Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Logic and Algebraic Programming 66 (2006) 71–111
THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING
www.elsevier.com/locate/jlap

Regular algebra applied to language problems

Roland Backhouse

School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, United Kingdom

Abstract

Many functions on context-free languages can be expressed in the form of the least fixed point of a function whose definition mimics the grammar of the given language. Examples include the function returning the length of the shortest word in a language, and the function returning the smallest number of edit operations required to transform a given word into a word in a language.

This paper presents the basic theory that explains when a function on a context-free language can be defined in this way. It is shown how the theory can be applied in a methodology for programming the evaluation of such functions.

Specific results include a novel definition of a regular algebra focusing on the existence of so-called "factors", and several constructions of non-trivial regular algebras. Several challenging problems are given as examples, some of which are old and some of which are new.

Keywords: Galois connection; Fixed point; Programming methodology; Problem generalisation; Factor theory; Factor matrix; Path-finding; Language inclusion

1. Introduction

A common technique for solving programming problems, particularly the more challenging ones, is to express the problem in terms of solving a system of so-called "simultaneous" equations (a collection of equations in a number of unknowns that are often mutually recursive). Having done so, a number of techniques can be used for solving the equations, ranging from simple iterative techniques to more sophisticated but more specialised elimination techniques.

A relatively straightforward and well-known example is the problem of finding shortest paths through a graph. The distances to any one node from the nodes in a graph can be expressed as a set of simultaneous equations, which equations can be solved using, for

E-mail address: rcb@cs.nott.ac.uk

1567-8326/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2005.04.008

example, Dijkstra's shortest path algorithm [10]. Another, similar but much less straightforward, problem is that of finding the *edit distance* between a word and a language—the minimum number of edit operations required to edit the given word into a word in the given language. In the case that the language is defined by a context-free grammar, the problem can be solved by constructing a system of equations in the edit distances between each segment of the word and each non-terminal in the grammar. This set of equations can then be solved using a simple iterative technique or Knuth's generalisation [23] of Dijkstra's shortest path algorithm.

A stumbling block for the use of simultaneous equations is that there is often a very big leap from a problem's specification to the construction of the system of simultaneous equations; the justification for the leap almost invariably involves a *post hoc* verification of the construction. Thus, whereas methods for solving the equations, once constructed, are well known and understood, the process of constructing the equations is not. The example of edit distances just given is a good example. Indeed, the theory of context-free languages offers many examples—determining whether the language generated by a given grammar is empty or not, determining the length of a shortest word in a context-free language, determining the *FIRST* set of each of the non-terminals in a grammar, and so on.

In this paper, we present a general theory, which expresses when the solution to a problem can be expressed as solving a system of simultaneous equations. We give several examples of the theorem together with several non-examples (that is, examples where the theorem is not directly applicable). The non-examples serve two functions. They highlight the gap between specification and simultaneous equations—we show in several cases how a small change in the specification leads to a breakdown in the solution by simultaneous equations—and they inform the development of a methodology for the construction of the equations.

There are three elements to the mathematical theory underlying this paper—the theory of Galois connections, fixed-point calculus and regular algebra. A brief introduction to Galois connections and fixed-point calculus is given in Section 3. Section 3 concludes with the so-called *fusion* theorem. This theorem has been used earlier by Jeuring and Swierstra [20,21] in the case of some of the simpler examples discussed in this paper. Proofs are not given in this initial section because the results are standard.

The novel contribution begins in Section 4. Here, we propose a novel definition of a regular algebra; this definition is chosen so that we can encompass, within one theorem, many examples of programming problems, including ones involving computations on context-free grammars as well as the standard examples on finite graphs. Several theorems are presented showing how complex regular algebras can be constructed from simpler ones.

Section 5 introduces the notion of a regular homomorphism. In combination with the fusion theorem of Section 3, this gives a fundamental *fusion* theorem showing how a regular homomorphism maps one interpretation of a context-free grammar into another.

The fusion theorem of Section 5 imposes quite stringent preconditions; but, these conditions are often met automatically by the way that the components are constructed. Section 6 is about circumstances in which this is the case. The theorems in this section show how to extend "measures" on the elements of a monoid to regular homomorphisms. An important theorem in this section is Theorem 6.2, which gives simple conditions guaranteeing that the image of a measure defines a regular algebra. The section is concluded by several non-trivial examples of measures on languages.

Section 7 discusses a novel application. Often, particularly in optimisation problems, it is sufficient to know a "bound" on a solution to the problem. For example, it may be

sufficient to know that a journey can be completed in at most two hours rather than knowing specific details of a shortest route. A more complex example is the language-inclusion problem: given a context-free grammar G and a language L, is the language generated by G a subset of the language L. This problem occurs in program analysis. The solution to the language-inclusion problem, in the case that L is regular, is due to De Moor [30,35]. Our contribution is to generalise the problem to the "bound" problem in an arbitrary regular algebra, and to show how it fits into the framework we have developed. In so doing, we remove the restriction on L being regular. We show that, for arbitrary language L, it is possible to express the inclusion problem as a fixed-point computation; however, this leaves open the problem of developing fixed-point algorithms that are applicable to wider classes of languages.

The paper is concluded by a discussion of the relevance of the results to programming methodology.

Because of length constraints, details of a number of calculations have been omitted. A version of the paper complete with all proofs is available from `www.cs.nott.ac.uk/~rcb/papers`.

2. Introductory examples

It is as well to begin with some introductory examples. Some of these have been alluded to above; we return to all of them in the course of the theory development below.

For concreteness, let us consider the following context-free grammar:

$$S ::= aSS \mid \varepsilon. \tag{1}$$

The language defined by this grammar is a least fixed point. Specifically[1],

$$S = \langle \mu_{\subseteq} X :: \{a\} \cdot X \cdot X \cup \{\varepsilon\} \rangle. \tag{2}$$

Now, if we want to determine whether S is empty, we solve the equation

$$S = \phi \equiv (\{a\} = \phi \vee S = \phi \vee S = \phi) \wedge \{\varepsilon\} = \phi.$$

More precisely,

$$(S = \phi) = \langle \mu_{\Leftarrow} X :: (\{a\} = \phi \vee X \vee X) \wedge \{\varepsilon\} = \phi \rangle. \tag{3}$$

That is, we determine a least fixed point, where "least" is defined by the $\Leftarrow$ ordering on booleans (that is, true is "smaller than" false, since true $\Leftarrow$ false). This, of course, is the greatest fixed point with respect to the $\Rightarrow$ ordering on booleans (which is how booleans are normally ordered).

We may also wish to compute the length of a shortest word in S, which we denote by $\#S$. This is done by solving the equation[2]

[1] We use the notation $\mu_{\preceq} f$ for the least fixed point of function f, where f is a monotonic endofunction on a set ordered by the relation $\preceq$. The concatenation of languages S and T is denoted by $S \cdot T$ and ε denotes the empty word. The notation for functions and function application follows the recommendations of Dijkstra and Scholten [13]. Angle brackets delimit the scope of bound variables, so that $\langle x : R : E \rangle$ denotes the function that maps a value x in the range given by predicate R to the value denoted by expression E, and $\langle \mu_{\preceq} x : R : E \rangle$ denotes its least fixed point with respect to the ordering $\preceq$ on its range. Later, function application is denoted by an infix dot. By common convention, function application is left-associative; that is, $f.x.y$ equals $(f.x).y$.

[2] $x \downarrow y$ denotes the minimum of x and y; $x \uparrow y$ denotes their maximum.

$$\#S = (\#\{a\} + \#S + \#S) \downarrow \#\{\varepsilon\}.$$

To be precise,

$$\#S = \langle \mu_{\geq} X :: (\#\{a\} + X + X) \downarrow \#\{\varepsilon\} \rangle. \tag{4}$$

That is, we determine a "least" fixed point, where "least" is defined by the $\geq$ ordering on numbers—so the "least" solution is what we would normally call the "greatest" solution.

Another test we might do is to determine whether S is "nullable", that is, whether the empty word is an element of S. This test is accomplished by solving the equation

$$(\varepsilon \in S) = ((\varepsilon \in \{a\} \wedge \varepsilon \in S \wedge \varepsilon \in S) \vee \varepsilon \in \{\varepsilon\}).$$

To be precise,

$$(\varepsilon \in S) = \langle \mu_{\Rightarrow} X :: (\varepsilon \in \{a\} \wedge X \wedge X) \vee \varepsilon \in \{\varepsilon\} \rangle. \tag{5}$$

That is, $\varepsilon \in S$ is a least fixed point, where, this time, the boolean values are ordered by the $\Rightarrow$ relation (so true is "larger than" false).

In each of these examples, the information we require is obtained by determining the least fixed point of a function that is derived directly from the grammar defining S. We invite the reader to compare (2)–(5).

Although the grammar we have chosen for illustration is very simple, extending the process of constructing the equations to more complicated grammars is not problematic. The only difference is that, when the grammar has more than one non-terminal, a system of equations has to be solved rather than a single equation, as here.

These are all standard textbook examples. But, the process of constructing the equations is not as straightforward as the literature would suggest. One complication, that is often overlooked, is knowing which ordering relation to use when solving the equations. Note how the tests $S = \phi$ and $\varepsilon \in S$ require converse orderings of the booleans. Also, small variations on the above problems do not appear to be solvable in this way. For example, if we replace "ε" by "aa" in (5), we do not get a valid equation, since

$$aa \in S \neq ((aa \in \{a\} \wedge aa \in S \wedge aa \in S) \vee aa \in \{\varepsilon\}).$$

(The left side is true whereas the right side is false.) Indeed, the general parsing problem—given a grammar G with sentence symbol S and a word x, determine $x \in S$—is a difficult problem. Yet, the special case where $x = \varepsilon$ is, somehow, easy! It is the goal of this paper to show when and how evaluating a function of a language can be mapped into solving a fixed-point equation.

An important element of this process is problem generalisation. The following example has been designed as an illustration. Suppose it is required to determine whether all words in the language generated by a given grammar G are of even length. We might try to solve the problem by constructing a system of equations in the boolean values $e.X$, where X is a non-terminal symbol of the grammar and $e.X$ means that all words in the language generated by X are of even length. This, however, is doomed to failure—for the obvious reason that all words in the concatenation $S{\cdot}T$ of two languages S and T have even length is not the same as all words in S have even length and all words in T have even length. It is, however, the case that all words in the concatenation $S{\cdot}T$ of two languages S and T have even length if one of the languages is empty; or, all words in S have even length and all words in T have even length; or, all words in S have odd length and all words in T have odd length. This suggests the strategy of first eliminating all non-terminals that generate the empty language, and then solving the generalised problem of computing e and o, where

$e.X$ is true exactly when all words in the language X have even length, and $o.X$ is true exactly when all words in the language X have odd length.

For our example grammar above, this would result in constructing the equation in the pair of booleans o and e:

$$(o.S, e.S) = (o.\{a\}, e.\{a\}) \times (o.S, e.S) \times (o.S, e.S) + (o.\{\varepsilon\}, e.\{\varepsilon\}),$$

where, for all booleans b, c, d and e,

$$(b, c) \times (d, e) = ((b \wedge e) \vee (c \wedge d)\,,\, (b \wedge d) \vee (c \wedge e))$$

and

$$(b, c) + (d, e) = (b \wedge c\,,\, d \wedge e).$$

Whether all words in the language generated by S have even length is then the second component of the greatest solution of this fixed-point equation, where pairs are ordered componentwise by implication.

In Section 7, we see how the construction of this equation is predicted by our theory.

3. Basic mathematics

This section summarises the mathematical theory needed to understand the later sections. References are supplied for further reading. We assume familiarity with the notions of a partial ordering and complete lattice. (See, for example, [16] for an extensive account.)

3.1. Galois connections

The concept of a Galois connection was introduced by Oystein Ore in 1944 [32]. Since Ore's introduction of the general notion, Galois connections have been used in many contexts, although often without specific reference. Examples include [25,19,9,17,14]. Since the 1980s, however, the notion has become part of the everyday vocabulary of many computing scientists, and its use is becoming more explicit. In the field of abstract interpretation, Cousot and Cousot [7,8] have done much to make the notion widely known. Other examples are [31,27,12,18,37,4]. Several equivalent definitions can be given. The following was first introduced in [34].

Definition 3.1 (*Galois connection*). Suppose $\mathcal{A} = (A, \sqsubseteq)$ and $\mathcal{B} = (B, \preceq)$ are partially ordered sets, and suppose $F \in A \leftarrow B$ and $G \in B \leftarrow A$. Then (F, G) *is a Galois connection between* $\mathcal{A}$ *and* $\mathcal{B}$ when, for all $x \in B$ and $y \in A$,

$$F.x \sqsubseteq y \equiv x \preceq G.y.$$

We refer to F as the *lower adjoint* and to G as the *upper adjoint*.

Many examples of Galois connections exist. Inverse functions (for example the exponential and logarithm functions) are special cases, the sets $\mathcal{A}$ and $\mathcal{B}$ being ordered by equality. More interestingly, negation of boolean values is both the lower and upper adjoint in a Galois connection between the booleans ordered by "if" and the booleans ordered by "only if". Also, for each boolean b, the functions $(b\wedge)$ and $(b \Rightarrow)$ are the lower and upper adjoints, respectively, in a Galois connection on the booleans ordered by "only if".

Other examples are the definitions of the floor and ceiling functions, the maximum and minimum operators on finite sets of numbers, weakest liberal preconditions and weakest prespecifications.

Suppose $(A, \sqsubseteq)$ and $(B, \preceq)$ are partially ordered sets and $f \in A \leftarrow B$ is a monotonic function. Then, a *supremum* of f is a solution of the equation:

$$x :: \langle \forall a :: x \sqsubseteq a \equiv \langle \forall b :: f.b \sqsubseteq a \rangle \rangle. \tag{6}$$

Eq. (6) need not have a solution. However, by definition, $(A, \sqsubseteq)$ is a *complete lattice* if, for all so-called *shape* posets $(B, \preceq)$ and all monotonic functions $f \in A \leftarrow B$, (6) has a solution. If it does, for a given f, we denote its solution by Σf. That is, in a complete lattice,

$$\langle \forall a :: \Sigma f \sqsubseteq a \equiv \langle \forall b :: f.b \sqsubseteq a \rangle \rangle. \tag{7}$$

A particular case is when B is a two-element set. The supremum operator in this case is sometimes called "addition", and is denoted by an infix "+" symbol. Formally, using x and y to denote the two values of f, for all a,

$$x + y \sqsubseteq a \equiv x \sqsubseteq a \wedge y \sqsubseteq a. \tag{8}$$

Dually, the *infimum* of f, assuming it exists, is denoted by Πf and satisfies

$$\langle \forall a :: a \sqsubseteq \Pi f \equiv \langle \forall b :: a \sqsubseteq f.b \rangle \rangle. \tag{9}$$

We will not need to denote the binary infimum operator.

It is well known that the existence of all infima in a poset is guaranteed by the existence of all suprema in the poset, and vice-versa. So the definition of a complete lattice can be expressed in terms of the existence of all infima or the existence of all suprema.

Definition 3.2. Suppose $(A, \sqsubseteq)$ and $(B, \preceq)$ are complete lattices. Function $f \in A \leftarrow B$ is said to be *sup-preserving* if, for all posets C and all functions $g \in B \leftarrow C$,

$$f.(\Sigma g) = \Sigma(f \circ g).$$

($f \circ g$ denotes the composition of f and g, defined by $(f \circ g).x = f.(g.x)$.)

In many applications, knowing that a function is a lower adjoint in a Galois connection is all that is needed, and the specific definition of its upper adjoint is not relevant. In such circumstances, the following existence theorem is often used to determine that a function is indeed a lower adjoint.

Theorem 3.3 (Fundamental theorem). *Suppose that $\mathcal{B}$ is a poset and $\mathcal{A}$ is a complete poset. Then a monotonic function $F \in \mathcal{A} \leftarrow \mathcal{B}$ is a lower adjoint in a Galois connection equivales F is sup-preserving.*

An example may help the reader to relate Theorem 3.3 to their own understanding. Recall that, for all booleans p and q,

$$(\neg p \Rightarrow q) = (p \Leftarrow \neg q).$$

This is a Galois connection. Now, in order to instantiate Theorem 3.3 we need to be very clear about the partially ordered sets involved: instantiate $\mathcal{A}$ to the booleans ordered by implication and $\mathcal{B}$ to the booleans ordered by follows-from. Observe that the supremum operator in $\mathcal{A}$ is existential quantification, and in $\mathcal{B}$ is universal quantification. Thus by application of the theorem:

$$\neg\langle\forall x :: p.x\rangle = \langle\exists x :: \neg(p.x)\rangle.$$

The final theorem on Galois connections is described by Lambek and Scott [29] as "the most interesting consequence of a Galois correspondence". As such, it deserves a name. The name "unity-of-opposites theorem" used here was suggested to us by Lambek's discussion of the theorem in [26].

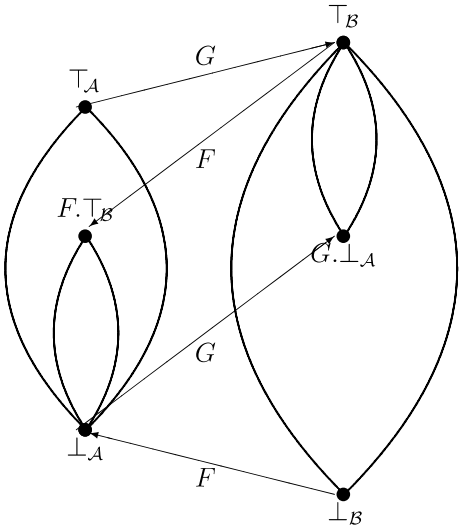
Theorem 3.4 (Unity of opposites). *Suppose $F \in \mathcal{A} \leftarrow \mathcal{B}$ and $G \in \mathcal{B} \leftarrow \mathcal{A}$ are Galois-connected functions, F being the lower adjoint and G being the upper adjoint. Then $F.\mathcal{B}$ and $G.\mathcal{A}$ are isomorphic posets. In particular,*

$$F \circ G \circ F = F,$$
$$G \circ F \circ G = G.$$

Moreover, if one of $\mathcal{A}$ or $\mathcal{B}$ is $\mathcal{C}$-complete, for some shape poset $\mathcal{C}$, then $F.\mathcal{B}$ and $G.\mathcal{A}$ are also $\mathcal{C}$-complete. Assuming that $\mathcal{B}$ is $\mathcal{C}$-complete, the supremum and infimum operators are given by

$$\Pi_{G.\mathcal{A}}.f = \Pi_{\mathcal{B}}.f,$$
$$\Sigma_{G.\mathcal{A}}.f = G.(F.(\Sigma_{\mathcal{B}}.f)),$$
$$\Pi_{F.\mathcal{B}}.f = F.(\Pi_{\mathcal{B}}.(G \circ f)),$$
$$\Sigma_{F.\mathcal{B}}.f = F.(\Sigma_{\mathcal{B}}.(G \circ f)).$$

Picturing the posets $\mathcal{A}$ and $\mathcal{B}$ as sets in which larger elements are above smaller elements, the unity-of-opposites theorem is itself summarised in the following diagram:



The two larger lenses picture the sets $\mathcal{A}$ (on the left) and $\mathcal{B}$ (on the right); the bottom-left lens pictures $F.\mathcal{B}$ and the top-right lens $G.\mathcal{A}$. The latter two sets are pictured as having the same size because they are isomorphic, whereas $\mathcal{A}$ and $\mathcal{B}$ are pictured as having different size because they will not be isomorphic in general. Note that F maps the least element of $\mathcal{B}$ (denoted $\bot_{\mathcal{B}}$ in the diagram) to the least element of $\mathcal{A}$ (denoted $\bot_{\mathcal{A}}$). Further, G maps the greatest element of $\mathcal{A}$ (denoted $\top_{\mathcal{A}}$ in the diagram) to the greatest element of $\mathcal{B}$

(denoted $\top_{\mathcal{B}}$). The posets $F.\mathcal{B}$ and $G.\mathcal{A}$ are "opposites" in the sense that the former contains small elements whereas the latter contains large elements. In particular, $F.\mathcal{B}$ includes the least element of $\mathcal{A}$ and $G.\mathcal{A}$ includes the greatest element of $\mathcal{B}$. They are "united" by the fact that they are isomorphic.

3.2. Fixed points

An immediate issue when confronted with a system of simultaneous equations is whether the system has no solutions, whether it has a unique solution or whether it has many solutions. In the case that the solution space is a complete lattice, the issue is reduced to whether or not the system has more than one solution. The relevant theory is the theory of fixed points and prefix points of a monotonic endofunction.

Suppose $\mathcal{A} = (A, \sqsubseteq)$ is a partially ordered set and suppose f is a monotonic endofunction on $\mathcal{A}$. Then, a *prefix point* of f is an element x of the carrier set A such that $f.x \sqsubseteq x$. We use $\mathsf{Pre}.f$ to denote the set of prefix points of f. A *least prefix point* of f is a solution of the equation

$$x :: f.x \sqsubseteq x \wedge \langle \forall y : f.y \sqsubseteq y : x \sqsubseteq y \rangle.$$

A least prefix point of f is thus a prefix point of f that is smaller than all other prefix points of f. A *least fixed point* of f is a solution of the equation

$$x :: f.x = x \wedge \langle \forall y : f.y = y : x \sqsubseteq y \rangle. \tag{10}$$

Theorem 3.5 (Least prefix point). *Suppose $\mathcal{A} = (A, \sqsubseteq)$ is an ordered set and suppose $f \in A \leftarrow A$ is monotonic. Then, f has at most one least prefix point, $\mu_{\sqsubseteq} f$, characterised by the two properties:*

$$\mu_{\sqsubseteq} f = f.\mu_{\sqsubseteq} f, \tag{11}$$

and, for all $x \in A$,

$$\mu_{\sqsubseteq} f \sqsubseteq x \Leftarrow f.x \sqsubseteq x. \tag{12}$$

Rule (11) states that the least prefix point of f is a fixed point of f. We call it the *computation* rule because it is often used as a left-to-right rewrite rule in the computation of $\mu_{\sqsubseteq} f$. Rule (12) is called the *induction* rule because it is the basis for inductive proofs on sets, such as the natural numbers, defined by a fixed-point equation.

We occasionally write μf, omitting explicit mention of the ordering relation. This is done mostly within prose, but sometimes elsewhere when the ordering is clear from the context.

In general, as suggested by the wording of Theorem 3.5, least fixed points need not exist. A well-known theorem is that a monotonic function on a complete poset is guaranteed to have a least fixed point. (See [28] for historical information.) The posets we consider in this paper are always complete, allowing us to ignore the niceties of existence problems in the statement of theorems and lemmas.

The most powerful of the two rules characterising least prefix points is the induction rule. Its power is, however, somewhat limited because it only allows one to calculate with orderings in which the μ operator is the *principal* operator on the *lower* side of the ordering (i.e., orderings of the form $\mu f \sqsubseteq \cdots$). A frequently used rule, which overcomes this

restriction on the induction rule, can be obtained by combining the calculation properties of Galois connections with those of fixed points.

Theorem 3.6 (μ-fusion). *Suppose $f \in A \leftarrow B$ is the lower adjoint in a Galois connection between the complete posets $(A, \sqsubseteq)$ and $(B, \preceq)$. Suppose that $g \in (B, \preceq) \leftarrow (B, \preceq)$ and $h \in (A, \sqsubseteq) \leftarrow (A, \sqsubseteq)$ are monotonic functions. Then*

$$f.\mu_{\preceq} g = \mu_{\sqsubseteq} h \Leftarrow f \circ g = h \circ f.$$

More generally, if the condition

$$f \circ g = h \circ f$$

holds, f is the lower adjoint in a Galois connection between the posets $(\mathsf{Pre}.h, \sqsubseteq)$ *and* $(\mathsf{Pre}.g, \preceq)$.

We call this theorem μ-"fusion" because it states when application of function f can be "fused" with a fixed point μg to form a fixed point μh. (The rule is also used to "defuse" a fixed point into the application of a function to another fixed point.) The fusion rule is the basis of so-called "loop-fusion" techniques in programming: the combination of two loops, one executed after the other, into a single loop. The theorem also plays a central role in the abstract interpretation of programs; Cousot and Cousot [7] introduced the theorem in this context in a different form.

In order to apply the μ-fusion theorem, we only need to know that f has an upper adjoint; we do not need to know the details of its definition. This is why Theorem 3.3 is important as a way of determining whether a given function does have an upper adjoint.

3.3. Applying fusion: example and non-example

This section discusses two related examples. The first is an example of how the fusion theorem is applied; the second illustrates how the fusion theorem need not be *directly* applicable. We return to the second example later (Example 6.5) and show how it can be generalised in such a way that the fusion theorem does become applicable.

Both examples are concerned with membership of a set. So, let us consider an arbitrary set $\mathcal{U}$. For each x in $\mathcal{U}$ the predicate $(x\in)$ maps a subset P of $\mathcal{U}$ to the boolean value true if x is an element of P and otherwise to false. The predicate $(x\in)$ preserves set union. That is, for all bags $\mathcal{S}$ of subsets of $\mathcal{U}$,

$$x \in \cup\, \mathcal{S} \equiv \langle \exists P : P \in \mathcal{S} : x \in P \rangle.$$

According to the fundamental theorem, the predicate $(x\in)$ thus has an upper adjoint. Indeed, we have, for all booleans b,

$$x \in S \Rightarrow b \;\equiv\; S \subseteq \mathsf{if}\ b \rightarrow \mathcal{U}\ []\ \neg b \rightarrow \mathcal{U} \backslash \{x\}\ \mathsf{fi}.$$

(The complicated definition of the adjoint function illustrates why we often do not wish to know the specific details of a function's upper adjoint!)

Suppose g is a monotonic function on sets (ordered by the subset relation). Let μg denote its least fixed point. The fact that $(x\in)$ is a lower adjoint means that we may be able to apply the fusion theorem to reduce a test for membership in μg to solving a recursive equation. Specifically

$$(x \in \mu g \equiv \mu h) \Leftarrow \langle \forall S :: x \in g.S \equiv h.(x \in S) \rangle.$$

That is, the recursive equation with underlying endofunction g is replaced by the equation with underlying endofunction h (mapping booleans to booleans) if we can establish the property

$$\langle \forall S :: x{\in}g.S \equiv h.(x{\in}S) \rangle.$$

An example of where this is always possible is testing whether the empty word is in the language defined by a context-free grammar. For concreteness, consider the grammar with just one non-terminal S and productions

$$S ::= aS \mid SS \mid \varepsilon.$$

Then the function g maps set X to

$$\{a\}{\cdot}X \cup X{\cdot}X \cup \{\varepsilon\}.$$

We compute the function h as follows:

$$\begin{array}{ll} & \varepsilon{\in}g.S \\ = & \quad \{\text{definition of } g\} \\ & \varepsilon{\in}(\{a\}{\cdot}S \cup S{\cdot}S \cup \{\varepsilon\}) \\ = & \quad \{\text{membership distributes through set union}\} \\ & \varepsilon{\in}\{a\}{\cdot}S \vee \varepsilon{\in}S{\cdot}S \vee \varepsilon{\in}\{\varepsilon\} \\ = & \quad \{\varepsilon{\in}X{\cdot}Y \equiv \varepsilon{\in}X \wedge \varepsilon{\in}Y\} \\ & (\varepsilon{\in}\{a\} \wedge \varepsilon{\in}S) \vee (\varepsilon{\in}S \wedge \varepsilon{\in}S) \vee \varepsilon{\in}\{\varepsilon\} \\ = & \quad \{\bullet\ h.b = (\varepsilon{\in}\{a\} \wedge b) \vee (b \wedge b) \vee \varepsilon{\in}\{\varepsilon\}\} \\ & h.(\varepsilon{\in}S). \end{array}$$

We have thus derived that

$$\begin{array}{l} \varepsilon{\in}\langle \mu_{\subseteq} X :: \{a\}{\cdot}X \cup X{\cdot}X \cup \{\varepsilon\} \rangle \\ \quad = \langle \mu_{\Rightarrow} b :: (\varepsilon{\in}\{a\} \wedge b) \vee (b \wedge b) \vee \varepsilon{\in}\{\varepsilon\} \rangle. \end{array}$$

Note how the definition of h has the same structure as the definition of g. Effectively, set union has been replaced by disjunction and concatenation has been replaced by conjunction. Of course, h can be simplified further (to the constant function true) but that would miss the point of the example.

Now suppose that, instead of taking x to be the empty word, we consider any word other than the empty word. Then, repeating the above calculation with "$\varepsilon\in$" replaced everywhere by "$x\in$", the calculation breaks down at the second step. This is because the empty word is the only word x that satisfies the property

$$x{\in}X{\cdot}Y \equiv x{\in}X \wedge x{\in}Y$$

for all X and Y.

This second example emphasises that the conclusion of μ-fusion demands *two* properties of f, g and h, namely that f be a lower adjoint, and that $f \circ g = h \circ f$. The rule is nevertheless very versatile since being a lower adjoint is far from being uncommon, and many algebraic properties take the form $f \circ g = h \circ f$ for some functions f, g and h. In cases when the rule is not immediately applicable, we have to seek generalisations of f and/or g that do satisfy both properties. Later in the paper, we give several examples.

Example 6.9 shows how this is done in the case of the general membership test for context-free languages, whilst Example 6.10 is a further extension to the problem of editing words to a syntactically correct form. These examples illustrate the ideas well but are not new. Section 7 is about a novel generic problem, viz. determining whether a "bound" can be placed on a function of a context-free grammar. A practical application of this problem is to program analysis [30,35].

4. Regular algebra

In this section, we propose a definition of a regular algebra motivated by a desire to exploit to the full the calculational properties of Galois connections. We give several examples of regular algebras and prove several theorems showing how more regular algebras can be built up from simpler algebras, these constructions also being illustrated by examples.

Our view of a regular algebra is that it is the combination of a monoid (the algebra of composition) and a complete poset (the algebra of choice), with an interface between the two structures. The interface is that composition distributes through choice in all circumstances; in other words, the product operator of the monoid structure admits left and right "division" or "factorisation" operators.

Various axiomatisations of regular algebra have been given in the past, in particular several by Conway [9]. Our definition is formally equivalent to what Conway calls a "standard Kleene algebra" or "**S** algebra". The novelty of our approach is the focus on the use of factors. In fact, Conway himself introduced "(left and right) factors" in the context of regular languages, exploiting implicitly the fact that the concatenation functions $(L\cdot)$ and $(\cdot L)$, for given language L, are both lower adjoints. Some of the most remarkable results in his book make significant use of the existence of factors. However, Conway did not base any of his axiomatisations on this fact, and did not use factors in any context other than regular languages. Other authors have striven for the weakest possible set of axioms that still permit equality of regular languages to be decided. See, for example, Kozen [24]. Such axiomatisations are, however, too weak to encompass all the constructions considered in this paper.

4.1. Definition and examples

Our definition of a regular algebra, Definition 4.3, is a combination of a monoid (Definition 4.1) and a complete lattice, with an interface between the two.

Definition 4.1. A *monoid* is a triple $(A, \times, 1)$, where A is a set, $\times$ is a binary operator and 1 is an element of A, satisfying the properties:

$$1 \times x = x = x \times 1, \quad \text{for all } x \in A \tag{13}$$

and

$$x \times (y \times z) = (x \times y) \times z, \quad \text{for all } x, y, z \in A. \tag{14}$$

The element 1 is called the *unit* of the monoid, and the operator $\times$ is called the *product* operator.

There are many examples of monoids. Numbers (real, integer or natural) form monoids under addition (with unit the number 0) and under multiplication (with unit the number 1). The booleans also form a monoid under disjunction with unit false, and under conjunction with unit true. Word algebras (defined below) are particularly fundamental because they form the free monoids generated by a given set. Other examples will be discussed later.

Example 4.2 (*Word algebra*). Let T be a finite set (the *alphabet*). A *word* is a finite string formed from elements of T. The *empty word* is the string of length zero, and will be denoted by ε . The *concatenation* of words x and y, denoted by $x{\cdot}y$, is the word formed from the string x immediately followed by the string y. The set of all words over the alphabet T is denoted by T^*. With these definitions, $(T^*, \cdot, \varepsilon)$ is a monoid.

Definition 4.3 (*Regular algebra*). A *regular algebra* is a tuple $(A, \times, +, \leq, 0, 1)$ where
(a) $(A, \times, 1)$ is a monoid,
(b) $(A, \leq, +, 0)$ is a complete lattice with least element 0 and binary supremum operator $+$,
(c) for all $a \in A$, the endofunctions $(a\times)$ and $(\times a)$ are both lower adjoints in Galois connections between $(A, \leq)$ and itself.

We use a notation similar to arithmetic division to denote the upper adjoints of the functions $(a\times)$ and $(\times a)$—specifically, we use $(a\backslash)$ and $(/a)$, respectively. Thus, the rules are: for all a, x and y,

$$a \times x \leq y \;\equiv\; x \leq a\backslash y \tag{15}$$

and

$$x \times a \leq y \;\equiv\; x \leq y/a. \tag{16}$$

The operators $\backslash$ and $/$ are called *division* operators, and we often paraphrase requirement 4.3(c) as *product admits* (*left and right*) *division*.

By the fundamental theorem of Galois connections, the existence of the upper adjoints $(a\backslash)$ and $(/a)$ implies that product distributes through addition, and 0 is a zero of product.

The use of the arithmetic symbols ("$\times$", "$+$", etc.), and the accompanying terminology, is suggestive of ordinary arithmetic, making some properties look familiar. Care should be taken, however, not to assume the rules of arithmetic. A possible source of confusion is that we sometimes want to use the symbols with their conventional meaning—real multiplication, real addition, etc. In such cases, we point out the overloading in the text. In order to avoid confusion when two regular algebras are being discussed, we sometimes subscript the operators and constants.

Example 4.4 (*Bool*). The set $\mathbb{B}$ containing the boolean values true and false is the carrier of a regular algebra. The ordering is implication, summation is disjunction, product is conjunction, and left division is implication. The zero of product is false and its unit is true. This algebra forms the basis of decision problems.

The set $\mathbb{B}$ is also the carrier of a dual regular algebra, where the ordering is follows-from, summation is conjunction and product is disjunction. The division $a\backslash b$ is $\neg a \wedge b$. The zero of product is true and its unit is false. We use $\mathbb{B}^D$ to denote this algebra. It, too, occurs in decision problems (where the nature of the problem dictates the computation of a strongest condition rather than a weakest condition).

Example 4.5 (*Min-cost algebra*). A regular algebra that occurs frequently in problems involving some cost function[3] has carrier the set of all real numbers, $\mathbb{R}$, augmented with a largest element, ∞, and a smallest element, $-\infty$. (That is, the carrier is $\mathbb{R} \cup \{\infty, -\infty\}$.) This set forms a monoid where product is defined by

$$\begin{array}{l} x \times y = \mathsf{if}\ x = \infty \vee y = \infty \to \infty \\ \qquad []\ x \neq \infty \wedge y \neq \infty \wedge (x = -\infty \vee y = -\infty) \to -\infty \\ \qquad []\ x{\in}\mathbb{R} \wedge y{\in}\mathbb{R} \to x + y \\ \qquad \mathsf{fi} \end{array}$$

and the unit of product is the real number 0. (The "+" operation on the right side of the above definition is ordinary addition of real numbers.) Ordering its elements by the at-least relation, where by definition $\infty \geq x \geq -\infty$, for all x, the set forms a complete lattice. The supremum is minimum. Moreover, product admits division. To check this, we have to define the upper adjoint of $(\times y)$ for $y{\in}\mathbb{R}$, $y = \infty$ and $y = -\infty$. (Product is clearly symmetric. So the upper adjoint of $(y\times)$ is the same as the upper adjoint of $(\times y)$.) By calculation we get that, for all y, $(\times y)$ is a lower adjoint, with upper adjoint $(/y)$ given by

$$\begin{array}{l} z/y = \mathsf{if}\ z = -\infty \to -\infty \\ \qquad []\ z = \infty \wedge y \neq \infty \to \infty \\ \qquad []\ y = \infty \to -\infty \\ \qquad []\ y = -\infty \wedge z \neq -\infty \to \infty \\ \qquad []\ y{\in}\mathbb{R} \wedge z{\in}\mathbb{R} \to z - y \\ \qquad \mathsf{fi}. \end{array}$$

Example 4.6 (*Bottleneck algebra*). Bottleneck problems are problems with a max–min requirement. For example, if it is required to drive a high load under a number of low bridges, we want to find the maximum over all different routes of the minimum-height bridge on the route. A regular algebra fundamental to bottleneck problems has carrier the set of all real numbers, augmented with largest and smallest values, ∞ and $-\infty$, respectively. The addition operator is maximum (so that the ordering relation is at-most) and the product operator is minimum. The minimum operator is easily seen to satisfy the property

$$x{\downarrow}y \leq z \equiv x \leq (\downarrow y)^\sharp z,$$

where

$$\begin{array}{l} (\downarrow y)^\sharp z = \mathsf{if}\ y \leq z \to \infty \\ \qquad []\ y > z \to z \\ \qquad \mathsf{fi}. \end{array}$$

That is, the product operator in the algebra admits division.

[3] "Costs" in real-world applications are usually non-negative. It is possible to define a regular algebra with carrier $\mathbb{R}^{\geq 0} \cup \{\infty\}$. However, the inclusion of negative numbers means that $(\times y)$ and $(/y)$ are inverses functions for $y{\in}\mathbb{R}$. This is important for some applications. See Section 4.2.

4.2. Lexicographic combinations

Normally in decision problems (such as determining whether or not there is a path between two given points in a graph), it is not sufficient to know that the answer is yes; we also want witnessing information justifying the yes answer. To cater for this, we consider an algebra of pairs, where the first element is a boolean, and the second element is a set of solutions. Similarly, when solving optimisation problems, we are not just interested in the optimum value of the cost function (for example, the length of a shortest path), but in determining some value in the domain of solutions that optimises the cost function (for example, a path that has minimum length). Again, the solution is to consider an algebra of pairs, where the first element is the cost and the second element is a set of values that have that cost. An algebra of pairs is also appropriate when two criteria for optimality are combined. For example, we may wish to determine, among all least-cost solutions to a given problem, a solution that optimises some other criterion, like size or weight.

Theorem 4.10 is the basis for the use of regular algebra in such circumstances. The theorem details the construction of an algebra of pairs in which the pairs are ordered lexicographically. In general, the construction is only useful if the "interesting" elements in the first algebra are "cancellative", as defined below. In the min-cost algebra defined in Example 4.5, the cancellative elements are the real numbers; ∞ and $-\infty$ are not cancellative. Further explanation is given after the statement and proof of Theorem 4.10.

Definition 4.7. An element y of a regular algebra is said to be *cancellative* if, for all x,

$$(x \times y)/y = (x/y) \times y = x = y \times (y \setminus x) = y \setminus (y \times x).$$

It is easy to verify that, in the min-cost algebra of Example 4.5, all real numbers are cancellative. This is because the product $x \times y$ of real values x and y is their real addition $x + y$, and the division x/y is their difference $x - y$; also, in the case that x is ∞ or $-\infty$ and y is a real number, the product $x \times y$ is x, as is x/y.

In a boolean algebra, true is cancellative because, for all x, $x \wedge \mathsf{true} = x$ and $x \Leftarrow \mathsf{true} = x$. (Conjunction, "$\wedge$", is the product operator, and "if", "$\Leftarrow$", is the division operator.)

Lemma 4.8. *Suppose z is cancellative. Then for all x and y, the following are equivalent*:

$$x = y, \quad x \times z = y \times z, \quad z \times x = z \times y, \quad x/z = y/z, \quad z\backslash x = z\backslash y.$$

Thus, so too are:

$$x \neq y, \quad x \times z \neq y \times z, \quad z \times x \neq z \times y, \quad x/z \neq y/z, \quad z\backslash x \neq z\backslash y.$$

Also, for all x and y, the following are equivalent:

$$x \leq y, \quad x \times z \leq y \times z, \quad z \times x \leq z \times y, \quad x/z \leq y/z, \quad z\backslash x \leq z\backslash y.$$

Thus, so too are:

$$x < y, \quad x \times z < y \times z, \quad z \times x < z \times y, \quad x/z < y/z, \quad z\backslash x < z\backslash y.$$

Lemma 4.9. *The set of cancellative elements of a regular algebra includes the unit* 1 *and is closed under product and under left and right division. (The cancellative elements almost form a group, but not quite*; $1/x$ *and* $x\backslash 1$ *are, respectively, the left and right inverses of (cancellative)* x*, but they need not be equal.)*

Conversely, if $x \times y$, x/y or $x\backslash y$ is not cancellative, at least one of x or y is not cancellative.

Theorem 4.10 (Lexicographic combination). *Suppose $\mathcal{R}_1$ and $\mathcal{R}_2$ are both regular algebras. Suppose further that the ordering of elements in $\mathcal{R}_1$ is total. Define the set P to be the set of ordered pairs (x, r) where $x \in \mathcal{R}_1$, $r \in \mathcal{R}_2$, and $r = 0_2$ if x is not cancellative. Order P lexicographically; specifically, let*

$$(x, r) \preceq (y, s) \equiv x <_1 y \vee (x = y \wedge r \leq_2 s).$$

Define product on elements of P coordinatewise:

$$(x, r) \otimes (y, s) = (x \times_1 y, r \times_2 s).$$

Define addition by

$$\begin{aligned}(x, r) \oplus (y, s) = \ &\textsf{if } x <_1 y \to (y, s)\\ &[]\ x = y \to (x, r +_2 s)\\ &[]\ y <_1 x \to (x, r)\\ &\textsf{fi}.\end{aligned}$$

Then $(P, \otimes, \oplus, \preceq, (0_1, 0_2), (1_1, 1_2))$ is a regular algebra.

Proof. It is easily shown that P is a monoid—the product of (x, r) and (y, s) is well-defined because, if $x \times_1 y$ is not cancellative, either x or y is not cancellative (by Lemma 4.9); so, either r or s is 0_2, and $r \times_2 s$ is also 0_2. The unit of product is the pair $(1_1, 1_2)$.

Second, P is complete. Any function, f, with range P has two components f_1 and f_2, say, with ranges $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. The first component of the supremum of f is the supremum of f_1, and the second component is the supremum of f_2, if this is cancellative, and 0_2, if it is not.

It remains to show that factors of the product operation exist. A guess at the definition of $(z, t)/(y, s)$ is

$$(z, t)/(y, s) = (z/_1 y, u),$$

where u is defined to be $t/_2 s$ if $z/_1 y$ is cancellative and, otherwise, is 0_2. This guess is indeed correct. The proof takes the following form, where only the middle step needs to be filled in:

$$\begin{aligned}&(x, r) \otimes (y, s) \preceq (z, t)\\ =\ &\{\text{definitions of } \otimes \text{ and } \preceq\}\\ &x \times_1 y <_1 z \vee (x \times_1 y = z \wedge r \times_2 s \leq_2 t)\\ =\ &\{u = t/_2 s \text{ if } z/_1 y \text{ is cancellative, } u = 0_2, \text{ otherwise.}\\ &\quad \text{See the case analysis below}\}\\ &x <_1 z/_1 y \vee (x = z/_1 y \wedge r \leq_2 u)\\ =\ &\{\text{definition of } (z, t)/(y, s) \text{ and } \preceq\}\\ &(x, r) \preceq (z, t)/(y, s).\end{aligned}$$

We split the verification of the penultimate step into two cases: y cancellative and y not cancellative. First,

$$\begin{aligned}&x \times_1 y <_1 z \vee (x \times_1 y = z \wedge r \times_2 s \leq_2 t)\\ =\ &\{\text{assume } y \text{ is cancellative}\}\end{aligned}$$

$$x <_1 z/_1 y \vee (x = z/_1 y \wedge r \leq_2 t/_2 s).$$

Second, assume y is not cancellative.

$$\begin{array}{ll} & x \times_1 y <_1 z \vee (x \times_1 y = z \wedge r \times_2 s \leq_2 t) \\ = & \{y \text{ is not cancellative, so } s = 0_2\} \\ & x \times_1 y <_1 z \vee x \times_1 y = z \\ = & \{\text{factors}\} \\ & x \leq_1 z/_1 y \\ = & \{y \text{ is not cancellative, so } s = 0_2\} \\ & x <_1 z/_1 y \vee (x = z/_1 y \wedge r \leq_2 t/_2 s). \end{array}$$

In both cases, we complete the calculation by considering two further cases: when $z/_1 y$ is cancellative and when $z/_1 y$ is not cancellative.

$$\begin{array}{ll} & x <_1 z/_1 y \vee (x = z/_1 y \wedge r \leq_2 t/_2 s) \\ = & \{\text{if } z/_1 y \text{ is cancellative, } u = t/_2 s, \text{ by definition;} \\ & \quad \text{if } z/_1 y \text{ is not cancellative, } x = z/_1 y \Rightarrow r = 0_2, \text{ and} \\ & \quad u = 0_2, \text{ by definition}\} \\ & x <_1 z/_1 y \vee (x = z/_1 y \wedge r \leq_2 u). \qquad \square \end{array}$$

Note how the construction of the lexicographic combination of two algebras effectively ignores information contained in the second component of a pair when the first component is not cancellative. This formalises the circumstances when the second component is of no interest to a computation. For example, we might use a lexicographic combination with the booleans as the first component in a searching problem. When the first component is true (which is cancellative), the search has succeeded and further information about the results of the search are desired. For example, in a path-finding problem, we may want to know how to find a path rather than just that a path exists. When the first component is false (which is not cancellative), the search has failed and further information is of no use. Similarly, in a cost-optimisation problem, we might use the lexicographic combination of the min-cost algebra with another algebra. (See the example below for a specific case.) An optimal cost of ∞ means that no solution exists; an optimal cost of $-\infty$ would indicate an improperly formulated problem (for example, cycles of negative length in a network of distances). In both cases, further information is irrelevant.

Of course, this means that the theorem is itself only applicable when the "interesting" elements are cancellative. The following example gives one application where the theorem is applicable together with a closely related application where the theorem is not applicable.

Example 4.11. An instance of Theorem 4.10 is when we combine the min-cost algebra of Example 4.5 with the bottleneck algebra of Example 4.6 *in that order*. The theorem is *not* applicable when the algebras are combined in the opposite order. This is because, in the bottleneck algebra, the only cancellative element is the unit of product (which happens to be ∞). This has important implications for the applicability of fixed-point algorithms.

Suppose we consider a network of cities connected by a number of roads. Each road has a certain length and along each road there is a low bridge. It is required to drive a high load from one of the cities to another by an "optimum" route (a route being a sequence of connecting roads).

One criterion of optimality is that we choose, among the shortest routes, a route that maximises the minimum height of bridge along the route. A second criterion of optimality is that, among the routes that maximise the minimum height of bridge along the route, we choose a shortest route.

The construction of Theorem 4.10 is applicable to the first criterion of optimality. The elements of the algebra are ordered pairs (*distance*, *height*). A route with "cost" (d, h) is better than a route with "cost" (e, k) if $d < e$ or $d = e$ and $h \geq k$. As this is a regular algebra, it is possible to apply the standard all-pairs path-finding algorithm (variously known as the (Roy-)Warshall algorithm [33,39], Floyd's algorithm [15]—see [5] for more details) to determine, for all pairs of cities, the cost of a best route between the cities. Assuming the cities are numbered from 1 to N and that the distance and height of the road connecting city i to city j is recorded in the $N \times N$ graphs d and h, the graphs are updated by the following code (in which all operators have their conventional arithmetic meaning). On termination d_{ij} is the length of a shortest route from i to j, and h_{ij} is the maximum over all routes with length d_{ij} of the minimum height of a bridge on the route.

```
for each k, 1 ≤ k ≤ N
do for each pair (i, j), 1 ≤ i, j ≤ N
   do if d_ij < d_ik + d_kj → skip
      [] d_ij = d_ik + d_kj → h_ij := h_ij ↑ (h_ik ↓ h_kj)
      [] d_ij > d_ik + d_kj → h_ij := h_ik ↓ h_kj
      fi;
      d_ij := d_ij ↓ (d_ik + d_kj)
   end_for
end_for
```

Note that the sequence of two assignment statements is equivalent to the single assignment

$$(d_{ij}, h_{ij}) := (d_{ij}, h_{ij}) \oplus ((d_{ik}, h_{ik}) \otimes (d_{kj}, h_{kj})),$$

where $\oplus$ and $\otimes$ are the addition and product operators as defined in Theorem 4.10. So the algorithm has exactly the same shape as the Roy–Warshall–Floyd algorithm.

The construction of Theorem 4.10 is *not* applicable to the second criterion of optimality; indeed an attempt to embed the lexicographical ordering on (*height*, *distance*) pairs in a regular algebra fails because product does not distribute through addition. As a consequence, *the following code does* ***not*** *determine the cost of an optimal route*, although a naive comparison of this code with that above might lead one to suspect that that is the case.

```
for each k, 1 ≤ k ≤ N
do for each pair (i, j), 1 ≤ i, j ≤ N
```

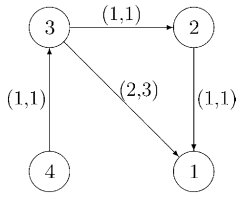


Fig. 1. (Height, distance) graph.

do if $h_{ij} > h_{ik} \downarrow h_{kj} \rightarrow$ skip
[] $h_{ij} = h_{ik} \downarrow h_{kj} \rightarrow d_{ij} := d_{ij} \downarrow (d_{ik} + d_{kj})$
[] $h_{ij} < h_{ik} \downarrow h_{kj} \rightarrow d_{ij} := d_{ik} + d_{kj}$
fi;
$h_{ij} := h_{ij} \uparrow (h_{ik} \downarrow h_{kj})$
end_for
end_for.

The incorrectness of the code is demonstrated by the graph shown in Fig. 1. In this graph, the first component of an edge label is the height, and the second component is the distance.

Note that the best route from the city numbered 4 to the city numbered 1 is the route 4, 3, 2, 1. However, the best route from the city numbered 3 to the city numbered 1 is the direct route, 3, 1. So, the route from 4 follows a suboptimal route from city 3. The code above correctly determines the minimum bridge-height on the route from 4 to 1, but incorrectly determines the distance to be 4 rather than the correct distance 3.

4.3. Vector algebras

Simultaneous equations typically involve several unknowns, possibly even an infinite set of unknowns. This situation is modelled in a straightforward and standard way. We consider a collection of equations in a collection of unknowns as a single equation in a single unknown, that unknown being a vector of values. And a vector is just a function with range the carrier set of a regular algebra. The set of functions with domain some arbitrary, fixed set and range a regular algebra forms a regular algebra if we extend the operators in the range algebra pointwise. Formally:

Theorem 4.12 (Vector algebras). *Suppose* $\mathcal{A} = (A, \times, +, \leq, 0, 1)$ *is a regular algebra and suppose* $\mathcal{B} = (B, \sqsubseteq)$ *is an arbitrary poset. Let* $A \leftarrow B$ *denote the set of monotonic functions to* $(A, \leq)$ *from* $(B, \sqsubseteq)$. *Define* $\mathcal{A} \leftarrow \mathcal{B}$ *to be* $(A \leftarrow B, \dot{\times}, \dot{+}, \dot{\leq}, \dot{0}, \dot{1})$, *where product, addition and ordering are defined pointwise* (*that is*, $(f \ \dot{\times} \ g).x = f.x \times g.x$, *and similarly for* $\dot{+}$, *and* $f \ \dot{\leq} \ g \equiv \langle \forall x :: f.x \leq g.x \rangle$), *and* $\dot{0}$ *and* $\dot{1}$ *are the constant functions returning* 0 *and* 1, *respectively. Then* $\mathcal{A} \leftarrow \mathcal{B}$ *is a regular algebra.*

Proof. Straightforward expansion of the definitions. □

4.4. Power-set regular algebras

Some important examples of regular algebras are power-set algebras. These are introduced in this section.

Definition 4.13 (*Power-set monoid*). Suppose $(A, \cdot, 1)$ is a monoid. Let 2^A denote the set of all subsets of A. The product operator is extended to 2^A by

$$X{\cdot}Y = \{x{\cdot}y \mid x{\in}X \wedge y{\in}Y\}.$$

Lemma 4.14. *Suppose* $(A, \cdot, 1)$ *is a monoid. Then, the algebra* $(2^A, \cdot, \cup, \subseteq, \phi, \{1\})$ (*where product is given by Definition 4.13*) *is a regular algebra.*

Example 4.15 (*Bool*). The simplest possible example of a monoid has carrier $\{1\}$. The subsets of this set are the empty set and the set itself. The power-set regular algebra is clearly isomorphic to the booleans. Choosing to map the empty set to false and $\{1\}$ to true, the product operation of the regular algebra is conjunction, and the addition operator is disjunction.

This is Example 4.4 discussed earlier. The dual algebra is obtained by choosing to map the empty set to true and $\{1\}$ to false.

Example 4.16. The standard example of a regular algebra has carrier set the set of words over some finite alphabet.

A *language* over alphabet T is a set of words, i.e., a subset of T^*. The power-set regular algebra constructed from the monoid $(T^*, \cdot, \varepsilon)$ (recall Example 4.2) has carrier the set of all languages over alphabet T.

It is easy to determine that, for languages X and Y, the factor X/Y is $\{z \mid \{z\}{\cdot}Y \subseteq X\}$. Dually, $X\backslash Y$ is $\{z \mid X{\cdot}\{z\} \subseteq Y\}$.

Example 4.17 (*First sets*). If we are interested in the prefixes of words in a language up to a certain length, a slight variation on the definition of concatenation of words is appropriate. Suppose k is a natural number. Let $T^{\leq k}$ denote the set of all words whose length is at most k. Suppose F_k is the function that truncates a word of length at least k to its first k symbols. Define the concatenation $x \times y$ of words x and y by

$$\begin{aligned} x \times y = {} & \textsf{if length.}(x{\cdot}y) \leq k \rightarrow x{\cdot}y \\ & \textsf{[] length.}(x{\cdot}y) \geq k \rightarrow F_k(x{\cdot}y) \\ & \textsf{fi.} \end{aligned}$$

Then $(T^{\leq k}, \times, \varepsilon)$ is a monoid. The power-set algebra has carrier the set of all languages whose words have length at most k. Concatenation of languages involves truncating words down to length at most k.

4.5. Graph algebras

If regular algebra is to be applied to path-finding problems, it is vital that the property of being a regular algebra can be extended to graphs/matrices[4] [5].

Often, graphs are supposed to have finite dimensions. In the present circumstances—the assumption of a complete lattice—there is no need to impose this as a requirement. Indeed, if we are to include the algebra of binary relations over some given, possibly infinite, set in the class of regular algebras then we certainly should not require graphs to have finite dimension. Other applications demand a very general definition of a graph. In the following, a *binary relation* is just a set of pairs.

Definition 4.18. Suppose r is a binary relation and suppose A is a set. A (*labelled*) *graph of dimension r over A* is a function f with domain r and range A. Elements of relation r are called *edges*.

We use $M_r A$ to denote the class of all labelled graphs of dimension r over A. If f is a graph and the pair (i, j) is an element of r, then $i\langle f\rangle j$ will be used to denote the application of f to the pair (i, j).

In order to link our work more easily with other literature, we call a graph a *matrix* if its dimension is a Cartesian product $M \times N$, for some sets M and N. Note, however, that we allow M and N to be infinite sets.

Defining addition and product of graphs involves a slight generalisation of the standard definitions of addition and product of matrices.

Definition 4.19 (*Addition and product*). Suppose $\mathcal{R} = (A, \times, +, \leq, 0, 1)$ is a regular algebra. Then zero and the addition and product operators of $\mathcal{R}$ can be extended to graphs over $\mathcal{R}$ as follows. Two graphs f and g of the same dimension r can be ordered according to the rule: for all pairs (i, j) in r

$$f \dot{\leq} g \equiv \langle \forall i, j :: i\langle f\rangle j \leq i\langle g\rangle j\rangle.$$

Evaluation of suprema is just pointwise. In particular, f and g of the same dimension r are added according to the rule: for all pairs (i, j) in r

$$i\langle f \dot{+} g\rangle j = i\langle f\rangle j + i\langle g\rangle j.$$

Two graphs f and g of dimensions r and s can be multiplied to form a graph of dimension $r \circ s$ (the relational composition of r and s) according to the rule: for all pairs (i, j) in $r \circ s$

$$i\langle f \dot{\times} g\rangle j = \langle \Sigma k : (i, k) \in r \wedge (k, j) \in s : i\langle f\rangle k \times k\langle g\rangle j\rangle.$$

Finally, the zero graph, denoted by $\mathbf{0}$, is defined by: for all pairs (i, j) in r,

$$i\langle \mathbf{0}\rangle j = 0.$$

It is straightforward to check that $\mathbf{0}$ is a unit of addition and a zero of product, as the name suggests.

[4] For us, the words *graph* and *matrix* are interchangeable. In some applications "graph" is the word that is traditionally used; in others, "matrix" is more conventional. For consistency, we mostly use "graph".

Theorem 4.20 (Graph algebras). *Suppose* $\mathcal{R} = (A, \times, +, \leq, 0, 1)$ *is a regular algebra, and suppose* r *is a reflexive, transitive relation. Define a partial ordering, addition and product operators as in Definition 4.19, and define the unit graph, denoted by* $\mathbf{1}$*, by*

$$\begin{array}{l} i\langle \mathbf{1}\rangle j = \mathsf{if}\ i = j \to 1 \\ \qquad\qquad [\!]\ i \neq j \to 0 \\ \qquad\qquad \mathsf{fi}. \end{array}$$

Then, the algebra $M_r\mathcal{R} = (M_r A, \dot{\times}, \dot{+}, \dot{\leq}, \mathbf{0}, \mathbf{1})$ *so defined is a regular algebra.*

There are a number of parts to the proof of Theorem 4.20, some of which are standard, and are omitted here. (For example, $\mathbf{0}$ is the zero of product, $\mathbf{1}$ is its unit, and addition and product are associative. Note that $M_r A$ is closed under the product operation and contains $\mathbf{1}$ on account of the assumptions that r is transitive and reflexive, respectively.)

Since graphs are functions and the supremum operator is defined pointwise, it is easy to show that $(M_r A, \dot{\leq})$ forms a complete lattice. (The fact that the domain of a graph is a relation is not relevant to this part of the proof.) Graph product is not defined pointwise, however, so we need a separate lemma to establish the existence of an upper adjoint for graph product.

Lemma 4.21. *The product operation on graphs admits left- and right-division operators. That is, there are operators* $\backslash$ *and* $/$ *such that, for all graphs* f, g *and* h, $f \mathbin{\dot{\times}} g \mathbin{\dot{\leq}} h \equiv f \mathbin{\dot{\leq}} h/g$ *and* $f \mathbin{\dot{\times}} g \mathbin{\dot{\leq}} h \equiv g \mathbin{\dot{\leq}} f\backslash h$.

Proof. We have to construct the upper adjoint of $(f\,\dot{\times})$ for any graph f with range A. In the following calculation, the range of the dummies i, j, and k is constrained by the requirements that the pairs (i, j), (j, k) and (i, k) are all in the relation r.

$$\begin{array}{ll} & f \mathbin{\dot{\times}} g \mathbin{\dot{\leq}} h \\ = & \quad \{\text{definition of pointwise ordering}\} \\ & \langle \forall i, k :: i\langle f \mathbin{\dot{\times}} g\rangle k \leq i\langle h\rangle k\rangle \\ = & \quad \{\text{definition of composition}\} \\ & \langle \forall i, k :: \langle \Sigma j :: (i\langle f\rangle j) \times (j\langle g\rangle k)\rangle \leq i\langle h\rangle k\rangle \\ = & \quad \{\text{supremum and nesting}\} \\ & \langle \forall i, j, k :: (i\langle f\rangle j) \times (j\langle g\rangle k) \leq i\langle h\rangle k\rangle \\ = & \quad \{\mathcal{R} \text{ is a regular algebra.} \\ & \qquad \text{Thus, } ((i\langle f\rangle j)\times) \text{ has upper adjoint } (i\langle f\rangle j)\backslash, \text{ say}\} \\ & \langle \forall i, j, k :: j\langle g\rangle k \leq (i\langle f\rangle j)\backslash(i\langle h\rangle k)\rangle \\ = & \quad \{\text{infima}\} \\ & \langle \forall j, k :: j\langle g\rangle k \leq \langle \Pi i :: (i\langle f\rangle j)\backslash(i\langle h\rangle k)\rangle\rangle \\ = & \quad \{\text{definition of pointwise ordering}\} \\ & g \mathbin{\dot{\leq}} \langle j, k :: \langle \Pi i :: (i\langle f\rangle j)\backslash(i\langle h\rangle k)\rangle\rangle. \end{array}$$

Dually, it is possible to construct the upper adjoint of $(\dot{\times} g)$ for any graph g. □

There are several important examples of graph regular algebras. The binary relations on some set A is one example. The underlying regular algebra is the booleans $\mathbb{B}$, and the edge relation is the Cartesian product $A \times A$. The relation represented by graph f is the set of pairs (i, k) such that $i\langle f\rangle k$. Graph addition corresponds to the union of relations, and graph product corresponds to the composition of relations. The divisions f/g and $f\backslash g$ are called residuals [11] in the mathematics literature, and weakest pre and post specifications [17] in the computing science literature.

Path problems on finite graphs provide additional examples of graph algebras. The standard example is shortest paths: the underlying algebra is the minimum-cost algebra introduced in Definition 4.5. For further examples see [5,36].

5. Fusion for context-free grammars

In Section 2, we discussed a number of problems that reduce to solving a system of recursive equations with the same structure as the context-free grammar from which they were derived. We view these equations as non-standard *interpretations* of the grammar, homomorphic to the standard interpretation, which is a system of equations in languages. In this section, we present a fundamental theorem that establishes when different interpretations of a context-free grammar are related. In words, a "regular homomorphism" maps one interpretation of a context-free grammar into another.

To make the theorem precise, we define the notion of a regular homomorphism (Definition 5.2) and an interpretation of a context-free grammar (Definition 5.3). Our theorem (Theorem 5.4) is then a relatively straightforward consequence of the definitions.

Definition 5.1 (*Monoid homomorphism*). Suppose $\mathcal{R} = (R, \times_R, 1_R)$ and $\mathcal{S} = (S, \times_S, 1_S)$ are monoids. Suppose m is a function with domain R and range S. Then, m is said to be a *monoid homomorphism* from $\mathcal{R}$ to $\mathcal{S}$ if m preserves units:

$$m.1_R = 1_S$$

and preserves product: for all x and y in R,

$$m.(x \times_R y) = m.x \times_S m.y.$$

Definition 5.2 (*Regular homomorphism*). Let $\mathcal{R} = (R, \times_R, +_R, \leq_R, 0_R, 1_R)$ and $\mathcal{S} = (S, \times_S, +_S, \leq_S, 0_S, 1_S)$ be regular algebras. Suppose m is a function with domain R and range S. Then, m is a *regular homomorphism* from $\mathcal{R}$ to $\mathcal{S}$ if m is a monoid homomorphism (from $(R, \times_R, 1_R)$ to $(S, \times_S, 1_S)$) and it is the lower adjoint in a Galois connection between the two orderings.

Regular homomorphisms are lower adjoints and preserve the structure of monoids. These are the conditions we need to apply the fusion theorem to systems of equations with the structure of a context-free grammar.

Suppose $\mathcal{R}$ is a regular algebra. We view a context-free grammar as a recipe for constructing an endofunction on the vector algebra $\mathcal{R} \leftarrow N$, where N is the set of non-terminals (ordered by equality, in order to meet our definition of a vector algebra). The standard interpretation of a context-free grammar is obtained by taking $\mathcal{R}$ to be $(2^{T^*}, \cdot, \cup, \subseteq, \phi, \{\varepsilon\})$, where T is the set of terminal symbols. For example, a grammar with two non-terminals E and F, and productions

$$E ::= EaE \mid F$$
$$F ::= b \mid E$$

would be interpreted as the function that maps a pair of languages (E, F) to the pair of languages $(E{\cdot}\{a\}{\cdot}E \cup F, \{b\} \cup F)$. The function L_G that maps a non-terminal to the language generated by the non-terminal is the least fixed point of the standard interpretation of the grammar.

In the standard interpretation of a context-free grammar, a terminal symbol, a say, is interpreted as $\{a\}$, and an empty right side of a production is interpreted as $\{\varepsilon\}$. A non-standard interpretation is given by interpreting concatenation and choice as the product and sum operators, respectively, in some regular algebra $\mathcal{R}$; an empty right side of a production is interpreted as the unit of the algebra; also, an interpretation in $\mathcal{R}$ of the terminal symbols needs to be provided.

For example, we might take the regular algebra $\mathbb{B}$ of booleans (Example 4.4), interpreting concatenation as conjunction, choice as disjunction, and each terminal symbol as the boolean true. In this way, a context-free grammar is interpreted as a mapping from a vector of booleans (indexed by the non-terminals) to a vector of booleans. The least fixed point of this function determines, for each non-terminal, whether the language generated by the non-terminal is non-empty.

In addition, context-free grammars have a certain structure (that distinguishes them from, for example, context-sensitive grammars). The following definitions capture this structure, together with the fact that a context-free grammar is a recipe for constructing endofunctions.

Definition 5.3 (*Context-free grammar*). A context-free grammar, with set of non-terminals N and set of terminals T, is a function that maps a regular algebra $\mathcal{R} = (R, \cdot, +, \leq, 0, 1)$ and a function f of type $R \leftarrow T$ to a monotonic endofunction on the vector algebra $\mathcal{R} \leftarrow N$. The allowed functions are those built inductively as follows.

If A is a non-terminal, $A ::= \varepsilon$ is a grammar; the function defined is given by

$$(A ::= \varepsilon)_{\mathcal{R},f}.g.A = 1,$$

and, for all non-terminals $B \neq A$,

$$(A ::= \varepsilon)_{\mathcal{R},f}.g.B = 0.$$

If A is a non-terminal, and a is a terminal, $A ::= a$ is a grammar; the function defined is given by

$$(A ::= a)_{\mathcal{R},f}.g.A = f.a,$$

and, for all non-terminals $B \neq A$,

$$(A ::= a)_{\mathcal{R},f}.g.B = 0.$$

If A and B are non-terminals, $A ::= B$ is a grammar; the function defined is given by

$$(A ::= B)_{\mathcal{R},f}.g.A = g.B,$$

and, for all non-terminals $C \neq A$,

$$(A ::= B)_{\mathcal{R},f}.g.C = 0.$$

If G and H are context-free grammars, then the *choice* $G \mid H$ is a context-free grammar; the function defined is given by, for all non-terminals A,

$$(G \mid H)_{\mathcal{R},f}.g.A = G_{\mathcal{R},f}.g.A + H_{\mathcal{R},f}.g.A.$$

Finally, if G and H are context-free grammars, then the *product* $G \times H$ is a context-free grammar; the function defined is given by, for all non-terminals A,

$$(G \times H)_{\mathcal{R},f}.g.A = G_{\mathcal{R},f}.g.A \cdot H_{\mathcal{R},f}.g.A.$$

(The *interpretations* of G are the functions $G_{\mathcal{R},f}$, where $\mathcal{R}$ is a regular algebra and f is a function to the carrier of $\mathcal{R}$ from the terminal set of G.)

To comply with Definition 5.3, a grammar with productions $E ::= TE \mid a$ and $T ::= E$ (for example) would be written

$$((E ::= T) \times (E ::= E)) \mid (E ::= a) \mid (T ::= E).$$

Formally, our definition captures not just standard Backus-Naur productions, but also semi-extended[5] BNF. For example,

$$(E ::= T) \times ((E ::= T) \mid (E ::= a)) \mid (T ::= E)$$

corresponds to a grammar with productions $E ::= T(T \mid a)$ and $T ::= E$. (The parentheses are metasymbols, not terminal symbols.) Note that, whenever $A \neq B$,

$$((A ::= \alpha) \times (B ::= \beta))_{\mathcal{R},f}.g.C = 0$$

for arbitrary α, β, $\mathcal{R}$, f, g and C. So, all grammars can be reduced to a choice of grammars, such that the left sides of the component in any individual choice are equal.

Theorem 5.4 (Fusion for context-free grammars). *Suppose G is a context-free grammar with non-terminal set N and terminal set T, and suppose $\mathcal{R}$ and $\mathcal{S}$ are regular algebras. Suppose also that m is a regular homomorphism from $\mathcal{R}$ to $\mathcal{S}$. Suppose f is a function to (the carrier of) $\mathcal{R}$ from T. Let F equal $G_{\mathcal{R},f}$ and H equal $G_{\mathcal{S},(m\circ f)}$. Then, for all non-terminals A,*

$$m.(\mu F.A) = \mu H.A.$$

Proof. We begin by putting the statement of the theorem into a form where we can use the fusion theorem (Theorem 3.6):

$$\begin{aligned} & \langle \forall A :: m.(\mu F.A) = \mu H.A \rangle \\ = \quad & \{\text{extensionality, definition of function composition}\} \\ & m \circ \mu F = \mu H. \end{aligned}$$

Now, the function $\langle g :: m \circ g \rangle$ is a lower adjoint in a Galois connection of the point-wise-ordered posets $R \leftarrow N$ and $S \leftarrow N$ (where R and S are the carriers of $\mathcal{R}$ and $\mathcal{S}$, respectively, and N is the set of non-terminals of G). Specifically, if m has upper adjoint $m^\sharp$, the upper adjoint of $\langle g :: m \circ g \rangle$ is $\langle g :: m^\sharp \circ g \rangle$. (This is a well-known, and easily verified, property of Galois connections.) Continuing the calculation:

$$m \circ \mu F = \mu H$$

[5] We say "semi-extended" because we do not consider, for example, iteration as an additional operator on grammars. Adding iteration does not cause any difficulty; our theory is easily extended to cope.

$$
\begin{array}{ll}
\Leftarrow & \{\text{Theorem 3.6, with} \\
& \quad f, g, h := \langle g :: m \circ g\rangle, F, H\} \\
& \langle g :: m \circ g\rangle \circ F = H \circ \langle g :: m \circ g\rangle \\
= & \{\text{extensionality, definition of function composition}\} \\
& \langle \forall g :: \langle \forall A :: m.(F.g.A) = H.(m \circ g).A\rangle\rangle \\
= & \{\text{definitions of } F \text{ and } H\} \\
& \langle \forall g :: \langle \forall A :: m.(G_{\mathcal{R},f}.g.A) = G_{\mathcal{S},(m\circ f)}.(m \circ g).A\rangle\rangle.
\end{array}
$$

Thus, we have to prove that, for all g and A,

$$m.(G_{\mathcal{R},f}.g.A) = G_{\mathcal{S},(m\circ f)}.((m \circ g).A).$$

This we do by induction on the structure of context-free grammars.

Base cases: In the case that the grammar G is $(A ::= \varepsilon)$, we have, first,

$$
\begin{array}{ll}
& m.((A ::= \varepsilon)_{\mathcal{R},f}.g.A) \\
= & \{\text{definition of } (A ::= \varepsilon)\} \\
& m.1_{\mathcal{R}} \\
= & \{m \text{ is a regular homomorphism}\} \\
& 1_{\mathcal{S}} \\
= & \{\text{definition of } (A ::= \varepsilon)\} \\
& (A ::= \varepsilon)_{\mathcal{S},(m\circ f)}.(m \circ g).A,
\end{array}
$$

and, second, for all non-terminals $B \neq A$,

$$
\begin{array}{ll}
& m.((A ::= \varepsilon)_{\mathcal{R},f}.g.B) \\
= & \{\text{definition of } (A ::= \varepsilon)\} \\
& m.0_{\mathcal{R}} \\
= & \{m \text{ is a regular homomorphism}\} \\
& 0_{\mathcal{S}} \\
= & \{\text{definition of } (A ::= \varepsilon),\ B \neq A\} \\
& (A ::= \varepsilon)_{\mathcal{S},(m\circ f)}.(m \circ g).B.
\end{array}
$$

In the case that G is $(A ::= a)$, where a is a terminal symbol, we have, first,

$$
\begin{array}{ll}
& m.((A ::= a)_{\mathcal{R},f}.g.A) \\
= & \{\text{definition}\} \\
& m.(f.a) \\
= & \{m.(f.a) = (m \circ f).a, \text{ definition of } (A ::= a)\} \\
& (A ::= a)_{\mathcal{S},(m\circ f)}.(m \circ g).A,
\end{array}
$$

and, second, for all non-terminals $B \neq A$,

$$
\begin{array}{ll}
& m.((A ::= a)_{\mathcal{R},f}.g.B) \\
= & \{\text{definition of } (A ::= a)\} \\
& m.0_{\mathcal{R}}
\end{array}
$$

$$
\begin{array}{ll}
= & \{m \text{ is a regular homomorphism}\} \\
& 0_{\mathcal{S}} \\
= & \{\text{definition of } (A ::= a),\ B \neq A\} \\
& (A ::= a)_{\mathcal{S},(m \circ f)}.(m \circ g).B.
\end{array}
$$

The final base case is when G is $(A ::= B)$, where B is a non-terminal symbol. In this case, we have, first,

$$
\begin{array}{ll}
& m.((A ::= B)_{\mathcal{R},f}.g.A) \\
= & \{\text{definition of } (A ::= B)\} \\
& m.(g.B) \\
= & \{m.(g.B) = (m \circ g).B, \text{ definition of } (A ::= B)\} \\
& (A ::= B)_{\mathcal{S},(m \circ f)}.(m \circ g).A,
\end{array}
$$

and, second, for all non-terminals $C \neq A$,

$$
\begin{array}{ll}
& m.((A ::= B)_{\mathcal{R},f}.g.C) \\
= & \{\text{definition of } (A ::= B)\} \\
& m.0_{\mathcal{R}} \\
= & \{m \text{ is a regular homomorphism}\} \\
& 0_{\mathcal{S}} \\
= & \{\text{definition of } (A ::= B),\ C \neq A\} \\
& (A ::= B)_{\mathcal{S},(m \circ f)}.(m \circ g).C.
\end{array}
$$

For the induction step, there are two cases. First, the case $(G \mid G')$

$$
\begin{array}{ll}
& m.((G \mid G')_{\mathcal{R},f}.g.A) \\
= & \{\text{definition of } (G \mid G')\} \\
& m.(G_{\mathcal{R},f}.g.A +_R G'_{\mathcal{R},f}.g.A) \\
= & \{m \text{ is a regular homomorphism}\} \\
& m.(G_{\mathcal{R},f}.g.A) +_S m.(G'_{\mathcal{R},f}.g.A) \\
= & \{\text{induction hypothesis}\} \\
& G_{\mathcal{S},(m \circ f)}.(m \circ g).A +_S G'_{\mathcal{S},(m \circ f)}.(m \circ g).A \\
= & \{\text{definition of } (G \mid G')\} \\
& (G \mid G')_{\mathcal{S},(m \circ f)}.(m \circ g).A.
\end{array}
$$

The final case, $(G \times H)$, proceeds similarly. □

6. Measures

Theorem 5.4 imposes quite strong requirements on the function m. It has to be a lower adjoint and it has to be a monoid homomorphism. In many circumstances, being a lower adjoint is met automatically by the way that m is constructed. Also, being a monoid homomorphism often reduces to a simpler requirement, again because of the way m is con-

structed. This section is about a general method of constructing regular homomorphisms for which this is the case. The heart of the construction is the extension of a function on the elements of a monoid—a so-called *measure*—to a function on the power-set regular algebra induced by that monoid.

The name "measure" comes from the most common sorts of application: these include the length of a word, the first k symbols in a word, and the edit distance of a word from some given word. For example, the length function on words is extended to the length-of-a-shortest-word function on languages. The construction method guarantees that the latter is a regular homomorphism.

Definition 5.1 imposes two requirements on the function m. Note, however, that the requirement that it preserve the unit of $\mathcal{R}$ is redundant if we restrict attention to the values in $m.R$, the image of R under m. After all, we have

$$m.x = m.(x \times_R 1_R) = m.x \times_S m.1_R$$

and, similarly,

$$m.y = m.(1_R \times_R y) = m.1_R \times_S m.y.$$

Also, the product operator of $\mathcal{S}$ is automatically associative, when restricted to $m.R$, since, for all x, y and z in R,

$$m.x \times_S (m.y \times_S m.z) = m.(x \times_R y \times_R z) = (m.x \times_S m.y) \times_S m.z.$$

As a consequence, $(m.R, \times_S, m.1_R)$ is a monoid, irrespective of whether or not $\mathcal{S}$ is a monoid. Similarly, if m is a lower adjoint in a Galois connection, the unity-of-opposites theorem tells us that $m.R$ is a complete lattice. In more complicated applications—see Example 6.10 and Section 7—these observations are vital. Formally, the range of a measure that is "compositional", that has domain a regular algebra, and is a lower adjoint, is a regular algebra—Theorem 6.2.

Definition 6.1 (*Compositional measure*). Let $\mathcal{R} = (R, \times, 1)$ be a monoid. Suppose S is a set that is closed under a binary "product" operator, which we denote by "$\otimes$". Suppose m is a function with domain R and range S. Then m is said to be *compositional* if, for all x and y in R,

$$m.(x \times y) = m.x \otimes m.y.$$

Theorem 6.2 (Range algebras). *Suppose $\mathcal{R} = (R, \times, +, \leq, 0, 1)$ is a regular algebra, and $\mathcal{S} = (S, \preceq)$ is a partially ordered set. Suppose S is closed under a binary product operator "$\otimes$". Suppose m is a function with domain R and range S that is compositional and is the lower adjoint in a Galois connection between the orderings. Let $m.R$ be the image of R under m and let $m^\sharp$ denote its upper adjoint. Then $m.\mathcal{R} = (m.R, \otimes, \oplus, \preceq, m.0, m.1)$ is a regular algebra, where, for all x and y in S,*

$$x \oplus y = m.(m^\sharp.x + m^\sharp.y).$$

Moreover, m is a regular homomorphism from $\mathcal{R}$ to $m.\mathcal{R}$.

Proof. As discussed above, it is easy to verify that compositionality of m implies that $(S, \otimes, m.1)$ is a monoid. Also, the unity-of-opposites theorem tells us that $(m.R, \preceq)$ is a complete lattice with binary supremum operator $\oplus$ as defined above, and least element $m.0$. To show that $m.\mathcal{R}$ is a regular algebra, it thus suffices to show that $m.\mathcal{R}$ admits left- and right-division operators.

To do this, we exploit the fundamental theorem of Galois connections, Theorem 3.3. Specifically, we show that product, on the left and on the right, preserves suprema in $(m.R, \preceq)$. Here is the proof that product-on-the-left preserves suprema. (As always, we use Σ to denote the supremum operator; the subscript indicates which poset is intended.) For all x in R and functions f with range $m.R$,

$$
\begin{array}{ll}
& m.x \otimes \Sigma_{m.R}.f \\
= & \quad \{\text{unity-of-opposites: Theorem 3.4}\} \\
& m.x \otimes m.\left(\Sigma_R.(m^\sharp \circ f)\right) \\
= & \quad \{m \text{ is compositional}\} \\
& m.\left(x \times \Sigma_R.(m^\sharp \circ f)\right) \\
= & \quad \{\text{product in } R \text{ admits division; so, by the fundamental} \\
& \quad \text{theorem (Theorem 3.3), it preserves suprema} \\
& \quad (y \text{ ranges over the domain of } f)\} \\
& m.\langle \Sigma_R y :: x \times (m^\sharp \circ f).y\rangle \\
= & \quad \{m \text{ is a lower adjoint, so preserves suprema}\} \\
& \langle \Sigma_{m.R} y :: m.(x \times (m^\sharp \circ f).y)\rangle \\
= & \quad \{m \text{ is compositional}\} \\
& \langle \Sigma_{m.R} y :: m.x \otimes m.((m^\sharp \circ f).y)\rangle \\
= & \quad \{f \text{ has range } m.R; \text{ that is, } f.y = m.z \text{ for some } z; \\
& \quad \text{So,} \\
& \quad\quad m.((m^\sharp \circ f).y) \\
& \quad\quad = \quad \{\text{definition of composition, } f.y = m.z\} \\
& \quad\quad (m \circ m^\sharp \circ m).z \\
& \quad\quad = \quad \{\text{unity of opposites}\} \\
& \quad\quad m.z \\
& \quad\quad = \quad \{f.y = m.z\} \\
& \quad\quad f.y\} \\
& \langle \Sigma_{m.R} y :: m.x \otimes f.y\rangle.
\end{array}
$$

The proof that product-on-the-right preserves suprema is entirely symmetrical. Finally, that m is a regular homomorphism is clear from the definitions. □

Another circumstance in which the conditions of Theorem 5.4 can be relaxed is when the measure is an "extension" to a power-set algebra of a monoid homomorphism. The precise property is as follows.

Theorem 6.3 (Monoidal extensions). *Suppose that* $(A, \cdot, 1)$ *is a monoid and that* $\mathcal{R}$ *is a regular algebra. Suppose* m *is a function with domain* A *and range the carrier of* $\mathcal{R}$*. Consider the power-set algebra* $(2^A, \cdot, \cup, \subseteq, \phi, \{1\})$ *as defined in Theorem 4.14. Define* $\hat{m}$*, the* extension *of* m *to subsets of* A *(elements of* 2^A*), by*

$$\hat{m}.S = \langle \Sigma x : x \in S : m.x\rangle.$$

Then, $\hat{m}$ *is a regular homomorphism equivales* m *is a monoid homomorphism.*

Proof. There are two parts to the proof: showing that if $\hat{m}$ is a regular homomorphism then m is a monoid homomorphism, and its converse. That m is a monoid homomorphism if $\hat{m}$ is a regular homomorphism follows from the fact that $\hat{m}$ is (by definition) a monoid homomorphism and instantiating the requirements on its being a monoid homomorphism in the case that the sets are singleton sets (i.e., have exactly one element).

For the converse, we have to show that the extended function $\hat{m}$ is a monoid homomorphism and that it is a lower adjoint. The details are omitted here. □

We consider several examples.

Example 6.4 (*Test for empty*). Suppose we wish to determine whether a language is empty or not. Consider the regular algebra $\mathbb{B}^D$ (Example 4.4). Define the measure m of a word to be false. Then the extension $\hat{m}$ of m to sets of words tests whether a language is empty or not. Specifically, by definition,

$$\hat{m}.S \equiv \langle \forall u : u \in S : \mathsf{false} \rangle$$

That is,

$$\hat{m}.S \equiv S = \phi.$$

The measure m is clearly a monoid homomorphism and so $\hat{m}$ is a regular homomorphism.

Example 6.5 (*Membership*). We return to the membership problem discussed in Section 3.3. Consider the regular algebra $\mathbb{B}$ (Example 4.4). Given a word X, define the measure m of a word u to be $u = X$. Then the extension, $\hat{m}$, of m to sets of words tests for membership of X in the set. That $\hat{m}$ is a regular homomorphism equivales m is a monoid homomorphism. But

$$\begin{array}{ll} & m \text{ is a monoid homomorphism} \\ = & \{\text{definition}\} \\ & \varepsilon = X \wedge \langle \forall u, v :: u{\cdot}v = X \equiv u = X \wedge v = X \rangle \\ = & \{\text{properties of strings}\} \\ & \varepsilon = X. \end{array}$$

So, the only example of a membership test on sets of words that is a regular homomorphism is the so-called *nullability* test: the test whether the empty word is in the set.

Example 6.6 (*First sets, continued*). Let k be a natural number. The construction of parsers typically involves computing the $FIRST_k$ sets of certain languages. The $FIRST_k$ set of language S is defined by

$$FIRST_k S = \langle \cup u : u \in S : \{F_k u\} \rangle,$$

where F_k is the function that truncates a word to its first k symbols. (See example 4.17.) $FIRST_k$ is the extension to sets of the measure $\langle u :: \{F_k u\} \rangle$ on words. (Conventionally, the right side of the above equation would be written $\{F_k u \mid u \in S\}$, obscuring the nature of the extension; the quantifier notation makes it very clear.)

Defining the product $x \times y$ of two strings each of length at most k as we did in Example 4.17, it is clear that $F_k(x{\cdot}y) = F_k x \times F_k y$. It follows that $\langle u :: \{F_k u\}\rangle$ is a monoid homomorphism. Hence, $FIRST_k$ is a regular homomorphism.

Theorem 6.7 (Vectorial extensions). *Let $\mathcal{R}$ and $\mathcal{S}$ be regular algebras with carrier sets R and S, respectively. Suppose I is a set. Suppose $m \in R \leftarrow S$ is a regular homomorphism. Define the extension $\hat{m}$ of m to the vector algebras $\mathcal{R} \leftarrow I$ and $\mathcal{S} \leftarrow I$ by, for all $i \in I$ and $f \in S \leftarrow I$,*

$$(\hat{m}.f).i = m.(f.i).$$

Then $\hat{m}$ is a regular homomorphism.

Proof. Straightforward application of the definitions. □

The next two examples involve graph algebras.

Example 6.8 (*Shortest paths*). Finding shortest paths through a labelled graph is a good example of the use of the lexicographic combination of algebras.

Given a graph, a *path* through the graph from node s to node t, of *edge length* n, is a finite sequence of nodes $x_0, x_1, \ldots, x_n$ such that $s = x_0$ and $t = x_n$ and, for each i, $0 \leq i < n$, there is an edge in the graph from x_i to x_{i+1}.

Suppose the edge labels in a graph are numbers, representing distances or costs. A measure m on paths is defined to be the sum of the edge labels comprising the path. This is a monoid homomorphism, by definition. Its extension $\hat{m}$ to sets of paths connecting a given pair of nodes is thus a regular homomorphism, thanks to Theorems 6.7 and 6.3.

As is well-known, the function from pairs of nodes s and t to the set of all paths from s to t in a graph is determined by a context-free grammar (in fact, a regular grammar[6]). So, by the fusion theorem (Theorem 5.4), the function from pairs of nodes s and t to the cost of a least-cost path from s to t is given by taking the least fixed point of the interpretation of the grammar in the cost algebra. However, this only gives information about the "cost" of paths; most often, it is required to determine a path itself having least cost. The most effective way to do this is to compute, for each pair of nodes s and t, the first node following s on a least-cost path from s to t.

Formally, we define the measure m of a path $x_0, x_1, \ldots, x_n$ to be an element of the lexicographic combination of two regular algebras $\mathcal{R}_1$ and $\mathcal{R}_2$. The algebra $\mathcal{R}_1$ is the minimum-cost algebra of Example 4.5. The second algebra is the first-set algebra of Example 4.17. The alphabet T is the set of nodes of the graph. The first component of the measure of path $x_0, x_1, \ldots, x_n$ is the sum of the labels of the edges comprising the path; the second component is $\{\varepsilon\}$ if $n = 0$, and $\{x_1\}$, if $1 \leq n$. That is, the measure of a path is the pair comprising the cost of the path and the node immediately following the starting node on the path.

If least-cost paths are computed using this algebra, one determines both costs and the set of "first-step" nodes on least-cost paths. For a given pair of nodes s and t, a "first-step" node

[6] Each terminal symbol in the grammar is the name of a node in the graph. The set of non-terminals comprises two sets, $A_{s,t}$ and $B_{s,t}$, say, where s and t range over nodes of the graph. For each node s, there is a production $A_{s,s} ::= \varepsilon$; for each edge in the graph from node s to node t, and for each node u, there is a production $A_{s,u} ::= t A_{t,u}$; finally, for each pair s,t, there is a production $B_{s,t} ::= s A_{s,t}$. The language generated by $B_{s,t}$ is the set of all paths from node s to node t in the graph.

is a node that immediately follows s on a least-cost path to node t. Theorem 4.10 justifies the use of, for example, the generic Roy–Warshall–Floyd algorithm [33,39,15] (that is, Gauss–Jordan elimination [5]) for this purpose. On the other hand, since the elements of a bottleneck algebra (Example 4.6) are not cancellative, it is not valid to compute bottleneck routes in this way. (In layman's terms, if someone asks you for the shortest route from A to B, it suffices to point out which direction to go first, and get them to ask again at the next junction. However, if the driver of a high load asks you for a route from A to B that maximises the minimum-height bridge to be negotiated, it's inadvisable to respond in the same way—if everyone does so, the driver may go around in circles!)

Example 6.9 generalises Example 6.5.

Example 6.9 (*General context-free parsing*). The general parsing algorithm invented by Cocke, Younger and Kasami exploits a regular homomorphism. (See [2, p. 332] for references to the origin of the Cocke–Younger–Kasami parsing algorithm.)

Let X be a given word, and let $\#X$ be the length of X. The problem is to determine whether X—the input string—is in a language L given by some context-free grammar.

We use X to define a measure on words and then we extend the measure to sets. The measure of word u is a graph of Booleans that determines which segments of X are equal to u. Specifically, let us index the symbols of X from 0 onwards. The edge relation of the graph is the set of pairs (i, j) such that $0 \leq i \leq j \leq \#X$ and will be denoted by *seg*. Note that this is a reflexive, transitive relation. For brevity, we omit this constraint on i and j from now on.

Let $X[i..j)$ denote the segment of word X beginning at index i and ending at index $j-1$. In particular, $X[i..i)$ is the empty word, and $X = X[0..\#X)$. Now, let

$$e.u = \langle i, j :: u = X[i..j)\rangle.$$

This defines $e.u$ to be a boolean graph. The extension of the measure e to sets is

$$\hat{e}.S = \langle i, j :: \langle \exists u : u \in S : u = X[i..j)\rangle\rangle.$$

So

$$0\langle \hat{e}.S\rangle \#X \equiv X \in S.$$

Given a context-free grammar with non-terminal set N, we apply Theorem 5.4 with $\mathcal{R}$ instantiated to the algebra of languages, m instantiated to $\hat{e}$, and $\mathcal{S}$ instantiated to the graph algebra $M_{seg}\mathbb{B}$. In this way, a fixed-point equation (a set of "simultaneous equations" in order of $(\#X)^2 \times |N|$ unknowns) is obtained, whose least solution gives the desired membership test. (Typically, the values of the unknowns are determined in increasing order of the length of the segment; the total number of terms in the fixed-point equation is of the order of $(\#X)^3$, giving an algorithm that is cubic in the length of the input string.)

Crucial to the correctness of the Cocke–Younger–Kasami algorithm is that $\hat{e}$ is a regular homomorphism; by Theorem 6.3, this is the requirement that e is a monoid homomorphism. This is proved as follows. Clearly $e.\varepsilon$ is the unit graph $\mathbf{1}$. Also,

$$\begin{array}{ll} & e.u \times e.v \\ = & \{\text{definition of } e\} \\ & \langle i, j :: u = X[i..j)\rangle \times \langle i, j :: v = X[i..j)\rangle \\ = & \{\text{definition of graph product in algebra } M_{seg}\mathbb{B}\} \end{array}$$

$$
\begin{array}{ll}
 & \langle i, j :: \langle \exists k :: u = X[i..k) \wedge v = X[k..j) \rangle \rangle \\
= & \quad \{\text{word calculus}\} \\
 & \langle i, j :: u{\cdot}v = X[i..j) \rangle \\
= & \quad \{\text{definition of } e\} \\
 & e.(u{\cdot}v).
\end{array}
$$

Note how the definition of e is immediately suggested by the requirement that it be a monoid homomorphism. In Example 6.5, we observed that the measure on u, $u = X$, is a monoid homomorphism equivales $\varepsilon = X$. Extending the measure to all segments of X is an obvious way of fulfilling the requirement.

The final example is the most complicated, and requires a more complicated justification.

Example 6.10 (*Error repair*). A general technique for error repair when parsing languages is to compute the minimum number of edit operations required to edit the input string into a string in the language being recognised [1]. The technique involves a generalisation of the Cocke–Younger–Kasami algorithm, similar to the generalisation that is made when going from the Roy–Warshall transitive-closure algorithm to Floyd's all-shortest-paths algorithm.

Let X be a given word (the input string) and let $\#X$ be the length of X. As in Example 6.9, we use X to define a measure on words and then we extend the measure to sets. The measure of word u is a graph of numbers that determines how many edit operations are required to transform each segment of X to the word u. Transforming one word to another involves a sequence of primitive edit operations. Initially the input index, i, is set to 0; the edit operations scan the input string from left to right, transforming it to the output string. The allowed edit operations and their effect on the input string are

- *Insert*(a). Insert symbol a after the current symbol in the output string.
- *Delete*. Increment the index i.
- *ChangeTo*(a). Increment the index i and add symbol a to the end of the output string.
- *OK*. Copy the symbol at index i of the input to the output. Then increment i.

(We will see that the choice of allowed edit operations is crucial to the correctness of the generalised algorithm.)

Let $dist(u, v)$ denote the minimum number of non-OK edit operations needed to transform word u into word v using a sequence of the above edit operations. Now define

$$d.u = \langle i, j :: dist(X[i..j), u) \rangle.$$

This defines $d.u$ to be a graph of numbers. The numbers, augmented by ∞, form the min-cost regular algebra discussed in Example 4.5. Thus graphs over numbers also form a regular algebra. Taking this as the range algebra, the extension of the measure d to sets is

$$\hat{d}.S = \langle i, j :: \langle \downarrow u : u \in S : dist(X[i..j), u) \rangle \rangle,$$

so that $0\langle \hat{d}.S \rangle \#X$ is the minimum number of edit operations required to repair the word X to a word in S. As in the standard Cocke–Younger–Kasami algorithm (Example 6.9), a fixed-point equation in these edit distances is obtained by applying Theorem 5.4, in this case with m instantiated to $\hat{d}$. (Knuth's algorithm [23] is an appropriate solution method.)

Crucial to the correctness of the generalised Cocke–Younger–Kasami algorithm is that d is compositional. This is proved as follows:

$$
\begin{array}{ll}
 & d.(u{\cdot}v) \\
= & \quad \{\text{definition of } d\} \\
 & \langle i, j :: dist(X[i..j), u{\cdot}v)\rangle \\
= & \quad \{\bullet \text{ property of } dist\} \\
 & \langle i, j :: \langle \downarrow k :: dist(X[i..k), u) + dist(X[k..j), v)\rangle\rangle \\
= & \quad \{\text{definition of graph product}\} \\
 & \langle i, j :: dist(X[i..j), u)\rangle \times \langle i, j :: dist(X[i..j), v)\rangle \\
= & \quad \{\text{definition of } d\} \\
 & d.u \times d.v.
\end{array}
$$

Note that a crucial step in this calculation is the second step

$$
\begin{array}{ll}
 & \langle i, j :: dist(X[i..j), u{\cdot}v)\rangle \\
= & \quad \{\bullet \text{ property of } dist\} \\
 & \langle i, j :: \langle \downarrow k :: dist(X[i..k), u) + dist(X[k..j), v)\rangle\rangle.
\end{array}
$$

This is a non-trivial property of the chosen collection of edit operations.

To see that the property is non-trivial, suppose we extend the set of edit operations to allow the transposition of two adjacent characters. (Transposing characters is a very common error when using a keyboard.) Then, the edit-distance function is not compositional. For example, ($dist$(“ab”, “ba”) is 1—it takes one transposition to transform the word “ab” to the word “ba”—but this is not equal to the minimum of $dist$(“ab”, “b”) + $dist$(ε, “a”), $dist$(“a”, “b”) + $dist$(“b”, “a”) and $dist$(ε, “b”) + $dist$(“ab”, “a”)—as it should be if the function d were to be compositional. Indeed, computing minimal edit distances for context-free languages is very difficult if the possibility of transpositions is included in the analysis.

Note that d is compositional but not a monoid homomorphism. In an algebra of graphs with underlying algebra minimum costs, the (i, j)th entry in the unit graph is ∞ whenever $i \neq j$. The (i, j)th entry in $d.\varepsilon$, on the other hand, is the cost of deleting all the symbols of $X[i..j)$. This is an instance where Theorem 6.2 is needed. The extension $\hat{d}$ of d is indeed a regular homomorphism; its domain is the algebra of languages and its range is the algebra of graphs in the image set of $\hat{d}$.

7. The bound problem

This section is about a novel application of the theory developed above. Formally, the problem is this. Suppose $\mathcal{R}$ is a regular algebra ordered by $\leq$, G is a context-free grammar with sentence symbol S, f is an interpretation in $\mathcal{R}$ of the terminal symbols in G, and k is an element of the carrier set of $\mathcal{R}$. Determine whether, $\mu_{\leq} G_{\mathcal{R},f}.S \leq k$.

A concrete example is the following: suppose w is a word, and G is a context-free grammar; suppose it is required to determine whether at most 2 edit operations are required to transform w to a word in the language generated by G.

A solution to this problem involves two generalisations. First, as in Example 6.10, we generalise to all segments of w and all non-terminals of the grammar G. Second, we generalise "2" to 0, 1 or 2. That is, we determine, for each segment u of w, and each non-terminal A of G, whether u can be edited to a word in the language generated by A in zero, one or two steps. This generalisation is quite obvious; in order to edit w to a word in $X{\cdot}Y$ (for languages X and Y) in at most 2 steps, we need to know which prefixes of w can be edited to a word in X in 0, 1 or 2 steps, and which suffixes of w can be edited to a word in Y in 0, 1 or 2 steps. More generally, if the bound on the number of edit operations is k, we generalise to all bounds $0, 1, 2, \ldots, k$. (This is not an artificial example: an efficient error-repair algorithm can be constructed by combining Knuth's algorithm [23] with Wagner's [38] method of improving the efficiency of Dijkstra's shortest-path algorithm. The details are beyond the scope of this paper.)

A second concrete example is the language-inclusion problem: given a context-free grammar G, calculate whether the language generated by G is a subset of some given language L. This problem arises in program analysis. Indeed, the solution to the language-inclusion problem presented here is due to De Moor [30,35]; our contribution is to show how his solution fits into a general programming methodology.

Two specific examples of the language-inclusion problem have already been discussed in Section 2. The first example is when L is the empty set. The problem in this case is to determine whether a given grammar generates the empty language. The solution to the general language-inclusion problem specialises in this case to the well-known algorithm discussed in Section 2. The second example is when L is $(T{\cdot}T)^*$, where T is some alphabet. The problem is then to determine whether all words in the language generated by G have even length. The solution presented in Section 2, which involved computing simultaneously whether all words in the language generated by G have even length and whether all words in the language generated by G have odd length, is predicted by the general theory developed in this section.

What unifies the bounded error-repair problem and the all-even-length word problem is factorisation. Suppose x is an element of a regular algebra. Call an element y a *factor* of x if there are elements u and v such that $y = u\backslash x/v$. (Left and right division are mutually associative, so it does not matter how the right side is bracketed.) Then, in the algebra of edit distances, where division is subtraction of numbers, the factors of natural number k are the numbers $0, 1, 2, \ldots, k$. In the algebra of languages, the factors of $(T{\cdot}T)^*$ (the set of all even-length words) are $(T{\cdot}T)^*$ itself, $T{\cdot}(T{\cdot}T)^*$ (the set of all odd-length words), T^* (the set of all words) and ϕ (the empty set). The all-even-length word problem for an arbitrary grammar is solved by generalising the problem to determining, for each non-terminal in the grammar, whether or not the language generated is all-even, all-odd, a subset of T^*—which, of course, is true—or empty.[7]

This section discusses the solution of the bound problem in full generality. The key is Conway's theory of the factors [9], and, in particular, the so-called "factor matrix". Section 7.1 explains the relevance of factors, preparing the way for the summary of factor theory in Section 7.2.

[7] The strategy of first eliminating all non-terminals that generate the empty language is suggested by factor theory, but space does not allow us to discuss this in detail. Suffice it to say that, for all X, the "factor matrix" (introduced below) of a factor of X is a submatrix of the factor matrix of X [3]. All languages different from T^* and ϕ include the factor matrix of ϕ as a proper submatrix. Exploitation of this structure gives the aforementioned decomposition.

This is not the first time that factor theory has been used in the solution of problems of this nature; it is used implicitly in solutions to the so-called "pattern-matching" problem: Given a word p (the "pattern") and a word x (the "text"), it is required to determine all occurrences of the pattern in the text. Formally, for each prefix w of x, it is required to determine whether p is a suffix of w. That is calculate

$$\{w\} \subseteq T^* \cdot \{p\}.$$

The method we discuss for solving the general bound problem specialises to the Knuth–Morris–Pratt pattern-matching algorithm [22]. That there is a connection between Conway's theory and pattern matching was observed by the author when the Knuth–Morris–Pratt algorithm was first published [6], but the connection was not properly understood. The results of this section clarify the connection completely.

7.1. Bounds as measures

Recall the assumptions made at the beginning of Section 7: $\mathcal{R} = (R, \times, +, \leq, 0, 1)$ is a regular algebra; G is a context-free grammar; f is an interpretation in $\mathcal{R}$ of the terminal symbols in G; and k is an element of the carrier set of $\mathcal{R}$. The problem is to find some generalisation of the yes–no question (for given non-terminal A)

$$\mu_{\leq} G_{\mathcal{R},f}.A \leq k,$$

that can be expressed as a fixed point computation

$$\mu_{\sqsubseteq} G_{\mathcal{R}',f'}.A$$

for some regular algebra $\mathcal{R}' = (R', \otimes, \oplus, \sqsubseteq, 0', 1')$, and some interpretation f' of the terminal symbols of G.

We seek a generalisation such that the carrier R' of $\mathcal{R}'$ is a tuple of booleans, organised as a graph or a vector. In order to see how to order the components, we observe that, for all X and all booleans b,

$$X \leq k \Leftarrow b \;\equiv\; X \leq \mathsf{if}\ b \to k \ []\ \neg b \to \top\ \mathsf{fi}. \tag{17}$$

(Once again, we use $\top$ for the largest element of R.) So, $\langle X :: X \leq k\rangle$ connects the poset $(R, \leq)$ and the booleans ordered by $\Leftarrow$. The addition operator corresponding to this ordering of the booleans is conjunction. So, the operator $\oplus$ is conjunction extended pointwise to a tuple of booleans.

Now, we have to identify a suitable product operator. The driving force behind the generalisation is that the function $\langle X :: X \leq k\rangle$ is not compositional in general. We do have, for all languages u and v,

$$u \cdot v \subseteq \phi \equiv u \subseteq \phi \vee v \subseteq \phi,$$

so that, in this case, we can take $\mathbb{B}^D$ as the regular algebra. Testing for emptiness of languages is one case where the theory applies directly. At the other extreme, we have (in any regular algebra)

$$u \cdot v \leq \top \equiv u \leq \top \vee v \leq \top,$$

but this is a case where the bound problem is trivial. In other cases, the appropriate generalisation is not at all obvious.

The equivalences $u \cdot v \leq k \equiv u \leq k/v$ and $u \cdot v \leq k \equiv v \leq u \backslash k$ suggest that factors have a crucial role in the generalisation. This is indeed the case. The next subsection reviews the salient properties.

7.2. Conway's factor matrix

This subsection is about Conway's factor theory. Conway developed the theory only in the context of the algebra of regular languages. However, much of his theory is applicable to regular algebras in general. For this reason, we present the theory in this more general context. Only at the point where specific properties of regular languages are used do we specialise to this application.

For brevity, only the properties needed to solve the bound problem are presented here. It is an interesting exercise—left to the reader—to reformulate all of Conway's theory in a calculational style.

Throughout this section, k denotes a fixed element of R (the carrier of a regular algebra). Variables u, v, w, x, y and z range over R.

Recall that a *factor* of k is any element of R that can be expressed in the form $x\backslash k/y$ for some x and y. A *left factor* of k is an element of the form k/y for some y, and a *right factor* of k is an element of the form $x\backslash k$ for some x.

Define the functions $\triangleleft$ and $\triangleright$ by

$$x\triangleleft = k/x, \tag{18}$$

$$x\triangleright = x\backslash k. \tag{19}$$

A left factor of k is an element of R that equals $x\triangleleft$ for some x, and a right factor of k is an element of R that equals $x\triangleright$ for some x. Now, the functions $\triangleleft$ and $\triangleright$ are Galois-connected, inverse functions. First, they are Galois-connected as follows:

$$x \leq y\triangleleft \;\equiv\; y \leq x\triangleright. \tag{20}$$

The general properties of Galois connections predict that $\triangleleft$ and $\triangleright$ are inverse functions:

$$x\triangleleft\triangleright\triangleleft = x\triangleleft, \tag{21}$$

$$x\triangleright\triangleleft\triangleright = x\triangleright. \tag{22}$$

A further property is that k is both a left and a right factor of itself:

$$k\triangleleft\triangleright = k = k\triangleright\triangleleft. \tag{23}$$

Properties (21) and (22) are instances of the property that the poset of left factors is isomorphic to the poset of right factors—see the unity-of-opposites theorem, Theorem 3.4. To illustrate calculations with factors, we show how to establish (23):

$$\begin{array}{ll} & k\triangleright\triangleleft = k \\ = & \{\text{antisymmetry}\} \\ & k\triangleright\triangleleft \leq k \wedge k \leq k\triangleright\triangleleft \\ = & \{(20), \text{reflexivity of } \leq\} \\ & k\triangleright\triangleleft \leq k \\ = & \{\text{definitions of } \triangleleft \text{ and } \triangleright\text{: (18) and (19)}\} \\ & k/(k\backslash k) \leq k \\ = & \{k = k/1\} \\ & k/(k\backslash k) \leq k/1 \\ \Leftarrow & \{\text{antimonotonicity: } k/u \leq k/v \Leftarrow v \leq u\} \end{array}$$

$$
\begin{array}{ll}
& 1 \leq k \backslash k \\
= & \quad \{\text{factors: (16), 1 is unit of product}\} \\
& \mathsf{true}.
\end{array}
$$

The proof that $k\triangleleft\triangleright = k$ is symmetrical.

Let $\mathcal{L}$ denote the set of all left factors of k. Define the *factor matrix* of k to be the binary operator $\backslash$ restricted to $\mathcal{L} \times \mathcal{L}$. That is, entries in the matrix take the form $i \backslash j$ where i and j are left factors of k. By definition of a left factor and a factor, all entries in the matrix are factors of k. Crucial to our goal is that k is itself an entry:

$$k = k\triangleleft \backslash k \triangleright\triangleleft, \tag{24}$$

and, for all x and y and all u and v,

$$x \times y \leq u\triangleleft \backslash v\triangleleft \;\equiv\; \langle \exists w :: x \leq u\triangleleft \backslash w\triangleleft \wedge y \leq w\triangleleft \backslash v\triangleleft \rangle. \tag{25}$$

7.3. The generalisation

We are now in a position to generalise the bound problem. As in Examples 6.9 and 6.10, we define the generalised measure to be a graph.

Letting i and j range over left factors of k, we define the measure m by

$$m.u = \langle i, j :: u \leq i \backslash j \rangle.$$

This defines $m.u$ to be a boolean graph of dimension $\mathcal{L} \times \mathcal{L}$. That is, m is a function from R to $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$. Since k is an entry in the factor matrix (see (24)), we have

$$k\triangleleft \langle m.u \rangle k \equiv u \leq k.$$

That is, the $(k\triangleleft, k)$th entry of the graph is $u \leq k$.

As discussed above, we order $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$ by the follows-from relation, extended pointwise. That is, for graphs $\mathbf{M}$ and $\mathbf{N}$, we define $\mathbf{M} \Leftarrow \mathbf{N}$ by

$$\mathbf{M} \Leftarrow \mathbf{N} \equiv \langle \forall i, j :: i \langle \mathbf{M} \rangle j \Leftarrow i \langle \mathbf{N} \rangle j \rangle.$$

This ordering ensures that m is the lower adjoint in a connection of the poset $(R, \leq)$ and $(M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}, \Leftarrow)$; the connection is the pointwise extension of (17). Note that the supremum $\mathbf{M} + \mathbf{N}$ of $\mathbf{M}$ and $\mathbf{N}$ corresponding to this ordering is componentwise conjunction of elements. That is,

$$\mathbf{M} + \mathbf{N} = \langle i, j :: i \langle \mathbf{M} \rangle j \wedge i \langle \mathbf{N} \rangle j \rangle.$$

The product of two graphs $\mathbf{M}$ and $\mathbf{N}$ in $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$ is defined by

$$\mathbf{M} \times \mathbf{N} = \langle i, j :: \langle \exists h :: i \langle \mathbf{M} \rangle h \wedge h \langle \mathbf{N} \rangle j \rangle \rangle.$$

This has the effect that m is compositional:

$$
\begin{array}{ll}
& m.u \times m.v \\
= & \quad \{\text{definition of } m\} \\
& \langle i, j :: u \leq i \backslash j \rangle \times \langle i, j :: v \leq i \backslash j \rangle \\
= & \quad \{\text{definition of graph product in algebra } M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}\} \\
& \langle i, j :: \langle \exists h :: u \leq i \backslash h \wedge v \leq h \backslash j \rangle \rangle
\end{array}
$$

$$
\begin{array}{ll}
= & \{(25)\} \\
& \langle i, j :: u \times v \leq i \backslash j \rangle \\
= & \{\text{definition of } m\} \\
& m.(u \times v).
\end{array}
$$

Note, however, that $m.1$ is not necessarily the unit of $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$. We do have that, for all left factors i of k, $1 \leq i \backslash i$—that is, the (i, i)th entry of $m.1$ is true. But, for unequal left factors i and j, it is not necessarily the case that $1 \leq i \backslash j$ is false. Also, product in $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$ does *not* admit division everywhere with respect to the ordering of graphs. Indeed, product does not always distribute through binary addition in $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$. For example, assuming $\mathcal{L}$ has two elements, and abbreviating true to t, and false to f, we have

$$
\begin{bmatrix} t & t \\ f & f \end{bmatrix} \times \begin{bmatrix} t & f \\ f & f \end{bmatrix} + \begin{bmatrix} t & t \\ f & f \end{bmatrix} \times \begin{bmatrix} f & f \\ t & f \end{bmatrix} = \begin{bmatrix} t & f \\ f & f \end{bmatrix},
$$

whereas

$$
\begin{bmatrix} t & t \\ f & f \end{bmatrix} \times \left(\begin{bmatrix} t & f \\ f & f \end{bmatrix} + \begin{bmatrix} f & f \\ t & f \end{bmatrix} \right) = \begin{bmatrix} f & f \\ f & f \end{bmatrix}.
$$

This is where Theorem 6.2 is needed once again. Although $M_{\mathcal{L}\times\mathcal{L}}\mathbb{B}$ is *not* the carrier of a regular algebra with the above definitions of the ordering relation and product operation, this *is* the case for the image of m. It takes some effort to verify that the definition of the addition of graphs $\mathbf{M}$ and $\mathbf{N}$ is indeed pointwise conjunction, as claimed above: according to the range-algebra theorem (Theorem 6.2), addition should be defined by

$$\mathbf{M} \oplus \mathbf{N} = \langle i, j :: m^\sharp.\mathbf{M} \leq i \backslash j \wedge m^\sharp.\mathbf{N} \leq i \backslash j \rangle,$$

where, for all graphs $\mathbf{X}$,

$$m^\sharp.\mathbf{X} = \langle \Pi i, j : i \langle \mathbf{X} \rangle j : i \backslash j \rangle.$$

(Determining the definition of $m^\sharp$ requires some straightforward calculation.) Fortunately, using the fact that $\top$ is a factor of k, whatever the value of k, the simplification to componentwise conjunction can be made.

Using $\mathcal{S}$ to denote the range algebra, the conclusion we obtain by applying Theorem 5.4 is that, for all non-terminals A in the grammar G,

$$\langle i, j :: \mu_{\leq} G_{R,f}.A \leq i \backslash j \rangle \equiv \mu_{\Leftarrow} G_{\mathcal{S},(m \circ f)}.A.$$

A least fixed point with respect to the follows-from relation is a greatest fixed point with respect to implication, which is the conventional way to order the booleans. So, in words, the general bound problem is solved by determining the greatest fixed point of a system of equations, having the same structure as the grammar G, in boolean matrices.

8. Programming methodology

This paper makes several novel contributions. We have proposed a novel formulation of a regular algebra, emphasising the existence of factors, and provided much justification for its significance: on the theoretical side, we have shown how complex regular algebras can be built in a variety of ways (including: by pointwise tupling, by forming graphs, and by forming range algebras); on the practical side, we have shown how these constructions arise in several challenging applications. The main contribution, however, is to the

development of programming methodology. A commonly occurring element of algorithm development is how to generalise a given problem to one that can be solved using known algorithms for fixed-point computation; we have shown how the existence of "factors" and a "compositionality property" is crucial to finding the appropriate generalisation. Without this insight, the examples we have considered are difficult to solve.

The paper has deliberately avoided discussing algorithms for fixed-point computation. This is an important separation of concerns. There is ample choice in the literature of fixed-point algorithms, but which algorithm is chosen depends critically on the structure of the fixed-point equation that has to be solved. Separating the process of constructing the equation from solving it is imperative for effectiveness. Also, for some instances of the example problems, it is possible to construct a fixed-point characterisation of the problem without necessarily knowing how to solve the fixed-point equation. This is the case for the language-inclusion problem, which is an instance of the general bound problem discussed in Section 7. Conway proves that the factor matrix of a given language is finite exactly when the language is regular. The boolean matrices defined in Section 7.3 thus have finite dimension exactly when the problem is to determine the inclusion of a context-free language in a *regular* language. A corollary of Section 7.3 is that it is always possible to express the inclusion of the language generated by a context-free grammar in any other language in terms of a fixed-point equation; but, in the case of non-regular languages, this corresponds to a system of equations with an infinite number of unknowns. This precludes the use of, for example, simple iterative techniques for computing fixed points, which typically assume that the solution domain has finite size.

A number of avenues for further research are worth highlighting. Very little is known about the structure of the factor matrix, even for regular languages. There is the potential for making substantial efficiency improvements in language-inclusion algorithms if more is known. Also, nothing is known about the structure of the factor matrices of context-free languages. (Conway did not even define the matrix in this case, although, as this paper demonstrates, it is well-defined.) It may be that special-purpose language-inclusion algorithms can be developed for particular classes of languages other than the regular languages. Other problems, that have hitherto been seen as difficult or impossible to solve efficiently, may now become more tractable. Alternatively, it may be possible to understand better why certain problems are intractable. In any case, Conway's theory of factors, which appears to have been almost entirely ignored since its publication in 1971, deserves to be better known and understood than it is at present.

Acknowledgements

My thanks to Kevin Backhouse, Diethard Michaelis, Bernhard Möller and the anonymous referees for many comments and suggestions.

References

[1] A.V. Aho, T.G. Peterson, A minimum-distance error-correcting parser for context-free languages, SIAM J. Comput., 1 (1972) 305–312.

[2] A.V. Aho, J.D. Ullman, The Theory of Parsing, Translation and Compiling, Series in Automatic Computation, vol. 1, Prentice-Hall, Englewood, 1972.

[3] R.C. Backhouse, Closure algorithms and the star-height problem of regular languages, Ph.D. Thesis, University of London, 1975.
[4] K. Backhouse, R. Backhouse, Safety of abstract interpretations for free, via logical relations and Galois connections, Sci. Comput. Programming 51 (1–2) (2004) 153–196.
[5] R.C. Backhouse, B.A. Carré, Regular algebra applied to path-finding problems, J. Inst. Math. Appl. 15 (1975) 161–186.
[6] R.C. Backhouse, R.K. Lutz, Factor graphs, failure functions and bi-trees, in: A. Salomaa, M. Steinby (Eds.), Fourth Colloquium on Automata, Languages and Programming, LNCS, vol. 52, Springer-Verlag, Berlin, 1977, pp. 61–75.
[7] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 238–252.
[8] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, TX, January 1979, pp. 269–282.
[9] J.H. Conway, Regular Algebra and Finite Machines, Chapman and Hall, London, 1971.
[10] E.W. Dijkstra, A note on two problems in connexion with graphs, Numer. Math. 1 (1959) 269–271.
[11] R.P. Dilworth, Non-commutative residuated lattices, Trans. Amer. Math. Soc. 46 (1939) 426–444.
[12] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, first ed., Cambridge Mathematical Textbooks, Cambridge University Press, Cambridge, 1990.
[13] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Texts and Monographs in Computer Science, Springer-Verlag, Berlin, 1990.
[14] W.H.J. Feijen, L. Bijlsma, Exercises in formula manipulation, in: E.W. Dijkstra (Ed.), Formal Development of Programs and Proofs, Addison-Wesley Publ. Co., Reading, MA, 1990, pp. 139–158.
[15] R.W. Floyd, Algorithm 97, Shortest Path, Comm. ACM 5 (6) (1962) 345.
[16] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, D.S. Scott, A Compendium of Continuous Lattices, Springer-Verlag, Berlin, 1980.
[17] C.A.R. Hoare, J. He, The weakest prespecification, Fund. Inform. 9 (51–84) (1986) 217–252.
[18] C.A.R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall International, Englewood, 1998.
[19] J. Hartmanis, R.E. Stearns, Pair algebras and their application to automata theory, Inform. Control 7 (4) (1964) 485–507.
[20] J. Jeuring, S.D. Swierstra, Bottom-up grammar analysis—a functional formulation, in: D. Sannella (Ed.), Proceedings Programming Languages and Systems, ESOP'94, LNCS, vol. 788, 1994, pp. 317–332.
[21] J. Jeuring, S.D. Swierstra, Constructing functional programs for grammar analysis problems, in: S.P. Jones (Ed.), Proceedings Functional Programming Languages and Computer Architecture, FPCA'95, June 1995.
[22] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (1977) 325–350.
[23] D.E. Knuth, A generalization of Dijkstra's shortest path algorithm, Inform. Process. Lett. 6 (1) (1977) 1–5.
[24] Dexter Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, in: Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, IEEE Society Press, 1991, pp. 214–225.
[25] J. Lambek, The mathematics of sentence structure, Amer. Math. Monthly 65 (1958) 154–170.
[26] J. Lambek, The influence of Heraclitus on modern mathematics, in: J. Agassi, R.S. Cohen (Eds.), Scientific Philosophy Today, D. Reidel Publishing Co., Dordrecht, 1981, pp. 111–121.
[27] J. Lambek, Some Galois connections in elementary number theory, J. Number Theory 47 (3) (1994) 371–377.
[28] J.-L. Lassez, V.L. Nguyen, E.A. Sonenburg, Fixed point theorems and semantics: a folk tale, Inform. Process. Lett. 14 (3) (1982) 112–116.
[29] J. Lambek, P.J. Scott, Introduction to Higher Order Categorical Logic, Studies in Cambridge University Press, vol. 7, Cambridge University Press, Cambridge, 1986.
[30] O. de Moor, S. Drape, D. Lacey, G. Sittampalam, Incremental program analysis via language factors, 2002. Available from: <http://web.comlab.ox.ac.uk/work/oege.de.moor/pubs.htm>.
[31] A. Melton, D.A. Schmidt, G.E. Strecker, Galois connections and computer science applications, in: D. Pitt, S. Abramsky, A. Poigné, D. Rydeheard (Eds.), Category Theory and Computer Programming, Lecture Notes in Computer Science, vol. 240, Springer-Verlag, Berlin, 1986, pp. 299–312.
[32] O. Ore, Galois connexions, Trans. Amer. Math. Soc. 55 (1944) 493–513.
[33] B. Roy, Transitivité et connexité, C.R. Acad. Sci. 249 (1959) 216.
[34] J. Schmidt, Beiträge zur Filtertheorie. II, Math. Nachr. 10 (1953) 197–232.

[35] G. Sittampalam, O. de Moor, K.F. Larssen, Incremental execution of transformation specifications, in: POPL'04, 2004.
[36] R.E. Tarjan, A unified approach to path problems, J. Assoc. Comp. Mach. (1981) 577–593.
[37] B. von Karger, A calculational approach to reactive systems, Sci. Comput. Programming (2000) 139–161.
[38] R.A. Wagner, A shortest path algorithm for edge-sparse graphs, J. Assoc. Comp. Mach. 23 (1976) 50–57.
[39] S. Warshall, A theorem on boolean matrices, J. ACM 9 (1962) 11–12.