclusion, we obtain from $a \bullet a^{\leq k} \subseteq a^+$ for $a \in A$ that

$$(0 \bullet 0^{\leq k} \bullet 1 \cup 1 \bullet 1^{\leq k} \bullet 0)^\omega \subseteq \mathcal{SCHED} .$$

□

5.7 Describing Behaviours by Snapshots

We want to characterise streams by certain sets of "relevant" or "admissible" **snapshots** in the form of finite prefixes. Such a set is called a **property** in this context. Assume a set $U \subseteq A^*$ of admissible snapshots. If a stream contains a subset $D \subseteq U$ of snapshots then D has to be directed. However, there may be arbitrary "gaps" between the snapshots in D. To reconstruct the stream we therefore have to "fill in the details" between the snapshots. This is done by taking the prefix closure $(\sqsubseteq ; D)$. Hence we define the set of streams, ie. the behaviour, "spanned" by snapshot set U as

$$\mathsf{str}\, U \stackrel{\text{def}}{=} \{(\sqsubseteq ; D) : D \subseteq U \text{ and } D \text{ directed}\} .$$

This is the set of streams that "interpolate" consistent snapshots in U. A related notion occurs in [19]. Note that $\mathsf{str}\, A^* = A^\infty$ and str is monotonic w.r.t. inclusion. A different characterisation of str is given by

Lemma 5.3 *For $S \in A^\infty$ and $Q \subseteq A^*$ the following statements are equivalent:*

1. $S \in \mathsf{str}\, Q$.
2. $S \subseteq (\sqsubseteq ; (S \cap Q))$.
3. $S = (\sqsubseteq ; (S \cap Q))$.

We have the following distributivity property for str:

Lemma 5.4 *Consider $N, P \subseteq A^*$. Then*

$$\mathsf{str}\,(N \cup P) = \mathsf{str}\, N \cup \mathsf{str}\, P \ .$$

This also shows once again the monotonicity of str. It should be noted, however, that str only distributes through finite unions and hence is not "continuous". For an instance of this see Example 5.7 below.

Note that it would not be adequate to work with the set $\mathsf{str}\,(\sqsubseteq ; P)$, the so-called *adherence* of P (see e.g. [59, 71]), instead of $\mathsf{str}\, P$. The reason is that infinite streams may "sneak" into a prefix closure although it results from a language of mutually $\sqsubseteq$-incomparable words which represent systems with finite behaviour only.

Example 5.5 The language $L \stackrel{\text{def}}{=} 0^* \bullet 1$ represents a behaviour with arbitrarily long but finite sequences of 0s terminated by the "explicit endmarker" 1. The words in L are mutually incomparable w.r.t. $\sqsubseteq$. Hence all directed subsets of L are singletons and their downward closures are principal streams and hence finite. So $\mathsf{str}\, L$ consists of finite streams only. However, the prefix closure $(\sqsubseteq ; L)$ contains the infinite stream 0^* representing the infinite stream 0^ω of 0s. So $\mathsf{str}\,(\sqsubseteq ; L) = \mathsf{str}\, L \cup \{0^\omega\}$. □

Using König's Lemma one can show that for finite A *every* infinite prefix-closed language contains an infinite stream. The general definition of str omits these undesired streams.

So, using str we can distinguish between erratic and angelic non-determinacy.

Example 5.6 Consider the recursive definition

$$\mathcal{B} = 0 \circ \mathcal{B} \ [\!] \ 1 \ ,$$

where $\circ$ denotes stream concatenation (see Section 5.9 for a precise definition) and $[\!]$ denotes non-deterministic choice. In an angelic interpretation of $[\!]$ always eventually the terminating branch 1 is chosen, and so $\mathcal{B}$ would equal $\mathsf{str}\, L$ of Example 5.5.

In an erratic interpretation of $[\!]$, on the other hand, no guarantee is given that the terminating branch will ever be chosen, and so $\mathcal{B}$ would equal $\mathsf{str}\,(\sqsubseteq ; L)$ of Example 5.5. □

We want to show now that str does not distribute through general union:

Example 5.7 Take $U = 0^*$. Then $U = \bigcup_{i \in \mathbb{N}} 0^i$. However, $\mathsf{str}\, U = \{0^*\} \cup \{(\sqsubseteq ; 0^i) : i \in \mathbb{N}\}$, whereas $\bigcup_{i \in \mathbb{N}} \mathsf{str}\, 0^i = \{(\sqsubseteq ; 0^i) : i \in \mathbb{N}\}$. □

We want to conclude this section by relating the str operator with temporal logic. Intuitively, a stream lies in $\mathsf{str}\, U$ iff it meets U "every now and then". Therefore the str operator corresponds to $\Box\Diamond$ as used in modal and temporal logic. This correspondence is made precise in [52, 53].

5.8 Infinite Streams

We define, for a behaviour $\mathcal{B}$, the set of its infinite streams as

$$\mathsf{inf}\,\mathcal{B} \stackrel{\text{def}}{=} \{S \in \mathcal{B} : |S| = \infty\} \ .$$

Clearly, inf distributes through arbitrary union and intersection:

$$\mathsf{inf}\,(\bigcup_{i\in I} \mathcal{B}_i) = \bigcup_{i\in I} \mathsf{inf}\,\mathcal{B}_i \ , \qquad \mathsf{inf}\,(\bigcap_{i\in I} \mathcal{B}_i) = \bigcap_{i\in I} \mathsf{inf}\,\mathcal{B}_i \ .$$

The following refinement laws will be used in the bounded buffer example in Section 5.17:

Lemma 5.8

$$\begin{array}{rcll} \mathsf{inf}\,\mathsf{str}\,(N \cup P) & = & \mathsf{inf}\,\mathsf{str}\,P & \Leftarrow \quad N \sqsubseteq P \wedge P \textit{ directed} \\ \mathsf{inf}\,\mathsf{str}\,N & = & \mathsf{inf}\,\mathsf{str}\,P & \Leftarrow \quad N \sqsubseteq P \wedge P \sqsubseteq N \wedge N, P \textit{ directed} \end{array}$$

The premise of this lemma states that N and P are majorants of each other, ie. that for every word in one a continuation can be found in the other. For that reason they record different ways of approximating the same set of "limits" as spanned by str.

Example 5.9 $\mathsf{inf}\,\mathsf{str}\,((a \bullet b)^* \bullet a) = \mathsf{inf}\,\mathsf{str}\,(a \bullet b)^*$. □

5.9 Stream Concatenation

As a prerequisite for defining infinite repetition we need stream concatenation which, for streams S, T is defined by

$$S \circ T \stackrel{\text{def}}{=} S \cup (\mathsf{max}\,S) \bullet T \ .$$

Let us explain this definition. If S is finite then $\mathsf{max}\,S$ is a singleton. This part of the overall behaviour then is prefixed to all traces in T to represent the concatenated behaviour. If S is infinite then $\mathsf{max}\,S = \emptyset$ and hence, by strictness of $\circ$, we get $S \circ T = S$, as is intuitively expected. We have

$$\mathsf{max}\,(S \circ T) = (\mathsf{max}\,S) \bullet (\mathsf{max}\,T) \ .$$

It is straightforward to show that $S \circ T$ is indeed a stream and that $(A^\infty, \circ, \varepsilon)$ is a monoid. As a shorthand notation we shall also allow words as first arguments of $\circ$. This is made precise by setting

$$u \circ T \stackrel{\text{def}}{=} (\sqsubseteq ; u) \circ T = (\sqsubseteq ; u) \cup u \bullet T \ .$$

Again, $\circ$ is extended pointwise to behaviours and, in the case of the above shorthand, to languages.

5.10 Infinite Repetition

We now give the usual greatest fixpoint definition of the set U^ω of streams that result from infinite repetition of words from a language $U \subseteq A^*$:

$$U^\omega \stackrel{\text{def}}{=} \nu\mathcal{X} \,.\, U \circ \mathcal{X}$$

(see Section 3.2 for the definition of ν). Hence we have

$$\begin{array}{l} U^\omega \;=\; U \circ U^\omega \;\wedge \\ \mathcal{X} \;=\; U \circ \mathcal{X} \;\;\Rightarrow\;\; \mathcal{X} \;\subseteq\; U^\omega \;. \end{array}$$

According to the Knaster-Tarski fixpoint theorem this is well-defined by monotonicity of $\circ$. Note that by this definition $\emptyset^\omega = \emptyset$. However, if $\varepsilon \in U$ then $U^\omega = A^\infty$. For that reason, U^ω is usually considered only for $\varepsilon \notin U$.

It should be noted that for $|U| \geq 2$ and $U \cap \varepsilon = \emptyset$ there are nontrivial solutions of $\mathcal{X} = U \circ \mathcal{X}$ properly less than U^ω. As an example consider the behaviour $U^* \circ \bigcup_{u \in U} u^\omega$ of all eventually periodic streams. We have

Lemma 5.10 *If $U \subseteq A^+$ satisfies the Fano condition, ie. the words in U are mutually incomparable w.r.t. $\sqsubseteq$, then*

$$U^\omega \;=\; \mathsf{inf}\,\mathsf{str}\,U^* \;.$$

Note that if $\varepsilon \in U$ then U satisfies the Fano condition iff $U = \varepsilon$; in this case the above equation doesn't hold, since then $\mathsf{inf}\,\mathsf{str}\,U^* = \emptyset$. It should also be mentioned that U satisfies the Fano condition iff $U = \mathsf{max}\,U$.

To see what happens if the Fano condition is not satisfied, we observe that by the recursion for $.^*$ and distributivity we have

$$\mathsf{inf}\,\mathsf{str}\,U^* \;=\; \mathsf{inf}\,\mathsf{str}\,\varepsilon \;\cup\; \mathsf{inf}\,\mathsf{str}\,U \;\cup\; \mathsf{inf}\,\mathsf{str}\,(U \bullet U^+) \;.$$

The first summand on the r.h.s. is $\emptyset$. However, the second summand may contain directed sets that do not correspond to infinite concatenations of words from U.

Example 5.11 Let $A = \{a, b\}$ and $U \stackrel{\text{def}}{=} \{a \bullet b^n : n \in \mathbb{N}\} \subseteq A^*$. Then $U \in \mathsf{dir}\,U^*$, since $U \subseteq U^*$ and U is directed. Hence $(\sqsubseteq ; U) = \varepsilon \cup U \in \mathsf{str}\,U^*$ and, since $(\sqsubseteq ; U)$ is infinite, we have even $(\sqsubseteq ; U) \in \mathsf{inf}\,\mathsf{str}\,U^*$. Now, $(\sqsubseteq ; U)$, viewed as a stream, represents an a followed by infinitely many bs; but this clearly does not arise from repeated concatenation of words in U. It is "sneaked in" by the fact that simply considering directed subsets of U^* throws away too much structural information. □

To allow a characterisation of U^ω for languages that do not satisfy the Fano condition, one can artificially enforce it by attaching a special endmarker to all words in U and remove it after singling out the infinite streams. To define it precisely we need as an auxiliary operation the **projection** of words to an alphabet $B \subseteq A$. It is defined inductively as follows:

$$\begin{array}{rcl} \varepsilon \dagger B & \stackrel{\text{def}}{=} & \varepsilon \\ (a \bullet s) \dagger B & \stackrel{\text{def}}{=} & \left\{ \begin{array}{ll} a \bullet (s \dagger B) & \text{if } a \in B \\ s \dagger B & \text{otherwise.} \end{array} \right. \end{array}$$

Projection is extended pointwise to languages and behaviours. The projection of a stream is a stream again. Now we have

Lemma 5.12 *Let $\# \notin A$ be a new letter and consider streams over the extended alphabet $A \cup \#$. Then for $U \subseteq A^+$,*

$$U^\omega \stackrel{\text{def}}{=} \mathsf{inf}\,\mathsf{str}\,(U \bullet \#)^* \dagger A \ .$$

For the somewhat tedious proof see [50].

The streams in $\mathsf{str}\,(U \bullet \#)^*$ correspond to finite and infinite sequences that result from concatenating arbitrary elements of U with the separator $\#$ in between. The operation inf then selects the infinite ones of these; if $\varepsilon \notin U$ these are precisely the infinite words that result from repeatedly concatenating words from U. The separators are used to record the "construction history" of the streams; they are finally thrown away again by the projection $\dagger A$. In this way subsets of U^* which are directed "by accident" are ignored. A similar mechanism for defining iteration is employed in [61] in the finite case and in [13] in the infinite case.

5.11 Transition Systems

It should be noted that our definitions of closures also apply to relations of higher arity. Hence one could use them on a representation of directed graphs with labeled edges by ternary relations $R \subseteq V \bullet L \bullet V$ where V is the set of nodes and L is the set of edge labels. R can, of course, also be viewed as a labeled transition relation.

Then $R^{\bowtie}$ is the set of all words $x_1 \bullet l_1 \bullet x_2 \bullet l_2 \bullet \cdots \bullet x_n \bullet l_n \bullet x_{n+1}$ with $x_i \in V$ and $l_i \in L$ such that $x_1 \bullet x_2 \bullet \cdots \bullet x_n \bullet x_{n+1}$ is a path in the graph or transition system and $l_1 \bullet l_2 \bullet \cdots \bullet l_n$ is the corresponding sequence of edge labels. By contrast, R^* is the set of all sequences $x \bullet l_1 \bullet l_2 \bullet \cdots \bullet l_n \bullet y$ with $x, y \in V$ and $l_i \in L$ such that there is a path in the graph or transition system from x to y and $l_1 \bullet l_2 \bullet \cdots \bullet l_n$ is the sequence of edge labels on that path.

By construction $R^{\rightsquigarrow}$ is a prefix-closed subset of A^*. So if we again view R as a transition relation, $R^{\rightsquigarrow}$ is the union of all streams of states generated by R. For a given state a then $\mathsf{str}\,(a \bowtie R^{\rightsquigarrow})$ is the set of all those streams that start with a.

Lemma 5.13 *$a \bowtie R^{\rightsquigarrow}$ is directed for all $a \in A$ iff R is deterministic, ie. satisfies $R^{\smile}; R \subseteq I_A$.*

Corollary 5.14 *For deterministic R we have $\mathsf{str}\,(a \bowtie R^{\rightsquigarrow}) = \{(\sqsubseteq\,; (a \bowtie R^{\rightsquigarrow}))\}$.*

To give two simple examples, we assume that $A = \mathbb{N}$ and consider the binary identity relation *id* and successor relation *succ* on A. Both relations are deterministic. Hence for $n \in \mathbb{N}$ the language $n \bowtie id^{\rightsquigarrow}$ represents the infinite constant stream of ns, whereas $n \bowtie succ^{\rightsquigarrow}$ represents the infinite stream of natural numbers from n on in ascending order.

Most interesting are binary generating relations R whose codomain $\mathsf{cod}\,R$ is non-empty and included in their domain $\mathsf{dom}\,R$, since then all paths can be extended to infinite ones. These properties are e.g. satisfied by total relations.

5.12 Streams of Functions

We have made no assumptions about our alphabet A. Hence it may even be a set of functions. Then streams over A model components with time-dependent behaviour. We have seen examples of this in the description of various faulty channels in Section 5.3.

A stream $S \in A^\infty$ of arguments is fed into a stream $F \in (A \to A^*)^\infty$ of functions by the operator $>>$. The images $f(a)$ of the elements a of A under the individual functions f in F are concatenated into an overall output stream. A wordwise definition of this is

$$\begin{array}{rcl} \varepsilon >> w & \stackrel{\text{def}}{=} & \varepsilon \ , \\ s >> \varepsilon & \stackrel{\text{def}}{=} & \varepsilon \ , \\ (a \bullet s) >> (f \bullet w) & \stackrel{\text{def}}{=} & f(a) \bullet (s >> w) \ . \end{array}$$

This operation is extended pointwise to languages and behaviours.

Example 5.15 For finite stream S we have

$$(S \bullet T) >> cchan_* \ = \ (S >> arbchan) \bullet (T >> cchan_*) \ .$$

This reflects the unbounded fairness of $cchan_*$: we have no guarantee *when* correct transmission occurs, and hence the elements of S may or may not be transmitted correctly. □

With bound assumptions one gets more precise information:

Example 5.16 We have

$$m > k \ \Rightarrow \ (a^m \bullet T) >> cchan_{\leq k} \ \in \ A^{\leq k} \bullet a \bullet A^\infty \ .$$

A channel with fairness bound k *must* transmit a correctly at least once if it receives more than k copies of a. □

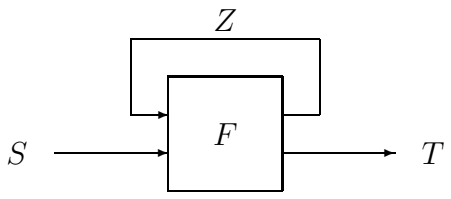
5.13 Feedback and State-Based Systems

5.13.1 The Feedback Operation

An essential operation on SPFs (cf. Section 5.4.1) is *feedback* of some outputs to the inputs. Assume an SPF $F : A^\infty \times B^\infty \to A^\infty \times C^\infty$. Then its feedback $feedF : B^\infty \to C^\infty$ is given by

$$(feedF)(S) \ = \ T \,\mathsf{where}\, (Z, T) = F(Z, S) \ .$$

It may be depicted as



The semantics of this recursive declaration is the usual least-fixpoint one. This version of the feedback operator hides the feedback stream. If this is to be made visible one simply copies it and feeds one copy back whereas the other is transmitted to the outside.

This operation together with streams of functions allows a very convenient and concise description of state-based systems.

Assume a set Q of states, an input alphabet A and an output alphabet B. Then a **time-dependent automaton** is given by a stream $H \in (Q \times A \to Q \times B)^\infty$.

We may now feed this automaton with a starting state $q_0 \in Q$ and a stream $S \in A^\infty$ of input values to produce a stream of output values in B^∞. The stream of states entered during the processing of the input is constructed by a feedback and hidden from the outside. This is described by

$$auto(H, q_0, S) = T \,\mathsf{where}\, (Z, T) = (q_0 \circ Z, S) >> H \ ,$$

in which Z is the stream of states that follow the initial state q_0.

By placing various restrictions on the entities involved, we can distinguish a hierarchy of automata:

- If no further restrictions are made, we obtain a **timed and state-dependent** automaton.
- If we require $|Q| = 1$ then we have a **timed and state-independent** automaton.
- If we take $H = f^\omega$ for some $f : Q \times A \to Q \times B$, we obtain a **timeless and state-dependent** automaton.
- If we again take $H = f^\omega$ but also require $|Q| = 1$, we have a **timeless and state-independent** automaton.

For example, an easy proof by induction over the structure of finite words shows

Lemma 5.17 *If* $|Q| = 1$ *then*

$$auto(f^\omega, q_0, S) \ = \ S >> g^\omega \ ,$$

where $g(i) \ = \ \pi_2(f(q_0, i))$.

We now illustrate the general case by the following

Example 5.18 We give a description of a one-place asynchronous buffer. The example is taken from [12]. Consider a set D of data. The input alphabet is

$$A \stackrel{\text{def}}{=} D \cup \{!\} \ .$$

An input $d \in D$ means that d is to be stored in the buffer, whereas ! means a request for the current contents of the buffer.

At each time point the buffer may accept or reject its input which is shown by a Boolean value. In addition to that the buffer will output data if it accepts the request signal. So we choose the output alphabet

$$B \stackrel{\text{def}}{=} (D \cup \{\varepsilon\}) \times \mathbb{B} \ ,$$

where ε models the case of no proper output.

As the set of states we choose

$$Q \stackrel{\text{def}}{=} D \cup \{\varepsilon\}$$

where ε models the state of being empty whereas $d \in D$ models the state of containing value d.

Now we define two transition functions

$$acc, rej : Q \times A \to Q \times B$$

which model acceptance and rejection of the input. We have

$$\begin{array}{rcl} acc(q,d) & = & (\mathsf{if}\ q = \varepsilon\ \mathsf{then}\ d\ \mathsf{else}\ q, (\varepsilon, q = \varepsilon)) \ , \\ acc(q,!) & = & (\varepsilon, (q, q \neq \varepsilon)) \ , \\ rej(q,x) & = & (q, (\varepsilon, \mathsf{false})) \ . \end{array}$$

The behaviour of a fair buffer, ie. one which rejects inputs only finitely many times before eventually accepting one, is then specified as

$$auto((rej^* \bullet acc)^\omega, \varepsilon) \ .$$

In particular, we can avoid the use of prophecy variables (see e.g. [12]) in this style. □

5.14 Processes and Synchronised Parallel Composition

While the previous two sections are appropriate for the SPF view of distributed systems, we now define operators that are adequate for the trace view (cf. Section 5.4). The particular definitions given here draw strongly on the corresponding ones in [26].

Assume an overall alphabet A for our streams. A **process** is a pair $(B, \mathcal{B})$ where $B \subseteq A$ is the **alphabet** of the process and $\mathcal{B} \subseteq B^\infty$ is a behaviour. We set

$$\alpha(B, \mathcal{B}) \stackrel{\text{def}}{=} B \ , \qquad \beta(B, \mathcal{B}) \stackrel{\text{def}}{=} \mathcal{B} \ .$$

Using projection (see Section 5.10) we can characterise processes in another way: the pair $(B, \mathcal{B})$ is a process iff $\forall\ S \in \mathcal{B} : S \dagger B = S$.

We need to lift the notion of refinement to processes. We allow that a process is refined by another one that has additional "internal" actions. Since then refinement amounts to inclusion of (the projection of) the behaviour, we abuse notation and write again $\subseteq$ for the refinement relation:

$$P \subseteq Q \Leftrightarrow \alpha P \supseteq \alpha Q \wedge (\beta P) \dagger \alpha Q \subseteq \beta Q \ .$$

In this case we say that P **refines** Q. It is easily checked that $\subseteq$ is a partial order on processes.

If behaviours are "loose enough" in that they allow arbitrary actions in between the "interesting" ones, one can model synchronised parallel composition very simply by intersection (see e.g. [27]). For general behaviours this works well only if they are "loosened" by interspersing arbitrary actions between the proper ones; this is again taken from [26]. The intersection then allows only traces in which the actions interesting to both partners occur in a sequence that is acceptable to both partners (ie. allowed in both behaviours) whereas the private actions of each partner are not constrained by the other partner.

Hence, for processes P, Q, we define the parallel composition $P||Q$ by setting

$$\begin{array}{rcl} \alpha(P||Q) & \stackrel{\text{def}}{=} & \alpha P \cup \alpha Q \ , \\ S \in \beta(P||Q) & \Leftrightarrow & S = S \dagger \alpha(P||Q) \wedge \\ & & S \dagger \alpha P \in \beta P \wedge \\ & & S \dagger \alpha Q \in \beta Q \ . \end{array}$$

Note, in particular, that || is commutative, associative and idempotent then. Moreover,

$$P \subseteq Q \;\Leftrightarrow\; P||Q = P \ .$$

If $\alpha P = \alpha Q$ then $\beta(P||Q) = \beta P \cap \beta Q$.

This parallel composition operator will be used in our extended example in Section 5.17.

5.15 Safety

We have already informally discussed safety and liveness (see e.g. [38, 2, 20]). We want to show how these notions can be expressed algebraically. In [2] and subsequent papers, a *property* is a set of infinite sequences of states. The appropriate counterpart in our setting is therefore a set of streams, ie. a *behaviour*. We note again that, since our definitions will be transformed into ones that only involve order-theoretic notions, out treatment again generalises to arbitrary ideal completions (see [54]).

5.15.1 Definition and Topological Properties

In [2] a behaviour $\mathcal{B} \subseteq A^\omega$ over infinite streams is called **safe** iff the following holds:

$$\forall\, S \in A^\omega : S \notin \mathcal{B} \Rightarrow (\exists\, s \in S : \forall\, T \in A^\omega : s \circ T \notin \mathcal{B}) \ .$$

This means that for every stream not in the behaviour there is a decisive finite prefix s where something went "irreparably wrong" in that *no* continuation of s can bring the computation back to the "good path".

Using straightforward logical rules, this formula can be transformed to

$$\mathsf{should}\,\mathcal{B} \subseteq \mathcal{B} \ ,$$

where

$$\mathsf{should}\,\mathcal{B} \stackrel{\text{def}}{=} \mathsf{str}\,\mathsf{pref}\,\mathcal{B}$$

and

$$\mathsf{pref}\,\mathcal{B} \stackrel{\text{def}}{=} \bigcup_{S \in \mathcal{B}} S$$

is the set of finite prefixes of a behaviour $\mathcal{B}$. Clearly, pref distributes through union and hence is $\subseteq$-monotonic.

As stated above, this simplified form involves only order-theoretic notions and hence generalises easily to arbitrary ideal completions. Note that for all $\mathcal{B} \subseteq A^\infty$ we have $\mathcal{B} \subseteq \mathsf{should}\,\mathcal{B}$. So a behaviour $\mathcal{B} \subseteq A^\infty$ is safe iff $\mathcal{B} = \mathsf{should}\,\mathcal{B}$. Moreover,

Lemma 5.19 *1. Safe behaviours are closed under arbitrary intersections and finite unions.*

2. should *is idempotent.*

3. $\mathsf{should}\,\mathcal{B}$ *is the least safe behaviour containing* $\mathcal{B}$*.*

By these properties, the safe behaviours coincide with the closed sets of a topology on A^∞ (cf. e.g. [70]) and should is the topological closure operator, in particular, a closure in the sense of Section 3.2.

Lemma 5.20 *For $P \subseteq A^*$, the behaviour $\mathsf{str}\, P$ is safe iff $(\sqsubseteq ; P) \subseteq P$, ie. iff P is prefix-closed.*

For that reason we call a snapshot set $P \subseteq A^*$ a **safety property** iff it is downward closed. We have

Corollary 5.21 *If $S \in A^\infty$ and P is a safety property, then $S \in \mathsf{str}\, P \Leftrightarrow S \subseteq P$.*

5.15.2 Continual Satisfaction

In connection with safety issues one is interested in the set of all objects that satisfy a property also in all their finite prefixes. Given a property $P \subseteq A^*$ we define the property $\mathsf{saf}\, P$ by

$$\mathsf{saf}\, P \stackrel{\text{def}}{=} \{x \in A^* : (\sqsubseteq ; x) \subseteq P\} .$$

The set $\mathsf{saf}\, P$ has also been termed the **prefix kernel** of P in [61, 78], since it is indeed a kernel operation as defined in Section 3.2. We have

Lemma 5.22 *1. $\mathsf{saf}\, P = P$ iff P is a safety property.*

2. saf is monotonic and strict w.r.t. $\emptyset$.

3. $\mathsf{saf}\,(P \cap Q) = \mathsf{saf}\, P \cap \mathsf{saf}\, Q$.

4. $S \in \mathsf{str}\,\mathsf{saf}\, P \Leftrightarrow S \subseteq P$.

Hence $\mathsf{saf}\, P$ is the greatest safety property contained in P. Moreover, saf distributes through intersection. Note that saf does not distribute through union. We can now state a further distributivity property for str:

Lemma 5.23 *Consider $N, P \subseteq A^*$. Then*

$$N = \mathsf{saf}\, N \Rightarrow \mathsf{str}\,(N \cap P) = \mathsf{str}\, N \cap \mathsf{str}\, P .$$

A grammar-like or automaton-like representation for safety properties of the form $\mathsf{saf}\, P$ for some $P \subseteq A^*$ can be derived using induction on the words involved (see again [54]). This yields

$$\begin{aligned} &\varepsilon \in \mathsf{saf}\, P \Leftrightarrow \varepsilon \in P , \\ &c \bullet s \in \mathsf{saf}\, P \Leftrightarrow c \bullet (\sqsubseteq ; s) \subseteq P . \end{aligned}$$

If P itself is already given in the form of an automaton-like recursion, then there is a systematic way for passing from that to a recursion for $\mathsf{saf}\, P$:

Lemma 5.24 *Suppose property $P \in \mathcal{P}(A^*)$ satisfies*

$$c \bullet U \subseteq P \Leftrightarrow U \subseteq F_c(P) .$$

Then, for $U \neq \emptyset$,

$$\begin{aligned} \varepsilon \in \mathsf{saf}\, P &\Leftrightarrow \varepsilon \in P , \\ c \bullet U \subseteq \mathsf{saf}\, P &\Leftrightarrow \varepsilon \in P \wedge c \in P \wedge U \subseteq \mathsf{saf}\, F_c(P) . \end{aligned}$$

Note that the assumption of the lemma means a Galois connection between $c\bullet$ and F_c. Assume now that we are given two properties P and Q.

Lemma 5.25 *Suppose P, Q satisfy*

$$(c \bullet U \subseteq P \Leftrightarrow U \subseteq F_c(P)) \wedge (c \bullet U \subseteq Q \Leftrightarrow U \subseteq G_c(P)) \ .$$

Then, for $U \neq \emptyset$,

$$\begin{aligned} \varepsilon \in \mathsf{saf}\,(P \cap Q) &\Leftrightarrow \varepsilon \in P \cap Q \ , \\ c \bullet U \subseteq \mathsf{saf}\,(P \cap Q) &\Leftrightarrow (\sqsubset ; c) \subseteq P \cap Q \wedge U \subseteq \mathsf{saf}\,(F_c(P) \cap G_c(Q)) \ . \end{aligned}$$

This corresponds to the construction of a product automaton.

5.16 Liveness

5.16.1 Definition and Topological Properties

Following again [2] we call a behaviour $\mathcal{B}$ over streams **live** iff

$$\forall\ s \in A^* : \exists\ T \in A^\omega : s \circ T \in \mathcal{B} \ .$$

Now, recalling the definition $\mathsf{pref}\,\mathcal{B} = \cup\mathcal{B}$ from Section 5.15.1, we can reduce this to

$$\forall\ s \in A^* : s \in \mathsf{pref}\,\mathcal{B}$$

and hence to

$$A^* \subseteq \mathsf{pref}\,\mathcal{B} \ .$$

Using this it is easy to show (see again [2])

Lemma 5.26 *$\mathcal{B}$ is live iff it is topologically dense in A^∞, ie. iff* $\mathsf{should}\,\mathcal{B} = A^\infty$.

Now we obtain

Theorem 5.27 *Every behaviour is the intersection of a live and a safe behaviour.*

Proof: We could copy the proof of the respective theorem in [2] verbatim, since it proceeds purely in topological terms. However, we give a simpler proof that avoids most of the topological reasoning in [2]. Assume $\mathcal{B} \subseteq A^\infty$. We have

$$\begin{aligned} & \mathcal{B} \\ = \quad & \{\![\ \text{since } \mathcal{B} \subseteq \mathsf{should}\,\mathcal{B}\ ]\!\} \\ & \mathsf{should}\,\mathcal{B} \backslash (\mathsf{should}\,\mathcal{B} \backslash \mathcal{B}) \\ = \quad & \{\![\ \text{definition of } \backslash, \text{ where } \neg\mathcal{C} \text{ denotes the complement} \\ & \quad \text{of } \mathcal{C} \text{ w.r.t. } A^\infty\ ]\!\} \\ & \mathsf{should}\,\mathcal{B} \cap \neg(\mathsf{should}\,\mathcal{B} \cap \neg\mathcal{B}) \\ = \quad & \{\![\ \text{de Morgan and double complement}\ ]\!\} \\ & \mathsf{should}\,\mathcal{B} \cap (\neg\mathsf{should}\,\mathcal{B} \cup \mathcal{B}) \ . \end{aligned}$$

Since $\mathsf{should}\,\mathcal{B}$ is safe, the claim is shown if $\neg\mathsf{should}\,\mathcal{B} \cup \mathcal{B}$ is live. We calculate

$$
\begin{array}{ll}
& \mathsf{should}\,(\neg\mathsf{should}\,\mathcal{B} \cup \mathcal{B}) \\
\supseteq & \quad \{\![\text{ since } \subseteq\text{-monotonic and hence superdistributive over } \cup \,]\!\} \\
& \mathsf{should}\,(\neg\mathsf{should}\,\mathcal{B}) \cup \mathsf{should}\,\mathcal{B} \\
\supseteq & \quad \{\![\text{ since } \mathsf{should} \text{ is extensive }]\!\} \\
& \neg\mathsf{should}\,\mathcal{B} \cup \mathsf{should}\,\mathcal{B} \\
= & \quad \{\![\text{ definition of complement }]\!\} \\
& A^\infty \ ,
\end{array}
$$

so that we are done by Lemma 5.26. □

Inspection of the proof shows that it can be abstracted to general Boolean algebras (see again [54]).

5.16.2 Liveness and Snapshot Sets

As in the case of safety, we now state when a property P spans a live behaviour. One obtains that $\mathsf{str}\,P$ is live iff $(\sqsubseteq\,;P) = A^*$. Hence we call $P \subseteq A^*$ a **liveness property** iff $(\sqsubseteq\,;P) = A^*$.

We now define that part of a snapshot set that is relevant for the infinite streams. We call a set $Q \subseteq A^*$ **lively** iff $Q \neq \emptyset \wedge \mathsf{max}\,Q = \emptyset$.

Now we define the **live part** of $P \subseteq A^*$ as

$$\mathsf{liv}\,P \stackrel{\text{def}}{=} \bigcup \mathcal{L}_P$$

where

$$\mathcal{L}_P \stackrel{\text{def}}{=} \{Q \subseteq P : Q \text{ lively}\} \ .$$

Then liv is again a kernel operator. Moreover,

$$\mathcal{L}_P \neq \emptyset \Rightarrow \mathsf{inf\,str}\,P \neq \emptyset \ .$$

Hence, to show that a snapshot set P spans infinite streams, it suffices to exhibit a lively $Q \subseteq P$. Such properties Q can frequently be constructed by induction.

5.17 Extended Example: Buffers and Queues

5.17.1 Specification of a Bounded Buffer

As an example of the use of our constructs, we give a specification of bounded buffer and queue modules. This example uses the trace view (cf. Section 5.4) of streams. It was motivated by the asynchronous bounded queue implementation in the collection of the IFIP WG10.5 benchmark problems in hardware verification [29].

The buffer module has one input and one output port. In describing such modules, we choose the letters a for the action of inputting and b for outputting and set $A \stackrel{\text{def}}{=} \{a, b\}$. Boundedness of a module can be enforced by requiring the number of input actions to exceed the number of output actions by at most some $n \in \mathbb{N}$ which then is the **capacity** of the device.

We denote by s_c the number of occurrences of $c \in A$ in $s \in A^*$. Formally,

$$
\begin{array}{rcl}
\varepsilon_c & \stackrel{\text{def}}{=} & 0 \ , \\
(a \bullet s)_c & \stackrel{\text{def}}{=} & \delta_{ac} + s_c \ ,
\end{array}
$$

where δ is the Kronecker symbol defined by

$$\delta_{xy} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x = y \ , \\ 0 & \text{otherwise} \ . \end{cases}$$

Generalising the above informal description slightly, we define, for $n \in \mathbb{Z}$ and $a, b \in A$, the set

$$\mathrm{EX}_n^{ab} \stackrel{\text{def}}{=} \{s \in A^* : s_a \leq s_b + n\}$$

of snapshots. Then $s \in \mathrm{EX}_n^{ab}$ may be pronounced as "a exceeds b by at most n in s". The specification is, however, very loose in that the balance between as and bs might be struck only at the very end of a word. For instance, $a^{k+n} \bullet b^k \in \mathrm{EX}_{ab}^n$. So the restriction may be violated in prefixes and only established in the end. For bounded devices, this is not possible. They need a stronger specification. Therefore we strengthen our snapshot set to the safety property

$$\mathrm{B}_n^{ab} \stackrel{\text{def}}{=} \mathsf{saf}\, \mathrm{EX}_n^{ab} \ .$$

Now $\mathsf{str}\, \mathrm{B}_n^{ab}$ is the set of all finite and infinite streams that satisfy EX_n^{ab} in all prefixes. However, we are interested in devices that work for an unbounded time. This is specified by considering as overall behaviour of such a device the set

$$\mathcal{B}_n^{ab} \stackrel{\text{def}}{=} \mathsf{inf}\, \mathsf{str}\, \mathrm{B}_n^{ab}$$

consisting only of infinite admissible streams.

A buffer is a device in which the number of outputs must not exceed the number of inputs. Hence we define

$$\mathcal{BF}^{ab} \stackrel{\text{def}}{=} \mathcal{B}_0^{ba} \ .$$

Note the reversal of the arguments in the superscript. The finitary property B_0^{ba} spells out to $s_b \leq s_a$, as required. This describes an unbounded buffer. A bounded buffer of capacity n then is described by

$$\mathcal{BB}_n^{ab} \stackrel{\text{def}}{=} \mathcal{BF}^{ab} \cap \mathcal{B}_n^{ab} \ .$$

This specifies the set of all infinite streams for which, in all finite prefixes, the number of outputs does not exceed the number of inputs and the number of inputs does not exceed the number of outputs by more than n.

5.17.2 Transformation to Automaton Form

We consider now again the family of properties EX_n^{ab}. From its predicative definition in the previous section we want to calculate a more "operational" description corresponding to a generating grammar or accepting automaton. This can be done by a simple unfold/fold transformation using induction on the words in A^*. For the induction basis we calculate

$$\begin{array}{ll} & \varepsilon \in \mathrm{EX}_n^{ab} \\ \Leftrightarrow & \quad \{\!\![\text{ definition of EX }]\!\!\} \\ & \varepsilon_a \leq \varepsilon_b + n \\ \Leftrightarrow & \quad \{\!\![\text{ definition of count }]\!\!\} \\ & 0 \leq 0 + n \\ \Leftrightarrow & \quad \{\!\![\text{ arithmetic }]\!\!\} \\ & 0 \leq n \ . \end{array}$$

For the induction step, we consider an arbitrary $c \in A$:

$$
\begin{array}{ll}
& c \bullet s \in \mathrm{EX}_n^{ab} \\
\Leftrightarrow & \quad \{\![\text{ definition of EX }]\!\} \\
& (c \bullet s)_a \leq (c \bullet s)_b + n \\
\Leftrightarrow & \quad \{\![\text{ definition of count }]\!\} \\
& \delta_{ca} + s_a \leq \delta_{cb} + s_b + n \\
\Leftrightarrow & \quad \{\![\text{ arithmetic }]\!\} \\
& s_a \leq s_b + n + \delta_{cb} - \delta_{ca} \\
\Leftrightarrow & \quad \{\![\text{ definition of EX }]\!\} \\
& s \in \mathrm{EX}^{ab}_{n+\delta_{cb}-\delta_{ca}} \ ,
\end{array}
$$

Note that the recursion relations are linear bi-implications. Therefore we obtain, for $U \neq \emptyset$,

$$
\begin{aligned}
\varepsilon \in \mathrm{EX}_n^{ab} &\Leftrightarrow 0 \leq n \ , \\
c \bullet U \ \subseteq \ \mathrm{EX}_n^{ab} &\Leftrightarrow U \ \subseteq \ \mathrm{EX}^{ab}_{n+\delta_{cb}-\delta_{ca}} \ .
\end{aligned}
$$

This corresponds to an infinite grammar with nonterminals EX_n^{ab} or an infinite automaton with states EX_n^{ab} ($n \in \mathbb{Z}$).

5.17.3 Counting Resumed

Next we want a similar representation for

$$
\mathrm{B}_n^{ab} \stackrel{\text{def}}{=} \mathsf{saf}\, \mathrm{EX}_n^{ab} \ .
$$

This can be done quite systematically using Lemma 5.24. We obtain, for $U \neq \emptyset$,

$$
\begin{aligned}
\varepsilon \in \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ , \\
c \bullet U \ \subseteq \ \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ \wedge \ 0 \leq n \ \wedge \ U \ \subseteq \ \mathrm{B}_n^{ab} \text{ for } c \in A \backslash \{a, b\} \ , \\
a \bullet U \ \subseteq \ \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ \wedge \ 0 \leq n - 1 \ \wedge \ U \ \subseteq \ \mathrm{B}_{n-1}^{ab} \ , \\
b \bullet U \ \subseteq \ \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ \wedge \ 0 \leq n + 1 \ \wedge \ U \ \subseteq \ \mathrm{B}_{n+1}^{ab} \ .
\end{aligned}
$$

which simplifies to

$$
\begin{aligned}
\varepsilon \in \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ , \\
c \bullet U \ \subseteq \ \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ \wedge \ U \ \subseteq \ \mathrm{B}_n^{ab} \text{ for } c \in A \backslash \{a, b\} \ , \\
a \bullet U \ \subseteq \ \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n - 1 \ \wedge \ U \ \subseteq \ \mathrm{B}_{n-1}^{ab} \ , \\
b \bullet U \ \subseteq \ \mathrm{B}_n^{ab} &\Leftrightarrow 0 \leq n \ \wedge \ U \ \subseteq \ \mathrm{B}_{n+1}^{ab} \ .
\end{aligned}
$$

In particular, $\mathrm{B}_n^{ab} = \emptyset$ for $n < 0$.

Now we consider the bounded buffer behaviour. We calculate:

$$
\begin{array}{ll}
& \mathcal{BB}_n^{ab} \\
= & \quad \{\![\text{ definition }]\!\} \\
& \mathcal{BF}^{ab} \cap \mathcal{B}_n^{ab} \\
= & \quad \{\![\text{ definition }]\!\} \\
& \mathcal{B}_0^{ba} \cap \mathcal{B}_n^{ab} \\
= & \quad \{\![\text{ definition }]\!\}
\end{array}
$$

$$
\begin{array}{ll}
& \mathsf{inf\ str}\, \mathrm{B}_0^{ba} \cap \mathsf{inf\ str}\, \mathrm{B}_n^{ab} \\
= & \quad \{\![\text{ by Corollary 5.8, since the B sets are specified as safety properties }]\!\} \\
& \mathsf{inf\ str}\, (\mathrm{B}_0^{ba} \cap \mathrm{B}_n^{ab}) \ .
\end{array}
$$

So the problem has been reduced to finding an explicit representation for $\mathrm{B}_0^{ba} \cap \mathrm{B}_n^{ab}$, which is a simple product automaton construction. It is a special case of the automaton for

$$\mathrm{G}_{mn} \stackrel{\text{def}}{=} \mathrm{B}_m^{ba} \cap \mathrm{B}_n^{ab} \ .$$

5.17.4 Decomposition

Let us now define a buffer process by setting

$$\mathrm{BB}_m^{ab} \stackrel{\text{def}}{=} (\{a, b\}, \mathcal{BB}_m^{ab}) \ .$$

Then using parallel composition we can state the following nice decomposition properties:

Lemma 5.28 *1.* $\mathrm{EX}_m^{ab} \cap \mathrm{EX}_n^{bc} \subseteq \mathrm{EX}_{m+n}^{ac}$.

2. $\mathrm{B}_m^{ab} \cap \mathrm{B}_n^{bc} \subseteq \mathrm{B}_{m+n}^{ac}$.

3. $S \dagger \{a, b\} \in \mathcal{BB}_m^{ab} \wedge S \dagger \{b, c\} \in \mathcal{BB}_n^{bc} \Rightarrow S \dagger \{a, c\} \in \mathcal{BB}_{m+n}^{ac}$.

4. $\mathrm{BB}_m^{ab} \mid\mid \mathrm{BB}_n^{bc} \subseteq \mathrm{BB}_{m+n}^{ac}$.

Proof:

1. $$
\begin{array}{ll}
& s \in \mathrm{EX}_m^{ab} \cap \mathrm{EX}_n^{bc} \\
\Leftrightarrow & \quad \{\![\text{ definition }]\!\} \\
& s_a \leq s_b + m \wedge s_b \leq s_c + n \\
\Rightarrow & \quad \{\![\text{ transitivity and monotonicity }]\!\} \\
& s_a \leq s_c + m + n \\
\Leftrightarrow & \quad \{\![\text{ definition }]\!\} \\
& s \in \mathrm{EX}_{m+n}^{ac} \ .
\end{array}
$$

2. immediate from 1., Lemma 5.22.3 and Lemma 5.22.2.

3. immediate from 2.

4. immediate from 3.

□

This allows decomposing a buffer of capacity n into a parallel composition of n buffers of capacity 1. Of course, it needs to be shown that the intersections/parallel compositions are non-empty. This follows from our results in Section 5.16: first, it is easy to show that, for $n \geq 1$,

$$
\begin{array}{rcl}
(a \bullet b)^* & \subseteq & \mathrm{EX}_n^{ab} \ , \\
\mathsf{inf\ str}\, (a \bullet b)^* & \subseteq & \mathcal{B}_n^{ab} \ .
\end{array}
$$

From this we get, for $m, n \geq 1$,

$$\mathsf{inf\ str}\, (a \bullet b \bullet c)^* \subseteq \beta(\mathrm{BB}_m^{ab} \mid\mid \mathrm{BB}_n^{bc}) \ .$$

Since $(a \bullet b \bullet c)^*$ is lively, $\mathsf{inf\ str}\, (a \bullet b \bullet c)^*$ and hence $\mathrm{BB}_m^{ab} \mid\mid \mathrm{BB}_n^{bc}$ are non-empty.

5.17.5 The One-Place Buffer

For the special case of $n = 1$ we have

$$\mathcal{BB}_1^{ab} \;=\; \mathsf{inf\ str}\, \mathrm{G}_{01} \;,$$

where, for $U \neq \emptyset$,

$$\begin{array}{rlrl} \varepsilon \in \mathrm{G}_{01} & \Leftrightarrow \mathrm{TRUE}\;, & \varepsilon \in \mathrm{G}_{10} & \Leftrightarrow \mathrm{TRUE}\;, \\ a \bullet U \subseteq \mathrm{G}_{01} & \Leftrightarrow U \subseteq \mathrm{G}_{10}\;, & a \bullet U \subseteq \mathrm{G}_{10} & \Leftrightarrow \mathrm{FALSE}\;, \\ b \bullet U \subseteq \mathrm{G}_{01} & \Leftrightarrow \mathrm{FALSE}\;, & b \bullet U \subseteq \mathrm{G}_{10} & \Leftrightarrow U \subseteq \mathrm{G}_{01}\;. \end{array}$$

This corresponds to a two-state accepting automaton for the bounded buffer property, which is sufficient for purposes of implementation.

However, the above can also be seen as a regular grammar or system of equations for languages. We can calculate from it a regular expression for $\mathrm{BB}_1^{ab} = \mathrm{G}_{01}$ using twice Arden's Rule (24). This gives

$$\mathrm{BB}_1^{ab} \;=\; (a \bullet b)^* \bullet (\varepsilon \cup a)\;.$$

Using

$$(a \bullet b)^* \bullet a \sim (a \bullet b)^*$$

and Corollary 5.8, we obtain

$$\mathcal{BB}_1^{ab} \;=\; \mathsf{inf\ str}\, (a \bullet b)^*\;.$$

Finally we use the fact that the language $a \bullet b$ as a singleton trivially satisfies the Fano condition, so that Lemma 5.10 gives

$$\mathcal{BB}_1^{ab} \;=\; (a \bullet b)^\omega\;,$$

as expected.

5.17.6 From Buffers to Queues

So far we have only talked about the relative order of input and output *events*. For queues also the relative order of input and output *values* is relevant. We use now the refined alphabet $A = C \times V$ where C is the set of channel names and V the set of values. An element of A will be denoted as $c\langle v\rangle$. As a shorthand we introduce

$$c \;=\; \{c\langle x\rangle : x \in V\}\;. \tag{39}$$

For a word $w \in A^*$ we define the word $\mathit{chans}(w) \in C^*$ of channels on which activity occurred and for each $c \in C$ the word $\mathit{vals}_c(W) \in V^*$ of values transmitted along c. Their inductive definitions read

$$\begin{array}{rcl} \mathit{chans}(\varepsilon) & = & \varepsilon\;, \\ \mathit{chans}(b\langle x\rangle \bullet w) & = & b \bullet \mathit{chans}(w)\;, \end{array}$$

$$\begin{array}{rcl} \mathit{vals}_c(\varepsilon) & = & \varepsilon\;, \\ \mathit{vals}_c(b\langle x\rangle \bullet w) & = & \left\{ \begin{array}{ll} x \bullet \mathit{vals}_c(w) & \text{if } b = c\;, \\ \mathit{vals}_c(w) & \text{otherwise}\;. \end{array} \right. \end{array}$$

These operations are extended pointwise to languages and behaviours.

With these operations we may specify the behaviour of a **faithful** component, ie. a component which does not re-order or lose messages when transmitting from channel a to channel b, as

$$\mathcal{FA}^{ab} \stackrel{\text{def}}{=} \{S : vals_a(S) = vals_b(S)\} .$$

A bounded queue is then specified as a faithful bounded buffer:

$$\mathcal{BQ}_n^{ab} \stackrel{\text{def}}{=} \mathcal{FA}^{ab} \cap \mathcal{BB}_n^{ab} .$$

Here a, b in $\mathcal{BB}_n^{ab}$ are to be understood according to abbreviation (39).

The decomposition properties for buffers carry over to queues, so that again a queue of capacity n can be refined into a parallel composition of n queues of capacity 1. Moreover, a similar calculation as before, using again Arden's rule, yields for the refinement

$$\mathcal{BQ}_1^{ab} = (\bigcup_{x \in V} a\langle x\rangle \bullet b\langle x\rangle)^\omega .$$

6 Network Algebra and Sequential Hardware

In this section we show how algebraic structures can be employed in calculational construction of hardware.

6.1 The Framework

We model hardware functionally in *Haskell*. The reasons for this are the following.

- Functional languages support various views of streams directly, eg. as lazy lists or functions from time to data.
- Polymorphism allows generic formulations and hence supports re-use.
- Since all specifications are executable, direct prototyping is possible.
- Functional languages are being considered for their suitability as bases of modern hardware description languages. Examples are — in historical order — Hydra [60], MHDL [64] (unfortunately abandoned), Lava [68], HAWK [25] and SLDL [69] (which is still in the requirements definition phase). Many other approaches to hardware specification and verification also use higher-order concepts to good advantage (see e.g. [24]).
- A transformation system ULTRA for the *Gofer* sublanguage of *Haskell* is being constructed at the University of Ulm [67]. It is an adaptation of the system CIP-S [8, 7]. A prototype version of ULTRA has been used to formally check most of the laws and derivations in our paper (which originally were done with paper and pencil) by machine. While the overall derivations were found to be correct, a number of minor errors and missing side conditions were discovered. The set of transformation rules obtained in this way can be re-used for further derivations that now, of course, should take place directly on the system.

The introduction of small but very effective sets of algebraic laws for special subproblem areas such as the treatment of delay and slowdown makes the approach particularly streamlined. Some of these laws correspond to manipulations of layout graphs; whenever appropriate, we therefore perform the derivations at the graphical level to make them easier to grasp.

The approach has also been used quite successfully in teaching the essentials of hardware to first-year students (who, of course, had been exposed to the *Gofer* sublanguage of *Haskell* in the beginning of the year).

A brief review of the essential concepts of *Haskell* and a notational extension that are used in this paper can be found in the Appendix.

6.2 Modules as Functions

6.2.1 The Basic Model

A hardware module will be modeled as a function taking a list of input streams to a list of output streams. Using lists of inputs and outputs has the advantage that the basic connection operators can be defined independent of the arities of the functions involved. The disadvantage is that we need uniform typing for all inputs/outputs, since *Haskell* does not allow heterogeneous lists.

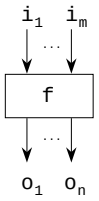
It turns out, though, that many of our combining operators are independent of the case of stream processing modules. Therefore we keep the treatment polymorphic in the types of input and output lists. This allows a uniform treatment of combinational and sequential circuits as elaborated in [55]. In this view, a function `f` describing a module with `m` inputs and `n` outputs will have the type `f :: Module` with

```
type Module = [a] -> [a]
```

However, `f` will be defined only for input lists of length `m` and always produce output lists of length `n`. Assuming

```
[o1,...,on] = f [i1,...,im]
```

we represent such a module diagrammatically as



The input/output lines are numbered from left to right or, if the picture is rotated to the left by 90 degrees, from bottom to top, in the corresponding lists.

In the case of synchronous hardware, each element of a stream reflects a value on the respective input/output line at one clock tick.

6.2.2 Pointwise and Point-Free Reasoning

We now discuss briefly the role of functions as modules of a system. In a higher-order language such as *Haskell* there are two views of functions:

- as routines with a body expression that depends on the formal parameters, as in conventional languages ("pointwise" view);
- as "black boxes" ("point-free" view) that can be freely manipulated by higher-order functions (combinators).

The latter view is particularly adequate for functional hardware descriptions, since it allows the direct definition of various composition operations for hardware modules.

However, contrary to other approaches we do not reason purely at the combinator level, i.e. without referring to individual in/output lines. While the combinator view often has advantages, it can become quite tedious in other places. So we prefer to have the possibility to switch.

The basis for reasoning about functions is the extensionality rule

```
f = g   iff f x = g x  for all x .
```

Many algebraic laws we use are equalities between functions, interpreted as extensional equalities.

Example 6.1 Function composition is defined in *Haskell* by

```
(f . g) x  =  f (g x)
```

with polymorphic combinator

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

A fundamental law is associativity of composition:

```
(f . g) . h  =  f . (g . h)
```

□

6.3 Another Model of Streams

As already mentioned, we model sequential hardware modules as stream transformers. The streams are used to describe the temporal succession of values on the connection wires. In this paper we deal with discrete time only. Even this leaves several options how to represent streams. One possibility would be to define

```
type Stream a = [a]
```

Since *Haskell* employs a lazy semantics, this allows finite as well as infinite streams. Time remains implicit, but can be introduced using the list indexing operation. This view of streams coincides with the one presented in Section 5. In the present paper we use a version which explicitly refers to time:

```
type Time = Int
type Stream a = Time -> a
```

This will carry over easily to real time. On the other hand, this does not directly support finite streams. They have to be modeled eg. by functions that become eventually constant, preferably yielding only the pseudo-value `undefined` after the "proper" finite part.

6.4 Modelling Connections

6.4.1 Rubber and Rigid Connections

We shall employ two views of connections between modules:

- that of "rubber wires", represented by formal parameters or implicitly by plugging in subexpressions as operands;
- that of "rigid wires", represented by special routing functions which are inserted using basic composition combinators.

Contrary to other approaches (e.g. [28, 30]) we proceed in two stages:

- We start at the level of rubber wiring to get a first correct implementation.
- Then we (mechanically) get rid of formal parameters by combinator abstraction to obtain a version with rigid wiring.

This avoids introducing wiring combinators at too early a stage and carrying them through all the derivation in an often tedious manner.

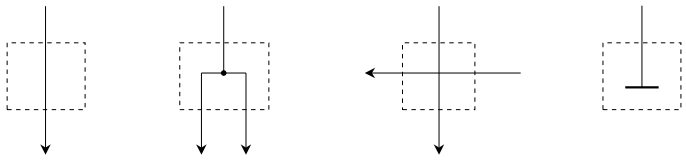
Formally, the transition form the rubber to the rigid view of wires is achieved by eliminating all formal parameters from functional expressions in favour of parallel and sequential composition and the basic wiring elements. This is analogous to the process of λ-abstraction in combinatory logic (see eg. [6]). Therefore we term this operation **combinator abstraction** (see [55] for details).

In drawing diagrams we shall be liberal and use views in between rubber and rigid wiring. In particular, we shall use various directions for the input and output arrows. So input arrows may not only enter at the top but also from the right or from the left; an analogous remark holds for the output arrows.

6.4.2 Basic Wiring Elements

The basic wiring elements are a straight wire, modeled by the identity function, the fan-out of degree 2 (fork), the crossing (swap) and the sink:

```
id   [x]     =  [x]        fork [x]    =  [x,x]
swap [x,y]   =  [y,x]      sink [x]    =  []
```



These operations can be extended in a straightforward way to wire bundles.

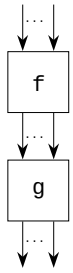
Finally, we have the invisible module `ide` with `0` inputs and `0` outputs:

```
ide [] = []
```

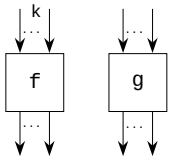
6.4.3 Sequential and Parallel Composition

Sequential composition simply is reverse function composition. We are a bit sloppy here about the arities of the functions; this has again to do with the already mentioned absence of tuples as first-class citizens. For parallel composition we need to tell the operator how many inputs are to be distributed to the first function; the remaining ones go to the second function.

```
(f  |> g ) xs   =   g  (f xs)
```




```
par k f g xs  =  f  (take k xs) ++ g (drop k xs)
```



We abbreviate `par 1` by the infix operator `|||`.

6.4.4 Basic Laws (Network Algebra I)

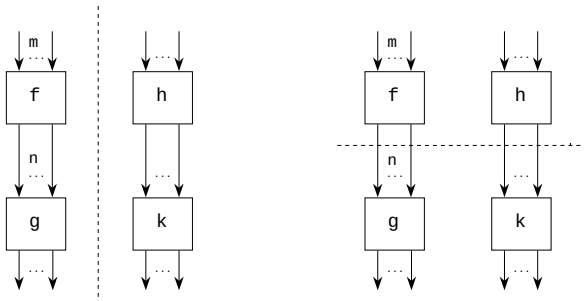
All semantic models for graph-like networks should enjoy a number of natural properties which reflect the abstraction that lies in the graph view. A systematic account of these properties has been given in [72].

Associativity:

```
f  |> (g |> h)  =  (f |> g) |> h
(f `par m` g) `par (m+k)` h   =  f `par m` (g `par k` h)
```

Abiding Law I:

```
((f |> g) `par m` (h |> k)) xs  =  ((f `par m` h) |> (g `par n` k)) xs
    <==  n = length (f (take m xs))
```



Neutrality:

```
id |> f  =  f  =  f |> id
(f `par m` ide) xs  =  f xs   <==   length xs = m
ide `par 0` f  =  f
```

Involution:

```
(swap |> swap) xs  =  xs   <==   length xs  =  2
```

Whereas associativity and abiding just allow "parenthesis-free layouts", use of neutrality or involution means simplification/complexification of abstract layouts.

6.5 Basic Modules

6.5.1 Lifting and Constant

To establish the connection with combinational circuits we need to iterate their behaviour in time. To this end we introduce liftings of operations on data to streams. A "unary" operation takes a singleton list of input data and produces a singleton list of output data. This is lifted to a function from a singleton list of input streams to a singleton list of output streams. It is the analogue of the apply-to-all operation map on lists. Since streams are functions themselves, the lifting may also be expressed using function composition. We have

```
lift1 :: (a -> b) -> [Stream a] -> [Stream b]
lift1 f [d]  =  [\ t -> f (d t)]
```

Alternatively, since streams are functions themselves, the lifting may also be expressed using function composition:

```
lift1 f [d]  =  [f.d]
```

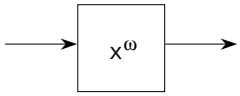
Similarly, we have for binary operations

```
lift2 :: (a -> a -> b) -> [Stream a] -> [Stream b]
lift2 g [d,e]   =  [\t -> g (d t) (e t)]
```

The inscriptions of the boxes follow notationally the view of infinite streams of functions used in Sections 5.3 and 5.12.

Another useful building block is a module that emits a constant output stream. For convenience we endow it with a (useless) input stream. So this module can also be viewed as the combination of a sink and a source. We define

```
cnst :: a -> [Stream b] -> [Stream a]
cnst x   =   lift1 (const x)
```

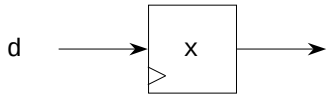


Here `const` is a predefined *Haskell* function that produces a constant unary function from a value.

6.5.2 Initialised Unit Delay

To model memory of the simplest kind we use a unit delay module. Other delays such as inertial delay or transport delay can be modeled similarly. For a value `x` the stream transformer `(x>-)` shifts its input stream by one time unit; at time `0` it emits `x` as the initial value:

```
(>-) :: a -> [Stream a] -> [Stream a]
(x >- [d])  =  [e]
  where  e t  |  t == 0  =  x
              |  t  > 0  =  d (t-1)
```



We now state laws for pushing delays through larger networks. They allow one, for each circuit constructor, to shift delay elements from the input side to the output side or vice versa under suitable change of the initialisation values. These laws are used centrally in our treatment of systolic circuits.

Lemma 6.2 (Delay Propagation Rules) *If* `f` *is strict, ie. undefined whenever its argument is, then*

```
(x>-) |> lift1 f    =  lift1 f  |> ((f x) >-)
```

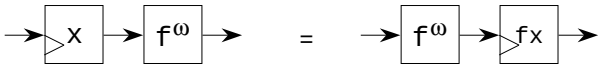
If `g` *is doubly strict, ie. is undefined whenever both its argument are, then*

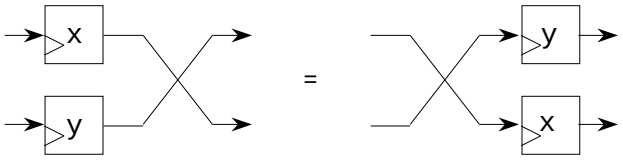
```
((x>-) ||| (y>-)) |> lift2 g  =  lift2 g  |> ((g x y)>-)
```

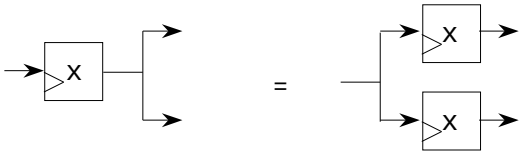
Moreover,

```
(x>-)              |> cnst y  =  cnst y |> (y>-)
((x>-) ||| (y>-)) |> swap    =  swap   |> ((y>-) ||| (x>-))
(x>-)              |> fork    =  fork   |> ((x>-) ||| (x>-))
```

These rules can be given in pictorial form as







For propagation through `|>` and `par` we may use associativity of `|>` and the abiding law. These simple laws are quite effective as will be seen in later examples. In our derivation of systolic circuits we actually use them in the direction from right to left to shift delays from outputs to inputs.

6.6 Feedback

6.6.1 The Feedback Operation

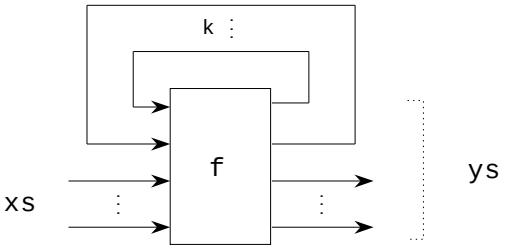
Another essential ingredient of systems with memory is feedback of some outputs to inputs, as already discussed in Section 5.13.1. We use

```
feed ::  Int  -> ([a] -> [a]) -> ([a] -> [a])
```

where the first parameter indicates how many outputs are fed back. The definition reads

```
feed k f xs  =  codrop k ys
   where ys  =  f (xs ++ cotake k ys)

cotake n xs  =  drop (length xs - n) xs
codrop n xs  =  take (length xs - n) xs
```



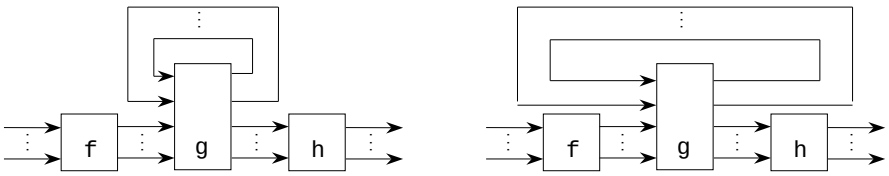
Note the recursive definition of `ys` which reflects the flowing back of information. This recursion is well-defined by the lazy semantics of *Haskell*.

6.6.2 Properties of Feedback (Network Algebra II)

The feedback operation enjoys a number of algebraic laws which show that it models the rubber wire abstraction correctly. For a systematic exposition see again [72].

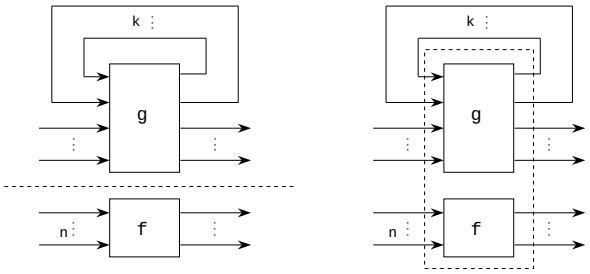
Stretching wires:

```
f  |>  feed k g  |>  h   =   feed k ((f `par m` id) |>  g  |>
                                  (h `par n` id) )
```



Abiding law II:

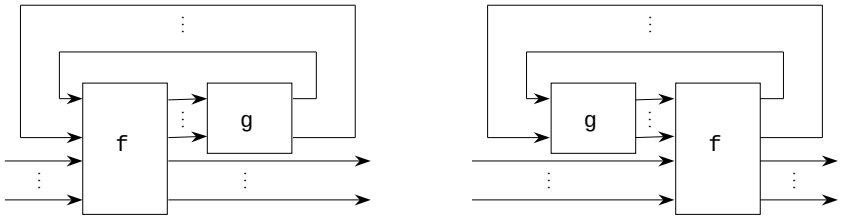
```
f `par n` feed k g   =   feed k (f `par n` g)
```



Shifting a module:

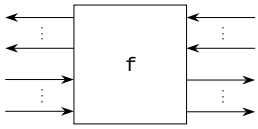
Assume that `f` is defined only for input lists of length `m+k` and always produces output lists of length `n+k` and that `g` is defined only for input lists of length `k` and always produces output lists of length `k`. Then we have

```
feed k (f |> (id `par n` g))   =   feed k ((id `par m` g) |> f)
```



6.7 Interconnection (Mutual Feedback)

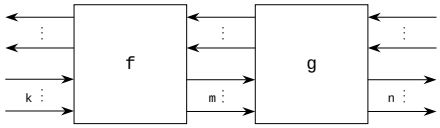
In more complex designs it may be convenient to picture a module `f` with inputs and outputs distributed to both sides:



We want to compose two such functions to model interconnection of the respective modules. To this end we introduce

```
connect :: Int -> Int -> Int -> Module -> Module -> Module
```

The three `Int`-parameters in `connect k m n f g` indicate that `k` inputs are supposed to come from the left neighbour of `f`, that `m` wires lead from `f` to `g`, and that `n` outputs go to the right neighbour of `g`.



We define therefore

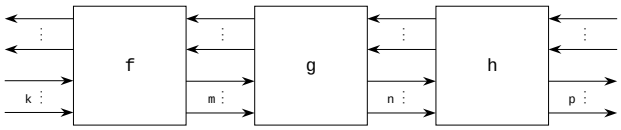
```
connect k m n f g xs  =  take n zs ++ drop m ys
  where ys  =  f (take k xs ++ drop n zs)
        zs  =  g (take m ys ++ drop k xs)
```

This involves a mutually recursive definition of `ys` and `zs` which again is well-defined by the lazy *Haskell* semantics.

Lemma 6.3 *Interconnection is associative in the following sense:*

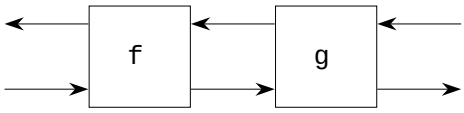
```
(f  `connect k m n` g)  `connect k n p`  h   =
   f  `connect k m n`  (g  `connect k m p`  h)
```

Moreover, `connect` *has the identity* `id` *as its neutral element.*



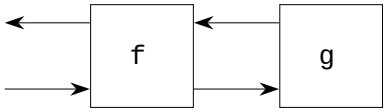
Two interesting special cases are

```
f =||= g  =  connect 1 1 1 f g
```

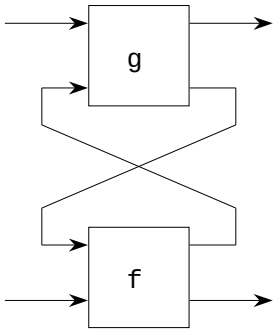


and

```
f =| g  =  connect 1 1 0 f g
```

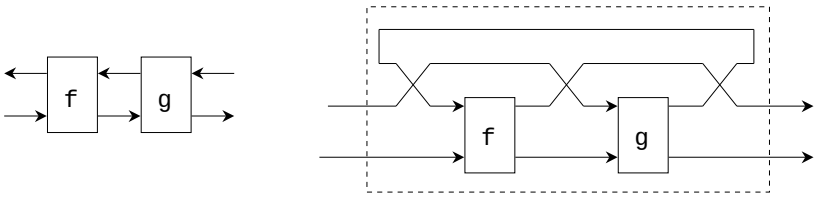


The operator `=||=` is also known as mutual feedback $\otimes$. The corresponding network can be depicted as



Using a suitable torsion of the network we can relate interconnection to feedback:

```
f1 =||= f2  =
   feed 1 ( (id ||| swap) |> (f1 ||| id) |> (id ||| swap) |>
                             (f2 ||| id) |> (id ||| swap)    )
```



With the help of this connection, the proof of Lemma 6.3 can be given using purely the laws of network algebra. Hence the lemma is valid for all models of network algebra, not just our particular one.

6.8 Example: A Convolver

We want to tackle a somewhat more involved example now. In particular, we want to prepare the way to systolic circuits. We treat a convolver, another of the IFIP verification benchmarks [29].

A non-programmable convolver of degree `n` uses `n` fixed weights to compute at each time point `t >= n` the convolution of its previous `n` inputs by these weights. Mathematically the convolution is defined as

$$\sum_{i=1}^{n} w_{n-i} * d(t-i) \ ,$$

where d is the input stream and the w_j are the weights. Convolution is used eg. in digital filters.

6.8.1 Specification

Using list comprehension, the above mathematical definition can be directly transcribed into a *Haskell* specification. For convenience we collect the weights into another stream `w`. Then the convolver is specified by

```
conv :: Stream Int -> Int -> [Stream Int] -> [Stream Int]
conv w n [d]  =
   [ \t -> if t < n
             then undefined
             else sum [ w (n-i) * d (t-i) | i <- [1..n] ] ]
```

It should be clear that the problem generalises to arbitrary compositions of fold and apply-to-all operations.

6.8.2 Derivation of a Convolver Circuit

We now want to derive from the formal specification a regular layout described by a recursion. The obvious parameter to drive the recursion is the number `n` of terms in the sum, since the summation function is defined recursively itself and we can carry over its recursion structure to the convolver circuit.

The base case is `n = 0`. For `t >= 0` and `[e] = conv w 0 d` we calculate

```
  e t
=   {| specification of e |}
  sum [ w (0-i) * d (t-i) | i <- [1..0] ]
=   {| definition of intervals |}
  sum [ w (0-i) * d (t-i) | i <- [] ]
=   {| definition of list comprehension |}
  sum []
=   {| definition of sum |}
  0
```

Hence `conv 0 = cnst 0` with `cnst` defined as in Section 6.5.1.

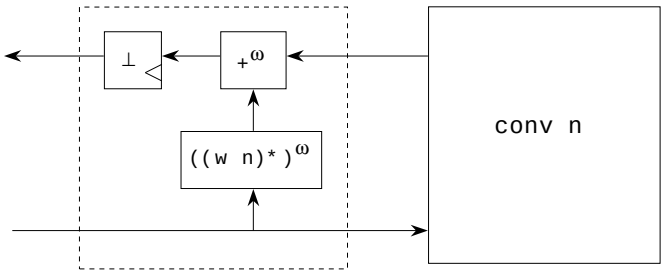
We now perform the induction step. For `t >= n+1` and `[e] = conv w (n+1) d` we obtain

```
    e t
=   {| specification of e |}
    sum [ w (n+1-i) * d (t-i) | i <- [1..n+1] ]
=   {| splitting the interval, definition of sum |}
    w n * d (t-1) + sum [ w (n+1-i) * d (t-i) | i <- [2..n+1] ]
=   {| index transformation |}
    w n * d (t-1) + sum [ w (n+1-(j+1)) * d (t-(j+1)) | j <- [1..n] ]
=   {| arithmetic |}
    w n * d (t-1) + sum [ w (n-j) * d (t-1-j) | j <- [1..n] ]
=   {| fold conv |}
    w n * d (t-1) + c (t-1)
    where [c] = conv w n d
```

In combinator form this reads

```
conv w (n+1)   =   (cell w n)  =|  (conv w n)
cell w k       =   (fork ||| id) |> (id ||| h)
h w k          =   (lift1 ((w k) *) ||| id) |>
                   (lift2 (+)) |> (undefined >-)
```

This recursive formation law for the basic convolver can be depicted as follows, where $\perp$ stands for `undefined`:

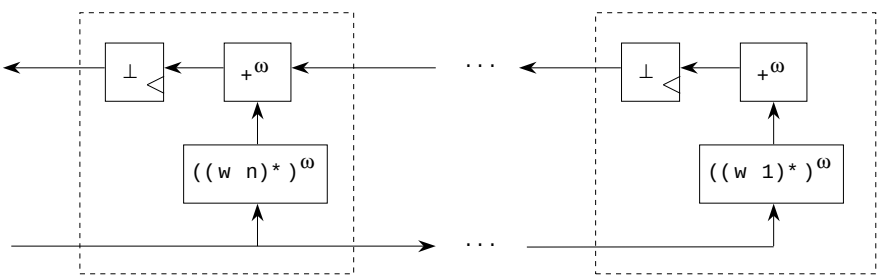


6.8.3 Unwinding the recursion

For fixed `n > 0` we obtain a regular design:

```
conv w n  =
  (foldr1 (=||=) [ cell w k | k <- [1..n] ]) =| cnst 0
```

After simplification of the rightmost cell this yields the design



6.8.4 Towards a Systolic Version

A circuit is **combinational** iff it uses only lifted operations and sequential or parallel composition. In a clocked circuit, the clock period is determined by the stabilisation time of the circuit which depends on its longest combinational path.

In systolic circuits one tries to minimise the clock period by making the combinational modules involved quite small. Then the clock period can be kept relatively short, namely it can be taken as the maximum of the stabilisation times of the combinational submodules involved. Since there is, however, no general rule for calling a combinational module "small" the precise definition of systolism avoids such a notion. Rather, a circuit is called **systolic** (cf. [39, 40, 21]) iff there is at least one delay element along every connection wire between any of its combinational modules. A related but somewhat different notion of systolism is used in the field of massively parallel systems; however, there no explicit delay elements are employed.

We want to obtain a systolic version of our convolver. Hence we have to introduce additional delay elements.

6.9 Speedup by Slowdown

The technique to introduce delays formally is slowdown (see e.g. [39, 40, 30]). The `k`-fold slowed down version of a circuit works on `k` interleaved streams. So each of these is processed at rate `k` slower than in the original circuit.

6.9.1 Interleaved Streams

To talk about the component streams of such a "multistream" we introduce

```
split k j d t  =  d (k*t + j)
```

So `split k j d` is the `j`-th of the `k` component streams where numbering starts with `0` again. Eg. `split 2 0 d` and `split 2 1 d` consist of the values in `d` at even and odd time points, respectively. Then `d` can be considered as an alternating interleaving of these. The interleaving of `k=4` streams may be depicted as follows:

The following properties of `split` are useful for proving the slowdown propagation rules below:

Lemma 6.4 *We have*

```
(x>-) |> split k 0  =  (split k (k-1)) |> (x>-)
(x>-) |> split k j  =  split k (j-1)            (0 < j < n)
```

To interleave `k` streams from a list `ss` we use

```
ileave k ss t  =  (ss !! (t `mod` k))(t `div` k)
```

Provided that `length ss >= k`, we have

```
split k j (ileave k ss)  =  ss !! j
```

A special case is the interleaving of `k` copies of the same stream:

```
rep k d =  ileave k (copy k d)
```

The above property yields

```
split k j (rep k d)  =  d
```

6.9.2 The Slowdown Function

Now the slowdown function is specified implicitly by

```
(slow k f ) |>  split k j   =   (split k j)  |> f
```

Here `f` is an arbitrary function on streams, not just a lifted unary operation. In particular, `f` may look at all the history of a stream. By this definition, `slow k f s` may be considered as splitting `s` into `k` substreams, processing these individually with `f` and interleaving the result streams back into one stream. An explicit version of `slow` is

```
slow k f s t  =  f (split k (t `mod` k) s) (t `div` k)
```

From the specification the following proof principle is evident:

Lemma 6.5 *If for a function* `h` *and all* `j` *in* `[1..k]` *we have*

```
h |> split k j  = (split k j) |> f
```

then `h = slow k f`*.*

6.9.3 Slowdown Algebra

The function `slow` distributes nicely through our circuit building operators:

Lemma 6.6 *We have*

```
slow k (x >-)       =   foldr (|>) id (copy k (x >-))
slow k (cnst x)     =   cnst x
slow k (f |>   g)   =   slow k f  |>    slow k g
slow k (f |||  g)   =   slow k f  |||   slow k g
slow k (f =||= g)   =   slow k f  =||=  slow k g
slow k (f =|   g)   =   slow k f  =|    slow k g
slow k (feed m f)   =   feed m (slow k f)
```

This means that the `k`-fold slowed down version of a circuit results by replacing each delay element by `k` identical delay elements. A further useful propagation law for `slow` is given by

Lemma 6.7 *Suppose that* `(x>-) |> f = f |> (y>-)`*. Then also*

```
(x>-) |> slow k f  = (slow k f) |> (y>-)
```

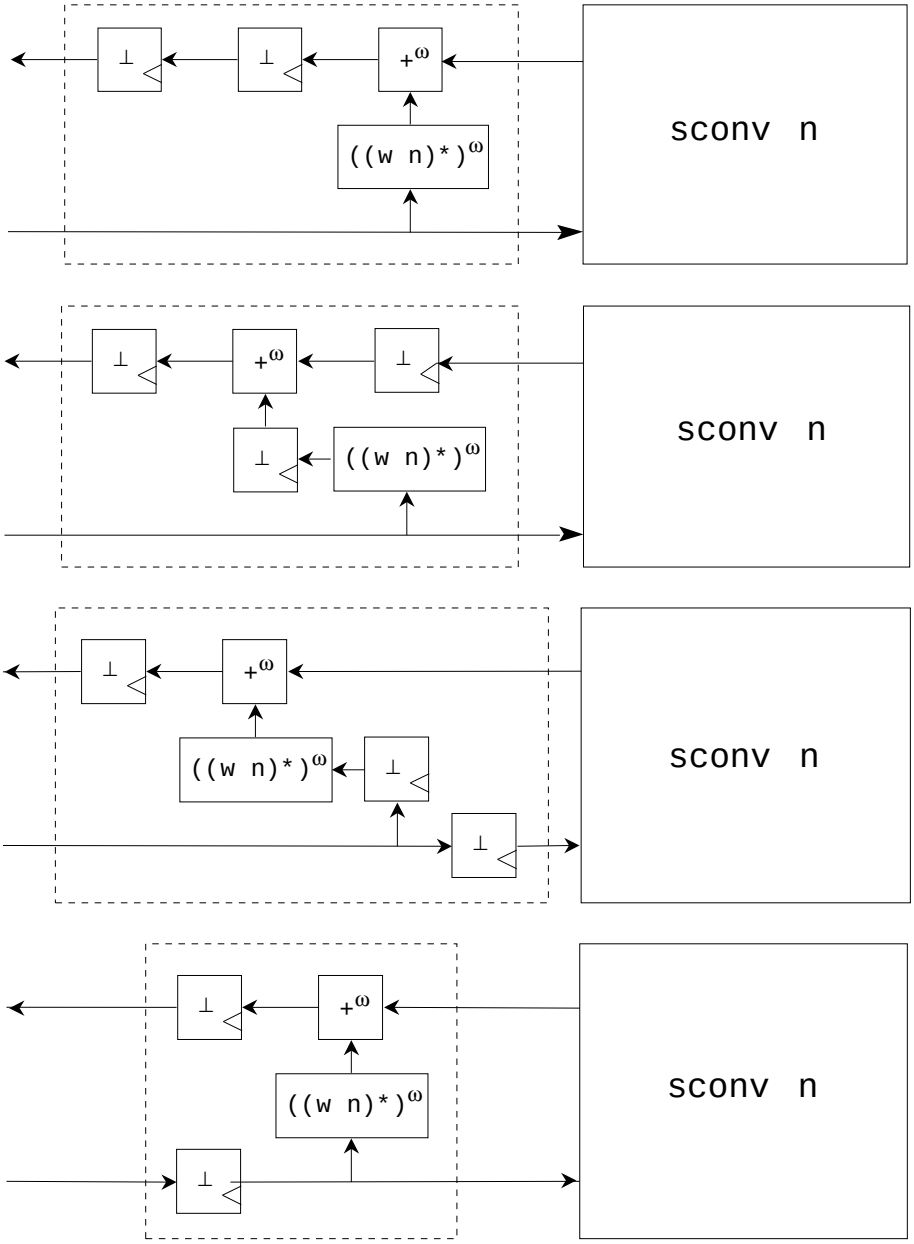
6.10 Example: A Systolic Convolver

Using `k`-fold slowdown we can interleave `k` streams or pad one stream with dummy elements by merging it with constant streams of dummies. The latter approach is usually taken in verification approaches to the systolic convolver: one uses slowdown by `2` and is interested only in the stream values at odd time points; at even time points eg. the value `0` is used.

We want to derive a systolic convolver by deductive design. We leave the decision whether to use proper interleaving or padding open; both can be achieved by suitable embeddings of the original `conv` function into the slowed down one defined by

```
sconv n  =  slow 2 (conv n)
```

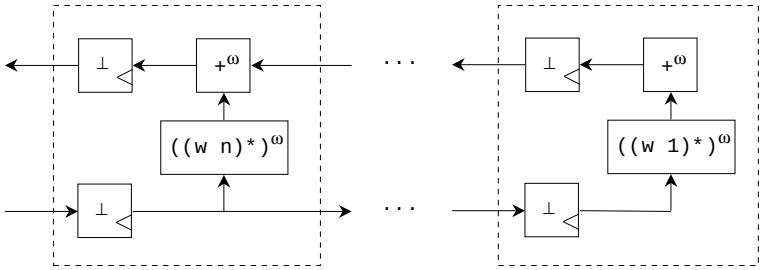
Now, employing the delay propagation rules, we push the second delay introduced by the slowdown through the various modules. We perform the derivation pictorially:



The step of pushing the delay through `sconv w n` is justified by Lemma 6.7. Unwinding the recursion again we obtain a regular systolic design:

```
sconv w n  =
   (foldr1 (=||=) [scell w k  |  k <- [1..n]]) =| cnst 0
scell w k  =   (undefined >-) |> (fork ||| id) |> (id ||| h)
h w k      =   (lift1 ((w k) *) ||| id) |>
               (lift2 (+)) |> (undefined >-)
```

This simplifies into



Of course, the techniques we have developed do not only apply to the convolver, but are of general interest for the derivation of systolic implementations of circuits. As a further case study, a systolic recogniser for regular expressions is developed in [56]. Moreover, a large part of the network algebra can be re-used for the specification and derivation of combinational circuits. For an account of this see again [55].

The derivations have been fairly short; moreover, the semantical basis is simple. So our approach compares favourably with verification approaches in this area (see eg. [43, 73]).

Acknowledgements Many helpful remarks on an earlier version of this paper were provided by L. Jenner and T. Ehm, the latter of whom also was of great help in integrating the pictures. Further valuable comments and suggestions were provided by R.M. Dijkstra and G. Ştefănescu.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman: The design and analysis of computer algorithms. Reading, Mass.: Addison Wesley 1974

[2] B. Alpern, F.B. Schneider: Defining liveness. Information Processing Letters **21**, 181–185 (1985)

[3] D.N. Arden: Delayed logic and finite state machines. In: Theory of computing machine design. Univ. of Michigan Press, Ann Arbor 1960, 1–35

[4] R.C Backhouse, P.J. de Bruin, G. Malcolm, E. Voermans, J. van der Woude: A relational theory of datatypes. Dept. of Mathematics and Computer Science, Eindhoven University of Technology. Available through
`http://www.win.tue.nl/cs/wp/papers/abstract.html`

[5] J.C.M. Baeten, W.P. Weijland: Process algebra. Cambridge Tracts in Theoretical Computer Science **18**. Cambridge: Cambridge University Press 1990

[6] H.P. Barendregt: Functional programming and lambda calculus. In: J. van Leeuwen (ed.): Handbook of theoretical computer science. Vol. B: Formal models and semantics. Amsterdam: Elsevier 1990, 321–363

[7] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner, P. Pepper: The Munich project CIP. Volume II: The program transformation system CIP-S. Lecture Notes in Computer Science **292**. Berlin: Springer 1987

[8] 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)

[9] U. Berger, W. Meixner, B. Möller: Calculating a garbage collector. In: M. Broy, M. Wirsing (Hrsg.): Methodik des Programmierens. Fakultät für Mathematik und Informatik der 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

[10] G. Birkhoff: Lattice theory, 3rd edition. American Mathematical Society Colloquium Publications, Vol. XXV. Providence, R.I.: AMS 1967

[11] J.D. Brock, W.B. Ackerman: Scenarios: a model of non-determinate computation. In: J. Diaz, I. Ramos (ed.): Formalization of programming concepts. Lecture Notes in Computer Science **107**. Berlin: Springer 1981, 252–259

[12] M. Broy: Specification and refinement of a buffer of length one. In: M. Broy (ed.): Deductive program design. NATO ASI Series, Series F: Computer and Systems Sciences **152**. Berlin: Springer 1996, 273–304

[13] M. Broy: Functional specification of time sensitive communicating systems. In: M. Broy (ed.): Programming and mathematical method. NATO ASI Series, Series F: Computer and Systems Sciences **88**. Berlin: Springer 1992, 325–367

[14] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T.F. Gritzner, R. Weber: The design of distributed systems — an introduction to FOCUS. Revised Version. Institut für Informatik der TU München, Report TUM-I9202-2 (SFB-Bericht Nr. 342/2-2/92 A), 1993

[15] T. Brunn, B. Möller, M. Russling: Layered Graph Traversals and Hamiltonian Path Problems – An Algebraic Approach. In: J. Jeuring (ed.): Mathematics of Program construction. Lecture Notes in Computer Science **1422**. Berlin: Springer 1998, 96–121

[16] K.M. Chandy, J. Misra: Parallel program design: a foundation. Reading, Mass.: Addison Wesley 1988

[17] J.H. Conway: Regular algebra and finite machines. London: Chapman and Hall 1971

[18] B.A. Davey, H.A. Priestley: Introduction to lattices and order. Cambridge: Cambridge University Press 1990

[19] M. Davis: Infinitary games of perfect information. In: M. Dresher, L.S. Shapley, A.W. Tucker (eds.): Advances in game theory. Princeton, N.J.: Princeton University Press 1964, 89–101

[20] F. Dederichs, R. Weber: Safety and liveness from a methodological point of view. Information Processing Letters **36**, 25–30 (1990)

[21] S. Even, A. Litman: On the capabilities of systolic systems. Math. Sytems Theory 27, 3-28 (1994)

[22] N.D. Gautam: The validity of equations of complex algebras. Arch. Math. Logik Grundlag. **3**, 117–124 (1957)

[23] S. Ginsburg: The mathematical theory of context-free languages. New York: McGraw-Hill 1966

[24] M.J. Gordon: Why higher-order logic is a good formalism for specifying and verifying hardware. In: G.J. Milne, P.A. Subrahmanyam (eds.): Formal aspects of VLSI design. North-Holland 1986

[25] HAWK — Specifying, Verifying, and Simulating Microprocessors. Web page under `http://www.cse.ogi.edu/PacSoft/Hawk/`

[26] C.A.R. Hoare: Communicating sequential processes. London: Prentice Hall 1985

[27] C.A.R. Hoare: Conjunction and concurrency. PARBASE 90, 1990

[28] G. Hotz, B. Becker, R. Kolla, P. Molitor: Ein logisch-topologischer Kalkül zur Konstruktion integrierter Schaltungen. Informatik — Forschung und Entwicklung 1, 28-47 and 72-82 (1986)

[29] IFIP WG 10.5: Benchmark circuits for hardware verification. Available through `http://goethe.ira.uka.de/hvg/benchmarks.html`

[30] G. Jones, M. Sheeran: Circuit design in Ruby. In: J. Staunstrup (ed.): Formal methods for VLSI design. Elsevier 1990, 13-70

[31] B. Jonsson: A fully abstract trace model for dataflow and asynchronous networks. Distributed Computing **7**, 197–212 (1994)

[32] G. Kahn: The semantics of a simple language for parallel processing. In: J.L. Rosenfeld (ed.): Information Processing 74. Proc. IFIP Congress 1974. Amsterdam: North-Holland 1974, 471–475

[33] 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

[34] S.C. Kleene: Introduction to metamathematics. New York: van Nostrand 1952

[35] B. Knaster: Un théorème sur les fonctions d'ensembles. Ann. Soc. Pol. Math. **6**, 133–134 (1928)

[36] J.N. Kok: A fully abstract semantics for data flow nets. In: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (eds.): PARLE, Parallel languages and architectures Europe, Volume I. Lecture Notes in Computer Science **259**. Berlin: Springer 1987, 351–368

[37] F. Kröger: Temporal logic of programs. EATCS Monographs on Theoretical Computer Science **8**. Berlin: Springer 1987

[38] L. Lamport: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **SE-3**, 125–143 (1977)

[39] C.E. Leiserson, J.B. Saxe: Optimizing synchronous systems. J. VLSI and Computer Systems 1, 41-68 (1983)

[40] C.E. Leiserson, J.B. Saxe: Retiming synchronous circuitry. Algorithmica 6, 5-35 (1991)

[41] P. Lescanne: Modèles non déterministes de types abstraits. R.A.I.R.O. Informatique théorique **16**, 225–244 (1982)

[42] G. Markowsky: Chain-complete posets and directed sets with applications. Algebra Universalis **6**, 53-68 (1976)

[43] K. Meinke, J. Steggles: Specification and verification in higher order algebra: a case study of convolution. In: J. Heering, K. Meinke, B. Möller, T. Nipkow (eds.): Higher order algebra, logic and term rewriting. Lecture Notes in Computer Science **816**. Berlin: Springer 1994, 189–222

[44] Mathematics of Program Construction Group: Fixed-point calculus. Information Processing Letters **53**, 131-136 (1995)

[45] R. Milner: Communication and concurrency. London: Prentice Hall 1989

[46] B. Möller: Derivation of graph and pointer algorithms. In: B. Möller, H.A. Partsch, S.A. Schuman (eds.): Formal program development. Lecture Notes in Computer Science **755**. Berlin: Springer 1993, 123–160

[47] B. Möller: Towards pointer algebra. Institut für Mathematik der Universität Augsburg, Report No. 279, 1993. Revised version in Science of Computer Programming **21**, 57–90 (1993)

[48] B. Möller: Algebraic calculation of graph and sorting algorithms. In: D. Bjørner, M. Broy, I.V. Pottosin (eds.): Formal methods in Programming and their Applications. Lecture Notes in Computer Science **735**. Berlin: Springer 1993, 394–413

[49] B. Möller: Ideal streams. In: E.-R. Olderog (ed.): Programming concepts, methods and calculi. IFIP Transactions A-56. Amsterdam: North-Holland 1994, 39–58

[50] B. Möller: Refining ideal behaviours. Institut für Mathematik der Universität Augsburg, Report Nr. 345, 1995

[51] B. Möller: Assertions and recursions. In: G. Dowek, J. Heering, K. Meinke, B. Möller (eds.): Higher order algebra, logic and term rewriting. Second International Workshop, Paderborn, Sept. 21-22, 1995. Lecture Notes in Computer Science **1074**. Berlin: Springer 1996, 163-184

[52] B. Möller: Temporal operators on partial orders. In: U. Berger, K.-H. Niggl, B. Reus (eds.): Proc. 3rd Domain Workshop, Munich, 29–31 May 1997. Institut f'ur Informatik, LMU Munich, Technical Report 9712, December 1997, 49–58

[53] B. Möller: Modal and Temporal Operators on Partial Orders. In: R. Berghammer, F. Simon (eds.): Programming languages and fundamentals of programming. Institut für Informatik und Praktische Mathematik, Universität Kiel, Report No. 9717, November 1997, 1–11. Extended version: Institut für Informatik der Universität Augsburg, Report Nr. 1997-2, November 1997

[54] B. Möller: Ideal stream algebra. In: B. Möller, J.V. Tucker (eds.): Prospects for hardware foundations. Lecture Notes in Computer Science **1546**. Berlin: Springer 1998, 69–116

[55] B. Möller: Deductive hardware design: a functional approach. In: B. Möller, J.V. Tucker (eds.): Prospects for hardware foundations. Lecture Notes in Computer Science **1546**. Berlin: Springer 1998, 421–468

[56] B. Möller: An algebraic approach to systolic circuits. Institut für Informatik der Universität Augsburg, Report 1998-01, January 1998. Also in: B. Möller, M. Sheeran (eds.): Proceedings of the Workshop FTH '98 — Formal Techniques for Hardware and Hardware-Like Systems, Marstrand, June 19, 1998. Chalmers Institute of Technology, Göteborg, 1998

[57] B. Möller: Calculating with acyclic and cyclic lists. Special issue on Relational Methods in Computer Science. Information Sciences — An International Journal (to appear)

[58] 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. Revised version in Science of Computer Programming **22**, 157–180 (1994)

[59] M. Nivat: Behaviors of processes and synchronized systems of processes. In: M. Broy, G. Schmidt (eds.): Theoretical foundations of programming methodology. Dordrecht: Reidel 1982, 473–551

[60] J. O'Donnell: From transistors to computer architecture: teaching functional circuit specification in Hydra. In: P.H. Hartel, R. Plasmeijer (eds.): Functional programming languages in education. Lecture Notes in Computer Science **1022**. Berlin: Springer 1995, 195–214

[61] E.-R. Olderog: Nets, terms and formulas. Cambridge: Cambridge University Press 1991

[62] H.A. Partsch: Specification and transformation of programs — A formal approach to software development. Berlin: Springer 1990

[63] D. Park: On the semantics of fair parallelism. In D. Bjørner (ed.): Abstract software specifications. Lecture Notes in Computer Science **86**. Berlin: Springer 1980, 504–526

[64] D.L. Rhodes: Analog modeling using MHDL. In: J.-M. Bergé (ed.): Current issues in electronic modeling, Issue #2 "Modeling in analog design. Kluwer 1995

[65] M. Russling: Deriving general schemes for classes of graph algorithms. Augsburger Mathematisch-Naturwissenschaftliche Schriften, Band 13 (1996).

[66] G. Schmidt, T. Ströhlein: Relations and graphs. Discrete Mathematics for Computer Scientists. EATCS Monographs on Theoretical Computer Science. Berlin: Springer 1993

[67] W. Schulte, T. Vullinghs: ULTRA: A Useful and Lean Transformation System. Fakultät für Informatik, Technical Report. Universität Ulm (to appear)

[68] S. Singh: Lava. Web page under
`http://www.dcs.gla.ac.uk/ satnam/lava/main.html`

[69] System level design language. Web page under
`http://www.intermetrics.com/SLDL/`

[70] M.B. Smyth: Topology. In: S. Abramsky, D.M. Gabbay, T.S.E. Maibaum (eds.): Handbook of logic in computer science. Vol. 1, Background: Mathematical structures. Oxford: Clarendon Press 1992, 641–761

[71] L. Staiger: Research in the theory of ω-languages. J. Inf. Process. Cybern. EIK **23**, 415–439 (1987)

[72] G. Ştefănescu: Algebra of flownomials. Institut für Informatik der TU München, Report TUM-I9437, 1994

[73] J. Steggles: Parameterised higher-order algebraic specifications. In: M. Hanus, J. Heering, K. Meinke (eds.): Algebraic and Logic Programming. Lecture Notes in Computer Science **1298**. Berlin: Springer 1997,76–97

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

[75] A. Tarski: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**, 285–309 (1955)

[76] W. Thomas: Automata on infinite objects. In: J. van Leeuwen (ed.): Handbook of theoretical computer science. Vol. B: Formal models and semantics. Amsterdam: Elsevier 1990, 133–191

[77] W. Thomas: Languages, automata and logic. In: G. Rozenberg, A. Salomaa (eds.): Handbook of formal languages. Vol. 3: Beyond words. Berlin: Springer 1997, 389–455

[78] J. Zwiers: Compositionality, concurrency and partial correctness. Lecture Notes in Computer Science **321**. Berlin: Springer 1989

Appendix: Essential Constructs of *Haskell*

Basic Types and Functions

For those not familiar with *Haskell*, we briefly repeat its essential elements. Basic types are `Int` for the integers and `Bool` for the Booleans with elements `True` and `False`. Exponentiation is written in the form `x^n`. The operations of conjunction and disjunction are denoted by `&&` and `||`, resp. These are the semi-strict versions evaluating their arguments from left to right, ie. satisfying

```
x && y  =  if x then y    else False
x || y  =  if x then True else y
```

The type of functions taking elements of type `a` as arguments and producing elements of type `b` as results is `a -> b`. The fact that a function `f` has this type is expressed as `f :: a -> b`.

Function application is denoted by juxtaposing function and argument, separated by at least one blank, in the form `f x`. Functions of several arguments are mostly used in curried form `f x1 x2 ... xn`. In this case `f` has the higher-order type `f :: t1 -> (t2 -> ... (tn -> t) ...)` or, abbreviated, `f :: t1 -> t2 -> ... tn -> t` (the arrow `->` associates to the right, whereas function application associates to the left).

Functions are defined by equations of the form `f x = E` or as (anonymous) lambda abstractions. Instead of $\lambda x.E$ one uses the notation `\x -> E`.

A two-place function `f :: a -> b -> c` may also be used as an infix operator in the form `x ‘f‘ y`; this is equivalent to the usual application `f x y`. Eg. instead of `div x y` one may also write `x ‘div‘ y`. To formulate a number of expressions and properties in a more readable way we use a small notational extension of this: we also use larger expressions (that do not contain the backquote) between backquotes as infix operators. Eg. for

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

we may then write `xs ‘zipWith (+)‘ ys` for the componentwise addition of lists `xs` and `ys`.

For a binary operator `?`, by supplying only one of its arguments one obtains a **residual function** or **operator section** of the form

```
(x ?)  =  \y -> x ? y  or  (? y) = \x -> x ? y
```

Case Distinction and Assertions

Haskell offers several possibilities for doing case distinctions. One is the usual `if then else` construct. To avoid cascades of ifs, a function may also be defined in a style similar to the one used in mathematics. The notation is

```
f x
| C1       =   E1
...
| Cn       =   En
```

The result is the value of the first expression `Ei` for which the corresponding `Ci` evaluates to `True`. If there is none, the result is undefined.

We also use this to make functions intentionally partial in order to enforce assertions about their parameters (see Section refsc:assert and [51]).

To avoid partiality one can use the predefined constant `otherwise = True` and add a final clause

```
| otherwise  =  En+1
```

Yet another way of case distinction is provided by defining a function through argument patterns. Several equations indicate what a function does on inputs that have certain shapes. The equations are tried in textual order; if no pattern matches the current argument, the function is again undefined at that point.

Example. By the equations

```
f 0 = 5
f 1 = 7
```

the function `f :: Int -> Int` is defined only for argument values `0` and `1`.

Lists

The type of lists of elements of type `a` is denoted by `[a]`. The list consisting of elements `x1,...,xn` is written as `[x1,...,xn]`; in particular, `[]` is the empty list. Concatenation is denoted by `++`. Prefixing an element to a list is denoted by the colon operator:

```
x:xs = [x] ++ xs
```

The function `length` returns the length of a list. The `i`th element of list `xs` is selected by the expression `xs!!i` (where numbering starts with `0`).

A list may be split into two parts using the functions

```
take, drop :: [a] -> Int -> [a]
```

For non-negative integer `k` the list `take k xs` consists of the first `k` elements of `xs` if `k <= length xs` and of all of `xs` if `k > length xs`. For negative `k` the expression `take k xs` is undefined. The list `drop k xs` results by removing `take k xs` from the front of `xs`. Hence one always has

```
take k xs ++ drop k xs = xs
```

A very useful specification feature is list comprehension in the form

```
[ f x | x <- L, px]
```

where `L` is a list expression, `f` some function on the list elements and `p` a Boolean function. The symbol `<-` may be viewed as a leftward arrow and pronounced as "drawn from" or as a form of element sign. In this latter view, the expression is the list analogue of the usual set comprehension $\{f\,x \mid x \in S, p\,x\}$. The meaning of the list comprehension expression `[ f x | x <- L, p x]` is again a list, constructed as follows:

- The elements of list `L` are scanned in left-to-right order.
- On each such element `x` the test `p` is performed.
- If `p x = True`, `f x` is put into the result list.
- Otherwise, `x` is ignored.

The list `[m,m+1,...,n]` of integers may be denoted by the shorthand `[m..n]`. The right bound `n` may be omitted; then the expression denotes the infinite list `[m, m+1, ... ]`.

A useful operation on non-empty lists is the folding of their elements using a binary operator `f :: :`

```
foldr1 f [x1,...,xn]  =  f x1 (f x2 ... (f  xn-1 xn)...)
```

Eg. `foldr1 (+) s` computes the sum of all elements of `s`. The function `foldr1` itself has the type `(a -> a -> a) -> [a] -> a`.

A variant of `foldr1` that also copes with empty lists is `foldr`; it uses an additional argument `e` that specifies the value for empty lists. The defining equations read

```
foldr f e []  =  e
foldr f e [x] ++ xs = f x (foldr f e xs)
```

Based on `foldr` one can define a universal quantifier over lists. For a predicate `p :: a -> Bool` one has

```
all p xs  =  foldr (&&) True [p x | x <- xs ]
```

So `all p xs` yields `True` iff `p x` yields `True` for all `x` in `xs`.