

# Universal Regular Path Queries

Oege de Moor, David Lacey and Eric Van Wyk

November 5, 2002

## Abstract

Given are a directed edge-labelled graph  $G$  with a distinguished node  $n_0$ , and a regular expression  $P$  which may contain variables. We wish to compute all substitutions  $\phi$  (of symbols for variables), together with all nodes  $n$  such that all paths  $n_0 \rightarrow n$  are in  $\phi(P)$ . We derive an algorithm for this problem using relational algebra, and show how it may be implemented in Prolog. The motivation for the problem derives from a declarative framework for specifying compiler optimisations.

## 1 Bob Paige and IFIP WG 2.1

Bob Paige was a long-standing member of IFIP Working Group 2.1 on Algorithmic Languages and Calculi. In recent years, the main aim of this group has been to investigate the derivation of algorithms from specifications by program transformation. Already in the mid-eighties, Bob was way ahead of the pack: instead of applying transformational techniques to well-worn examples, he was applying his theories of program transformation to new problems, and discovering new algorithms [16, 48, 52]. The secret of his success lay partly in his insistence on the study of general algorithm design strategies (in particular finite differencing [47, 51] and data structure selection [19]) rather than the study of tiny derivational steps that some of the working group had focussed on.

His success in the systematic discovery of new algorithms was in itself remarkable, but perhaps even more impressive was the fact that he succeeded in automating his derivations in the APTS system [20, 49]. This provided the ultimate proof that he had succeeded in identifying deep principles in algorithm design: an automated derivation leaves no room for cheating. The mechanism for applying transformations in the APTS system was that of rewrite rules, and a fast pattern matching algorithm (invented, naturally, by Bob himself [21]) provided the basic engine. The rules could have side conditions expressed as queries on a ‘database’ of facts about the program under consideration. The facts in the database could be any result of program analyses.

A major difficulty, which we repeatedly discussed with Bob, was to express the queries in a declarative meta-language, and to maintain the database incrementally as the rules are applied [50]. These remain major problems in the field of automated program transformation, and the present paper is a small contribution towards solving them. We follow Bob’s example in our attempt to derive the relevant algorithm itself in a transformational fashion.

Bob’s influence on working group 2.1 has been immense, and he provided much inspiration for its members. He will be much missed. His work, however, lives on in the current research of the group, and this paper is but a small example of that.

## 1.1 Specifying compiler optimisations

Several of the phases of a compiler can be generated from declarative specifications: for instance, there are commonly used tools for syntax analysis (lex and yacc), for semantic analysis (attribute grammar systems such as FNC-2 and the SG [39, 56]) and also for instruction selection (IBURG [33]). There is however no such widely accepted tool for the declarative specification of optimising transformations, although there have been many proposals, e.g. [4, 10, 17, 25, 31, 41, 43, 61–63].

In a traditional compiler, the optimising transformations are typically performed as rewrites on the flow graph [2, 3, 46]. The difficulty lies in the specification of the necessary side conditions. For example, consider *constant propagation*. In essence, it is simply the rewrite rule

$$x := y \Rightarrow x := c$$

where  $y$  is a program variable, and  $c$  a constant. The rule is applicable only if on all execution paths to the assignment  $x := y$ , the last modification to  $y$  was an assignment of the form  $y := c$ . How can one conveniently express this condition?

Let us assume that edges in the flow graph are labelled with atomic propositions about their target statement. For example, each edge to a node that is of the form  $y := E$  (or otherwise modifies  $y$ ) would be labelled with the proposition  $def(y)$ . We can think of a path in the flow graph as a sequence of edges, or alternatively as a sequence of edge-labelling propositions. The side condition of constant propagation then becomes the requirement that all paths from program entry to  $x := y$  are in the regular language

$$P = (-)* ; y := c ; (\neg def(y))* ; x := y$$

Here  $(-)$  denotes a wildcard,  $(;)$  is sequential composition and  $(-)*$  is the usual closure operation. The symbols  $y$ ,  $c$  and  $x$  are pattern variables: we seek to compute substitutions  $\phi$  that instantiate these variables, coupled with nodes  $n$  such that all paths to  $n$  are in the regular language  $\phi(P)$ . In Figure 1, an example is shown that has two solutions, namely  $(\{y \rightarrow q, c \rightarrow 0, x \rightarrow s\}, n_1)$  and  $(\{y \rightarrow q, c \rightarrow 0, x \rightarrow t\}, n_2)$ . Note the difference between *pattern* variables  $(x, y, c)$  which appear in the regular expression and *program* variables  $(p, q, r)$  which appear in the flow graph. Also notice that one of the paths in the solution, from  $n_0$  to  $n_2$ , cannot occur in actual program runs.

As another example consider common sub-expression elimination. Here the rewrite is equally simple, we replace an assignment to a complex expression with an assignment to a variable

$$x := y + z \Rightarrow x := w$$

The side condition must say that every path to the assignment  $x := y + z$  passes through the assignment  $w := y + z$  and nothing must interfere between these two assignments. The

```

// entry  $n_0$ 
if  $p > 3$ 
then   $q := 0;$ 
         $r := 1$ 
else   $r := 2;$ 
         $q := 0;$ 
 $s := q;$  // node  $n_1$ 
if  $q > 3$ 
then   $t := q$  // node  $n_2$ 
else   $t := r$ 

```

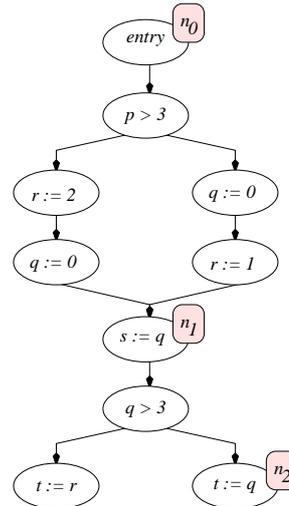


Figure 1: An example flow graph

pattern that all paths from the entry to the  $x := y + z$  node must match is

$$P_{cse} = (-)*; w := y + z; (\neg def(w) \wedge \neg def(y) \wedge \neg def(z))*; x := y + z$$

Furthermore, the variable  $w$  should not be equal to  $y$  or  $z$ . That additional requirement could have been encoded in the above formula, but we prefer to treat it separately for expository reasons.

It is now apparent that our initial description of the problem was somewhat oversimplified, because in general we are interested in many different facts about edges in the flow graph, not just the atomic propositions such as  $def(x)$ . We therefore need to consider edges to be labelled by *composite* propositions that are true of the target node. Composite propositions are built from atomic propositions, and the usual logical connectives ( $\neg, \wedge, \vee$ ). A path in the flow graph thus corresponds to a sequence  $[p_0, p_1, \dots, p_{n-1}]$ , where each  $p_i$  is a composite proposition. Similarly, the alphabet in the regular expression is that of composite propositions. We seek to compute all (substitution, node) pairs  $(\phi, n)$  that satisfy the following condition. For every path to  $n$  in the flow graph (say  $[p_0, p_1, \dots, p_{m-1}]$ ) there exists a word  $[q_0, q_1, \dots, q_{m-1}]$  in the language of the pattern  $\phi(P)$  such that  $p_i \Rightarrow q_i$  for each  $0 \leq i < m$ .

To illustrate, the program in Figure 2 is annotated with composite propositions about the use and definition of different program variables. Consider the pattern  $P_{cse}$  that we introduced above. One solution is  $(\phi, n_5)$ , where  $\phi = \{x \rightarrow t, w \rightarrow p, y \rightarrow 5, z \rightarrow q\}$ . For example, one of the paths from the entry to  $n_5$  is through  $n_1, n_2$  and  $n_4$ . This path is labelled by the propositions

$$[ \begin{array}{l} p := 5 * q \wedge def(p) \wedge use(q), \\ r > 4 \wedge use(r), \\ t := 3 \wedge def(t), \\ t := 5 * q \wedge def(t) \wedge use(q) \end{array} ]$$

Each element of this path implies an associated element in a path in  $\phi(P_{cse})$ , namely:

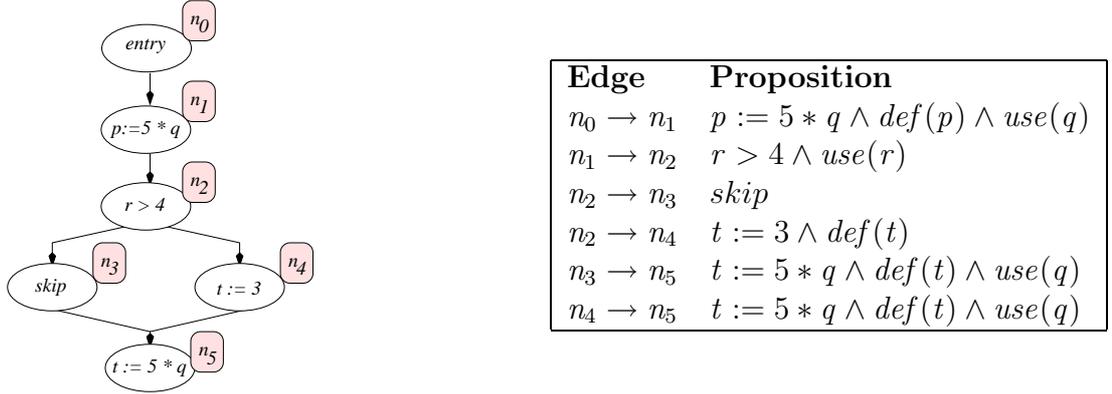


Figure 2: A program annotated with composite propositions

$$\begin{array}{l}
 [ \quad p := 5 * q, \\
 \quad \neg \text{def}(p) \wedge \neg \text{def}(5) \wedge \neg \text{def}(q), \\
 \quad \neg \text{def}(p) \wedge \neg \text{def}(5) \wedge \neg \text{def}(q), \\
 \quad t := 5 * q \quad ]
 \end{array}$$

In this paper, we shall initially ignore the propositional structure of the alphabet, and solve a (seemingly) simpler problem first, as a stepping stone towards the above application. We aim to develop an algorithm for solving *universal regular path queries* of the following form. Given a regular expression  $P$  that contains a number of variables, an edge-labelled directed graph  $G$  and a distinguished node  $n_0$  of  $G$ , it is required to compute all (substitution, node) pairs  $(\phi, n)$  so that all paths  $n_0 \rightarrow n$  are in the regular language  $\phi(P)$ . Naturally we are only interested in those pairs where  $n$  is actually reachable, so that there exists at least one path  $n_0 \rightarrow n$  in  $\phi(P)$ .

The structure of this paper is as follows. First we derive an algorithm for the case that  $P$  does not contain variables. Our purpose in presenting this derivation is to promote the use of universal algebra in reasoning about problems in automata theory; using a number of well-understood concepts from universal algebra, the derivation is a calculation of merely six steps. Next, we encode that algorithm as a Prolog program.

This paper is an exploratory step towards a tool for programming optimising transformations in a declarative style, and we conclude with a discussion of the further work required. We also briefly discuss some intriguing connections with other fields, in particular that of query languages for semi-structured data.

## 2 Specification and derivation

We shall weave our specification of the problem in hand and the derivation with a review of some standard notions from the calculus of relations and universal algebra. Readers familiar with any of [1, 7–9, 11–15, 26–29, 32, 36, 57] can safely skim subsections 2.1 and 2.2, and pass quickly on to Section 2.3, which summarises the calculation leaving out all the introductory material.

## 2.1 Specification

**Relations** We write  $R : X \leftarrow Y$  to indicate that  $R$  is a subset of  $X \times Y$ . This slightly unusual notation (with arrows pointing backwards from source  $Y$  to target  $X$ ) makes it a little easier to read formulae involving composition, defined below. The predicate  $xRy$  is shorthand for  $(x, y) \in R$ . Two relations  $R : X \leftarrow Y$  and  $S : Y \leftarrow Z$  can be composed to form  $R \cdot S$ :

$$x(R \cdot S)z \equiv \exists y \in Y : xRy \wedge ySz .$$

A relation  $R : X \leftarrow Y$  is said to be a *function* if it relates each  $y \in Y$  to exactly one  $x \in X$ . A particular example of a function is the identity relation  $id_X : X \leftarrow X$ , which maps each element of  $X$  to itself.

**Automata and folds** To formulate the specification of our problem as a relation, we shall first need to cast the familiar notion of an automaton in relational calculus. Functional programmers know that automata are very similar to the *fold-left* function  $([init, step])$ , which takes a constant *init*, a transition function *step*. When applied to a list, it sums the elements from left to right using *step*, starting with the constant *init* :

$$\begin{aligned} & ([init, step])[a_0, a_1, \dots, a_{n-1}] \\ & = \\ & ((init \text{ 'step' } a_0) \text{ 'step' } a_1) \dots \text{ 'step' } a_{n-1} . \end{aligned}$$

Indeed, fold-left exactly operates like a deterministic state machine, with initial state *init* and transition relation *step*. In a pioneering paper on algebra and automata theory, Eilenberg and Wright [30] have shown that fold-left can be generalised to take relational arguments. This has the obvious intuitive interpretation, where each application of *step* makes a nondeterministic choice among the possible transitions. All the familiar identities of functional programming generalise to the relational setting. We shall see several examples of such identities shortly.

For convenience, we shall think of *init* and *step* as relations with types:

$$\begin{aligned} init & : S \leftarrow \mathbf{1} , \quad \text{and} \\ step & : S \leftarrow S \times A . \end{aligned}$$

Here  $\mathbf{1}$  denotes a set that has only one element, which we denote as  $\bullet$ . If *init* is a function, it picks out exactly one element in  $S$ . In general, *init* corresponds to a subset of  $S$ , the set of all initial states of a nondeterministic state machine. That is,  $s(init)\bullet$  if  $s$  is an initial state. In what follows, we shall refer to a pair  $(init, step)$  (that has the above signature for some sets  $S$  and  $A$ ) as a *machine*.

In our problem, both the flow graph and the pattern can be modelled as machines. To wit, the flow graph is a machine

$$G = (G_0 : N \leftarrow \mathbf{1}, G_1 : N \leftarrow N \times A) .$$

Here  $G_0$  is the distinguished start node of the flow graph, and  $G_1$  specifies the edges. Thus  $([G]) : N \leftarrow A^*$ . The pattern is a machine

$$P = (P_0 : S \leftarrow \mathbf{1}, S_1 : S \leftarrow S \times A) ,$$

corresponding to the regular expression, and thus we have  $([P]): S \leftarrow A^*$ . In addition to this machine for the pattern, we also need a specification of its final states. This we chose to model as a relation

$$F : 1 \leftarrow S .$$

Note that (as in the case of initial states) we can identify such a relation  $F$  with a subset of  $S$ . That is,  $\bullet(F)s$  if  $s$  is a final state. The advantage of defining the final states in this way is the following concise definition of the language of the pattern:

$$F \cdot ([P]) : 1 \leftarrow A^* .$$

In words, a string  $x$  (a list with elements drawn from  $A$ ) is in the language of the pattern if  $([P])$  relates some final state to  $x$ . Below we shall sometimes write  $L(F, P)$  for the subset of  $A^*$  defined in this way.

To complete the specification of our problem, we need an operator that encodes universal quantification in the relational calculus. Given two relations  $R : X \leftarrow Z$  and  $S : Y \leftarrow Z$  that share the same source type, the *division* of  $R/S : X \leftarrow Y$  is defined by

$$x(R/S)y \equiv \forall z : ySz \Rightarrow xRz .$$

That is,  $R/S$  is the largest relation  $T$  such that  $T \cdot S \subseteq R$ . Expressed as an equivalence, that means

$$T \subseteq R/S \equiv T \cdot S \subseteq R \quad , \text{ for all } T : X \leftarrow Y .$$

Readers familiar with relational semantics of imperative programs will recognise the weakest prespecification [34, 35] in this formula.

**Specification** Here is the problem that we wish to solve: compute each node  $n$  of the flow graph such that

$$\forall xs \in A^* : n([G])xs \Rightarrow xs \in L(F, P) .$$

We could also have formulated that requirement thus:

$$\forall xs \in A^* : n([G])xs \Rightarrow \bullet(F \cdot ([P]))xs .$$

Hardened veterans of the relational calculus will spot that this formula can be expressed more concisely with division, thus obtaining

$$\bullet(F \cdot ([P])) / ([G]) n .$$

Now we have arrived at the official specification from which we wish to derive an algorithm:

$$(F \cdot ([P])) / ([G]) : 1 \leftarrow N .$$

The conciseness of this expression may appear somewhat forbidding. As we shall see, however, it allows us to give a very compact presentation of the algorithm that solves our problem.

## 2.2 Derivation

**From infinite to finite universal quantification** It is worth noting that our starting point is non-executable. To see why, consider the universal quantification

$$\forall xs \in A^* : n(\llbracket G \rrbracket)xs \Rightarrow xs \in L(F, P) .$$

Here we quantify over an infinite range, namely all strings with elements drawn from  $A$ . It stands to reason, therefore, that our first step towards an algorithm is to try and reduce that infinite quantification to a finite one. In terms of the official specification

$$(F \cdot \llbracket P \rrbracket) / \llbracket G \rrbracket ,$$

we aim to achieve that by shunting  $\llbracket P \rrbracket$  from the numerator to the denominator.

This is the purpose of the so-called *shunting* law:

$$(R \cdot f) / S = R / (S \cdot f^\circ) . \tag{1}$$

Here  $f$  is required to be a function, and  $f^\circ$  stands for the converse of  $f$  (the relation  $f$  with all pairs flipped round). In the left hand side the quantification is over the source type of  $f$ , whereas on the right hand side, the quantification is over the source type of  $R$ . Unfortunately the shunting law is not applicable here, because the relation  $\llbracket P \rrbracket$  that we wish to shunt is not necessarily a function.

There is hope, however, because every relation of  $R : X \leftarrow Y$  can be represented as a function  $\Lambda R : \mathbf{P}X \leftarrow Y$  that maps  $Y$  to the powerset of  $X(\mathbf{P}X)$ :

$$\Lambda R y = \{ x \mid xRy \} .$$

We call  $\Lambda R$  the *power transpose* of  $R$ . The function  $(\Lambda)$  is a bijection, and the original relation can be retrieved by composing with the membership relation  $mem : X \leftarrow \mathbf{P}X$ :

$$mem \cdot \Lambda R = R . \tag{2}$$

Let us now return to our original goal, namely to reduce the universal quantification in the specification from finite to infinite. We calculate:

$$\begin{aligned} & (F \cdot \llbracket P \rrbracket) / \llbracket G \rrbracket \\ = & \{ \text{Equation (2)} \} \\ & (F \cdot mem \cdot \Lambda \llbracket P \rrbracket) / \llbracket G \rrbracket \\ = & \{ \text{Equation (1)} \} \\ & (F \cdot mem) / (\llbracket G \rrbracket \cdot (\Lambda \llbracket P \rrbracket)^\circ) \end{aligned}$$

This has achieved the desired reduction, because instead of universally quantifying over all strings, we are now quantifying over all sets of states. As the set of states is finite, so is the collection of all its subsets. It may be helpful to spell out the details of this reduction from an infinite to a finite quantification. Consider the types in the division  $(F \cdot mem) / (\llbracket G \rrbracket \cdot (\Lambda \llbracket P \rrbracket)^\circ)$ . The right hand side operand has type  $1 \leftarrow \mathbf{P}S$  and the left hand operator has type  $N \leftarrow \mathbf{P}S$ . The division thus quantifies over the elements of  $\mathbf{P}S$ , which is a finite set. By contrast, in the original specification we quantified over the source type of  $\llbracket P \rrbracket$ , which is the infinite set of words  $A^*$ .

**Eliminating converse** Unfortunately it appears that we have created a new obstacle to executability in the denominator, however. The composition  $\langle [G] \rangle \cdot (\Lambda([P]))^\circ$  now involves an existential quantification over all strings since composition  $(\cdot)$  quantifies over the target type of  $\Lambda([P])^\circ$  (the source of  $\langle [G] \rangle$ ) which is  $A^*$ . To get rid of this new infinity, we first aim to massage the converse operator away. For that, we shall need some auxiliary facts about splits, and the range of a relation. Some readers will recognise that at this point we are heading for the construction of a product automaton — the precise sense in which that is true will become apparent shortly.

Given two relations  $R : X \leftarrow Y$  and  $S : Z \leftarrow Y$ , we can form a new relation  $\langle X, Y \rangle : (X \times Z) \leftarrow Y$  such that

$$(x, z)\langle R, S \rangle y = xRy \wedge zSy .$$

This is called the *split* of  $R$  and  $S$ . As an example,  $\langle [P], \Lambda([G]) \rangle$  is a relation of type

$$(N \times PS) \leftarrow A^* .$$

The *range* of a relation  $T : U \leftarrow V$  is a subset of the identity relation on  $U$ , defined by

$$u(\text{ran}(T))u' = u = u' \wedge \exists v : uTv .$$

Writing  $\text{outl} : X \leftarrow X \times Z$  and  $\text{outr} : Z \leftarrow X \times Z$  for the obvious projection functions, we have

$$R \cdot S^\circ = \text{outl} \cdot \text{ran}\langle R, S \rangle \cdot \text{outr}^\circ . \quad (3)$$

Both sides of this equation are merely ways of formulating the familiar predicate

$$x(R \cdot S^\circ)z = \exists y : xRy \wedge zSy = x(\text{outl} \cdot \text{ran}\langle R, S \rangle \cdot \text{outr}^\circ)z .$$

Applying Equation (3) to our programming problem, we obtain

$$\begin{aligned} & (F \cdot \text{mem}) / (\langle [G] \rangle \cdot (\Lambda([P]))^\circ) \\ &= (F \cdot \text{mem}) / (\text{outl} \cdot \text{ran}\langle [G], \Lambda([P]) \rangle \cdot \text{outr}^\circ) \end{aligned}$$

At first it might appear that little has been gained here. After all, the range operator itself is not executable. However, we have now set the scene for applying some well-known identities from functional programming to the fold-left operators. In what follows, we shall exclusively concentrate on obtaining an executable expression for the relation

$$\text{ran}\langle [G], \Lambda([P]) \rangle ,$$

which we can regard as a subset of  $(N \times PS)$ .

**Simplifying the fold-lefts, and range** First, consider  $\Lambda([P])$ . This is the function corresponding to a non-deterministic machine. It is well-known that this function can itself be expressed in terms of a deterministic machine. That is, there exists a function  $P'$  such that

$$\Lambda([P]) = \langle [P'] \rangle . \quad (4)$$

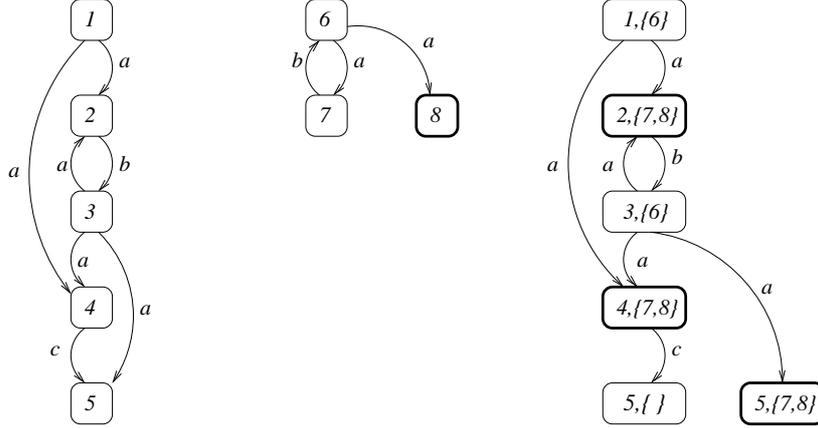


Figure 3: A graph, pattern, and their cross product.

In fact, the general proof of this equation was one of the main achievements of the paper by Eilenberg and Wright [30] that we mentioned earlier. We refrain from spelling out the detailed definition of  $P'$ .

Using the above equation in our programming problem, we obtain a subexpression of the form  $\langle ([G]), ([P']) \rangle$ . As every functional programmer knows, this is an inefficient program, because it makes two independent traversals of its input list. That is, given  $xs \in A^*$  we traverse the  $xs$  list twice; once to compute  $([G])xs$  and once to compute  $([P'])xs$ . Using the *tupling* transformation, the same result can be achieved in a single pass over  $xs$ . Formally, there exists a composite machine  $G \otimes P'$  such that

$$\langle ([G]), ([P']) \rangle = ([G \otimes P']) . \tag{5}$$

This identity is affectionately known as the *banana-split* law [44]. It was not invented to reason about automata, but rather to capture the tupling transformation [23, 54] in a calculational style. Again we leave it to the interested reader to work out the detailed definition of  $G \otimes P'$ . In formal language theory, it is known as the *product machine* of  $G$  and  $P'$  [37, page 59]. An example of a graph, a pattern and their cross product is shown in Figure 3. The final states of the pattern and cross product have thickly drawn edges.

We have achieved our task if we can give an executable expression for

$$ran([G \otimes P']) .$$

Before continuing however, we need a closure operator on relations. The *closure* of a relation  $R : X \leftarrow X$  is the smallest reflexive and transitive relations that contains  $R$ . We write  $R^*$  for the closure of  $R$ :

$$R^* = id \cup R \cup (R \cdot R) \cup (R \cdot R \cdot R) \cup \dots$$

In describing  $ran([G \otimes P'])$  in terms of machines, we are asking for the set of reachable states of the machine  $G \otimes P'$ . Given that reading, it should not come as a surprise that

$$ran([M]) = ran((M_1 \cdot outl^\circ)^* \cdot M_0) . \tag{6}$$

In words, to find the reachable states of a machine  $M$ , proceed as follows. Start with the initial states  $M_0$ . Then find all states reachable via zero or more transitions in the step relation  $M_1$ . Naturally this reachability problem can be implemented through depth-first search.

Summarising the results of this section, we derived that

$$\text{ran}\langle([G]), \Lambda([P])\rangle = \text{ran}(((G \otimes P')_1 \cdot \text{outl}^\circ)^* \cdot (G \otimes P')_0) .$$

This completes our derivation.

### 2.3 Summary and complexity analysis

The above exposition was leisurely, aimed at readers who are unfamiliar with relational calculus. We now repeat the same calculation as an expert would have written it in his notebook, and we analyse the result. First, the problem is reduced to computing the range of a relation:

$$\begin{aligned} & (F \cdot ([P])) / ([G]) \\ = & \quad \{\text{cancelling } \Lambda \text{ (reverse)}\} \\ & (F \cdot \text{mem} \cdot \Lambda([P])) / ([G]) \\ = & \quad \{\text{shunting}\} \\ & (F \cdot \text{mem}) / (([G]) \cdot (\Lambda([P]))^\circ) \\ = & \quad \{\text{split and range}\} \\ & (F \cdot \text{mem}) / (\text{outl} \cdot \text{ran}\langle([G]), \Lambda([P])\rangle \cdot \text{outr}^\circ) \end{aligned}$$

Next, we elaborate the range expression:

$$\begin{aligned} & \text{ran}\langle([G]), \Lambda([P])\rangle \\ = & \quad \{\text{Eilenberg-Wright}\} \\ & \text{ran}\langle([G]), ([P'])\rangle \\ = & \quad \{\text{banana split}\} \\ & \text{ran}([G \otimes P']) \\ = & \quad \{\text{range of fold-left}\} \\ & \text{ran}(((G \otimes P')_1 \cdot \text{outl}^\circ)^* \cdot (G \otimes P')_0) \end{aligned}$$

In words, we have derived an algorithm that proceeds in four steps:

1. Let  $P'$  be the deterministic equivalent of  $P$ .
2. Take product machine  $G \otimes P'$ .
3. Compute the reachable states of  $G \otimes P'$ .
4. Return the set

$$\{ n \mid \forall s : (n, s) \text{ reachable} : s \text{ final in } P' \} .$$

What is the time complexity of this algorithm? There are a number of characteristics of our application that simplify the analysis. First, the pattern is very small compared to the flow graph, so we can regard its size as a constant. Furthermore, in a typical flow graph all nodes have a bounded out-degree, so the number of edges is linear in the number of nodes. It thus stands to reason that we measure the complexity in terms of the number of nodes in  $G$ . The crucial step is the third, where we compute the reachable states of the product machine. The size of that machine is linear in the size of  $G$ . Furthermore, the reachable states can be computed by depth-first search, again linear in  $G$ . We conclude that the overall complexity of the algorithm is linear. It is also worth noting that when using this algorithm in an optimising compiler, the patterns for recognising optimisation opportunities are fixed. Its only input is the program which is converted to the flowgraph  $G$ . Thus, the computational expense imposed by the patterns can be considered constant.

### 3 Implementing universal regular path queries

Let us now return to the original problem from the introduction, which may have seemed slightly more general than the algorithm that we have just derived.

First of all, our machines operate on predicates, not symbols that are drawn from a finite set. That is only a seeming generalisation, for we did not exploit the finiteness of the alphabet anywhere in our proofs.

The notion of acceptance of sequences of propositions is also a special case of our earlier definitions. A transition  $a \xrightarrow{p} b$  in the pattern is possible on input  $q$  precisely when  $q \Rightarrow p$ . This is a slightly more complicated way of mapping labelled edges to transitions, but there is nothing special about the resulting transition relation.

All that remains, therefore, is to cater for the presence of free variables, and finding substitutions for those variables. This we can do by regarding the pattern with variables as mere shorthand for a family of ground patterns. By writing the algorithm we have derived above in a logic programming language, with the pattern as a predicate that takes the variables as explicit arguments, we get a program that exhibits exactly the desired behaviour. However, as we shall see, the program employs logical negation, which is logically sound only if all variables have been instantiated to ground terms.

Astute readers will have noticed a subtle discrepancy between our informal discussion of the problem in the introduction, and its formalisation in relational calculus. In the introduction, we stipulated that there must exist a path  $v \rightarrow w$  in the language of the pattern, as well as requiring that all paths  $v \rightarrow w$  are in that language. The difference is only important, of course, if  $w$  is not reachable from  $v$ . Here we shall take advantage of the insistence on the existence of a suitable path (which is dictated by our application to program transformation), and first run an existential path query to instantiate the variables appropriately. We can then run the ground universal query to check that the instantiation is indeed a valid answer.

Below we shall show how a particular query can be compiled to a Prolog program. Such compilation happens when the transformations and the associated path queries are known; the queries are then run when we also have a flow graph to transform. The particular implementation of Prolog that we have chosen is called XSB [55, 58]. It is particularly

suitable for experiments in program transformation and analysis [24].

### 3.1 Common Subexpression Elimination

Common subexpression elimination is applicable at node  $N$  if all paths from program entry to  $N$  can be split into four parts:

- First there is a part that we do not care about, consisting of zero or more edges.
- Next, we encounter an edge whose target is an assignment  $W := A$ , where  $A$  is a non-trivial expression and the set of variables used in  $A$  is  $Vs$ . Also, the variable  $W$  should not occur in  $Vs$ .
- Then we have zero or more edges to nodes that do not define  $W$ , nor any of the variables in  $Vs$ .
- Finally, we have an edge target at  $N$ , where the statement is of the form  $X := A$ .

If this condition is satisfied, and there exists at least one path of the appropriate form, the statement at node  $N$  can be replaced by the assignment  $X := W$ .

We could write the above condition as the following regular path query.

```
{}*;  
{tgt'(assign(W,A)),  
  not(triv(A)),  
  uses(A,Vs),  
  not(elem(V,Ws))} ;  
{not(def'(W)), not(somedefs'(Vs))}* ;  
{tgt'(assign(X,A))}
```

Here each of the goals in curly brackets matches a single edge. Each predicate that is marked with an apostrophe takes the edge as an implicit argument. Below we show how to map this query into Prolog, and that mapping makes the implicit edge arguments explicit. In addition the program will check that the variable  $W$  does not occur in the list of variables  $Vs$ . To illustrate the use of the query, an example program is shown in Figure 4, and the results of running the query in Prolog are shown in Figure 5. Note that it is a property of this query that it can succeed only by instantiating all the free variables to ground terms. We shall rely on that property in what follows.

The query will depend on its free variables  $(W, A, Vs, X)$ . We can represent these variables as free variables in XSB. We can also put them in a term wrapper (`subst`) to give us a representation of substitution  $\Phi$  which can then be passed between predicates

```
Phi = subst(W,A,Vs,X)
```

To define the flow graph itself in Prolog, we declare the nodes and edges with clauses such as these:

```

0:  entry
1:  w := a + g(b,c)
2:  i := g(a,b)
3:  if i < 10 then goto 4 else goto 6
4:  x := a + g(b,c)
5:  i := i + x; goto 3
6:  a := g(a,b)
7:  w := a + g(b,d)
8:  i := g(a,b)
9:  if i < 10 then goto 10 else goto 12
10: x := a + g(b,d)
11: i := i + x; goto 9
12: a := g(a,b)
13: exit

```

Figure 4: Example program

```
| ?- univpaths(subst(W,A,Vs,X),0,N).
```

```

W = w
A = plus(var(a),g(var(b),var(c)))
Vs = [a,b,c]
X = x
N = 4;

```

```

W = w
A = plus(var(a),g(var(b),var(d)))
Vs = [a,b,d]
X = x
N = 10;

```

```
no
```

Figure 5: Example run of query

```
node(0) .
node(1) .
edge(0,e0,1) .
```

The middle argument of the `edge` predicate is an identifier for the relevant edge, so `edge(N_0,E,M_0)` and `edge(N_1,E,M_1)` implies that  $N_0=N_1$  and  $M_0 = M_1$ .

The statements at each node are specified by the relevant clauses of the `stmt` predicate:

```
stmt(assign(i,g(var(a),var(b))),2) .
stmt(if(less(var(i),const(10))),3) .
```

We can then check for the statement at the target of an edge as follows:

```
tgt(S,E) :- edge(_N,E,M), stmt(S,M) .
```

Note how the edge  $E$  has been made explicit. Similar definitions can be made for all the constituents of the above regular path query, and a summary can be found in Figure 6. Using these definitions, we can define the predicate `goal`, with a separate case for each of the constituents of our query:

```
:- table goal/3 .
```

```
goal(0,subst(_W,_A,_Vs,_X),_E) .
```

```
goal(1,subst(W,A,Vs,_X),E) :- tgt(assign(W,A),E),
                             not(triv(A)),
                             uses(A,Vs) .
```

```
goal(2,subst(W,_A,Vs,_X),E) :- not(def(W,E)), not(somedefs(Vs,E)) .
```

```
goal(3,subst(_W,A,_Vs,X),E) :- tgt(assign(X,A),E) .
```

The tabling directive for `goal` is important for efficiency, as the same goal may be evaluated many times at a particular edge.

With the definition of `goal` in hand, we can construct the nondeterministic automaton that corresponds to the pattern. We name the states of the pattern  $\{a,b,c\}$ , with  $a$  the initial state, and  $c$  the final state. One can now define the possible transitions thus:

```
npattern(Phi,a,a,E) :- goal(0,Phi,E) .
npattern(Phi,a,b,E) :- goal(1,Phi,E) .
npattern(Phi,b,b,E) :- goal(2,Phi,E) .
npattern(Phi,b,c,E) :- goal(3,Phi,E) .
```

```

elem(A, [A|_As]) .
elem(A, [_B|As]) :- elem(A, As) .

src(S, E) :- edge(N, E, _M), stmt(S, N) .
tgt(S, E) :- edge(_N, E, M), stmt(S, M) .

def(V, E) :- tgt(assign(V, _X), E) .
somedefs(Vs, E) :- elem(V, Vs), def(V, E) .

triv(const(_C)) .
triv(var(_V)) .

append([], X, X) .
append([A|X], Y, [A|Z]) :- append(X, Y, Z) .

uses(const(_C), []) .
uses(var(V), [V]) .
uses(plus(P, Q), Z) :- uses(P, X), uses(Q, Y), append(X, Y, Z) .
uses(g(P, Q), Z) :- uses(P, X), uses(Q, Y), append(X, Y, Z) .

```

Figure 6: Prolog preliminaries

### 3.2 Solving existential path queries

To solve an existential path query, we first construct the product of the flow graph and the non-deterministic pattern:

```

nproduct(Phi, N1, A1, N2, A2) :- edge(N1, E, N2),
                                npattern(Phi, A1, A2, E) .

```

This definition says that there exists a transition from  $(N1, A1)$  to  $(N2, A2)$  in the product if there exists corresponding transitions from  $N1$  to  $N2$  and from  $A1$  to  $A2$ .

We now need to compute what states are reachable in the product. Using *tabling*, we can write this simply as the definition of reflexive transitive closure in Prolog:

```

:- table nreach/5 .
nreach(_Phi, N1, P1, N1, P1) .
nreach(Phi, N1, P1, N2, P2) :- nproduct(Phi, N1, P1, Na, Pa),
                                nreach(Phi, Na, Pa, N2, P2) .

```

There exists a path from  $N1$  to  $N2$  in the language of the pattern if there exists a path from  $(N1, a)$  to  $(N2, c)$  in the product. Accordingly, we define:

```

somepaths(Phi, N1, N2) :- nreach(Phi, N1, a, N2, c) .

```

Interested readers may find it an amusing exercise to formally derive this program in relational calculus: the derivation for existential queries is much simpler than that for universal ones.

### 3.3 Solving universal regular path queries

Our next task is to write a program for solving universal regular path queries. For that, we need a deterministic version of the pattern automaton. The construction of that automaton is a little tricky, so we defer its discussion till later. For now it suffices to know that it is given by three predicates, namely `state`, `final` and `pattern`. The first of these corresponds to `node` and it is true of all the states in the pattern automaton. The second predicate singles out those states that are final. The last predicate

```
pattern(Phi,P1,P2,E)
```

checks whether for a particular substitution `Phi`, the transition from state `P1` to `P2` is implied by the edge `E`.

Recall that in our algorithm, we need to compute the product automaton of the flow graph and the pattern. The definition is analogous to that in the nondeterministic case:

```
product(Phi,N1,P1,N2,P2) :- edge(N1,E,N2),
                             pattern(Phi,P1,P2,E) .
```

That is, we can make the transition from the product state `(N1,P1)` to `(N2,P2)` if there are relevant edges in the flow graph, and in the pattern.

The next step in the algorithm is to compute the reachable states in the product automaton. Again we use tabling:

```
:- table reach/5 .
```

```
reach(_Phi,N1,P1,N1,P1) .
reach(Phi,N1,P1,N2,P2) :- product(Phi,N1,P1,Na,Pa),
                           reach(Phi,Na,Pa,N2,P2) .
```

Finally, we need to ensure that all paths from `N1` to `N2` are in the language of our pattern, that is

```
allpaths(Phi,N1,N2) :- bagof(P,
                             reach(Phi,N1,p1,N2,P),
                             Ps),
                    all(final)(Ps) .
```

The `bagof` primitive collects all `P` that can be reached (with the given instantiations of the variables) in `Ps`. It then only remains to check that all states in `Ps` are final. The higher-order predicate `all` is defined by

```
:- hilog all .
```

```
all(_P)([]).
all(P)([X|Xs]) :- P(X), all(P)(Xs) .
```

In the introduction of this section, we already indicated that the determinization of the pattern requires negation, and that this imposes the requirement that all variables in the substitution `Phi` are ground in the predicate `allpaths(Phi,N1,N2)`. To ensure that this requirement is indeed satisfied, we define

```

univpaths(Phi,N1,N2) :- somepaths(Phi,N1,N2),
                        ground(Phi),
                        allpaths(Phi,N1,N2) .

```

The program given here is close to that we derived in Section 2.2. Although we have not worked out the formal details, we believe the transition from relational algebra to a Prolog program could be mechanised. Obviously the difficult point in such a mechanisation would be the treatment of negation in association with non-ground queries.

### 3.4 Deterministic pattern automaton

It now remains to define the transition relation of the deterministic pattern automaton, which we have called `pattern(Phi,P1,P2,E)`. We shall split its definition into two parts:

```

pattern(Phi,P1,P2,E) :- goals(P1,I,Phi,E), pat(I,P1,P2) .

```

The role of the two parts on the right hand side is as follows:

- The second part is a predicate `pat(I,P1,P2)`. Here `I` is a 4-bit number that we shall interpret as a bit vector: the least significant bit indicates whether `goal(0,...)` was proved at edge `E`, the second bit indicates whether `goal(1,...)` was proved, and so on. Based on the bitvector `I`, `pat(I,P1,P2)` gives the transition relation between the states `p0..p7`.
- The first part of `pattern` is `goals(P1,I,Phi,E)`. It sets the bit vector `I` according to the provability of the four goals.

As an example, let us consider the state `p2`, which corresponds to the singleton set `{b}`. We can make transitions according to `goal2` or `goal3` from this state:

```

goals(p2,I,Phi,E) :- try(2,I2,Phi,E),
                    try(3,I3,Phi,E),
                    I is 4*I2 + 8*I3 .

```

The predicate `try(K,B,Phi,E)` attempts to prove goal number `K`, setting bit `B` to 0 or 1 accordingly:

```

try(K,1,Phi,E) :- goal(K,Phi,E) .
try(K,0,Phi,E) :- tnot(goal(K,Phi,E)) .

```

Here `tnot` is the special version of negation necessary for the correct handling of tabled predicates. Like its ordinary counterpart in logic programming, in general we require that it is applied only to ground arguments. This is the reason that we needed to resort to existential queries to do the variable binding before calling the algorithm we derived earlier.

It is interesting to note that if we restrict our attention to variables whose values are drawn from a finite domain, one could employ a constraint logic programming language to implement the algorithm for universal queries. The logical negation operator `tnot` could then be replaced by negation for constraints. We are currently investigating such an implementation, and weighing its advantages against the restrictions it places on the expressiveness of path queries.

## 4 Discussion

This paper is part of a larger effort to construct a toolkit for easy experimentation with compiler optimisations. In previous work, we have specified the side conditions of such transformations in a variant of temporal logic [42], inspired by the work of Steffen and his coworkers on specifying data flow analyses through temporal logic [59]. The formulae in temporal logic are verified using a model checker, which also finds instantiations of free variables. We noticed that many of our examples do not use the full power of temporal logic, and this motivated the exploration of universal regular path queries. In hindsight it is a very obvious thing to do, given the close connection between path problems, regular algebra and program analysis [6, 60, 61]. It remains to be seen how the algorithm presented here compares in practice with our use of a model checker.

Two of us (Van Wyk and Lacey) have been working on a method of proving the correctness of program transformations whose side conditions are stated in temporal logic, in collaboration with Neil Jones [38]. It was their draft paper that prompted the exploration of a simplified formalism for the side conditions. It appears that the proof methods for the temporal formalism carry over without much modification to regular expressions.

There are multiple ways in which the present algorithm could be improved. In particular, it will pay to start out with a minimal deterministic automaton for the pattern [2]. It is still possible for the composite automaton to become nondeterministic during the processing of substitutions, but at least the nondeterminism will be restricted to those places where it is actually necessary.

We did not set out to apply our previous work on relational algebra to this problem, but the proof turned out to be so pretty that we decided to include it in this paper. It would be interesting to see a further exploration of automata and language theory in this style (see also [5, 13, 53]). The application of program transformation to algorithms involving automata was pioneered by Bob Paige [16, 22, 40].

Very recently we became aware that *existential* regular path queries are a well studied subject in the database community. In particular, the query language UnQL provides the possibility of querying hierarchically structured data through the use of regular expressions that contain variables [18]. The queries are existential in that it suffices to find *some* path that is in the regular language, rather than requiring that *all* paths (between the relevant vertices) are in the regular language. It would be interesting to investigate whether the techniques used to speed up existential regular path queries [45] can be adapted for universal queries as well. This connection with database research also follows a lead of Bob Paige: his own language for expressing side conditions of transformations in APTS [49] was inspired by DataLog.

## Acknowledgements

Three anonymous referees suggested many improvements to the presentation of this paper. Oege de Moor would like to thank Annie Liu for many constructive criticisms on the work presented here; she especially helped to improve the presentation of the derivation. She also made us aware of the existence of XSB as an appropriate tool for this type of experiment.

Shin-Cheng Mu and Richard Bird provided valuable comments on an early draft. We would like to thank Microsoft Research for its generous support of this research programme, as part of the Intentional Programming project.

## References

- [1] C. J. Aarts, R. C. Backhouse, P. F. Hoogendijk, E. Voermans, and J. C. S. P. Van der Woude. A relational theory of datatypes. Available from URL <http://www.win.tue.nl/win/cs/wp/papers/papers.html>, September 1992.
- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1985.
- [3] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- [4] U. Assmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In P. Fritzson, editor, *Compiler Construction 1996*, volume 1060 of *Lecture Notes in Computer Science*. Springer, 1996.
- [5] R. Backhouse. Fusion on languages. In *10th European Symposium on Programming, ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2001.
- [6] R. C. Backhouse and B. A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and its Applications*, 15:161–186, 1975.
- [7] R. C. Backhouse, P. De Bruin, P. F. Hoogendijk, G. Malcolm, T. S. Voermans, and J. C. S. P. Van der Woude. Polynomial relators. In M. Nivat, C. S. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology, Workshops in Computing*, pages 303–362. Springer-Verlag, 1992.
- [8] R. C. Backhouse, P. De Bruin, G. Malcolm, T. S. Voermans, and J. C. S. P. Van der Woude. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. Elsevier Science Publishers, 1991.
- [9] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer-Verlag, 1993.
- [10] A. J. C. Bik, P. J. Brinkhaus, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Transformation mechanisms in mt1. Technical report, Leiden Institute of Advanced Computer Science, 1998.
- [11] R. S. Bird and O. De Moor. From dynamic programming to greedy algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 43–61. Springer-Verlag, 1993.

- [12] R. S. Bird and O. De Moor. Solving optimisation problems with catamorphisms. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, volume 669 of *Lecture Notes in Computer Science*, pages 45–66. Springer-Verlag, 1993.
- [13] R. S. Bird and O. De Moor. Relational program derivation and context-free language recognition. In A. W. Roscoe, editor, *A Classical Mind: Essays dedicated to C.A.R. Hoare*, pages 17–35. Prentice Hall International, 1994.
- [14] R. S. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1996.
- [15] R. S. Bird, P. F. Hoogendijk, and O. De Moor. Generic programming with relations and functors. *Journal of Functional Programming*, 6(1):to appear, 1996.
- [16] B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995.
- [17] J. M. Boyle, K. W. Dritz, M. M. Muralidharan, and R. Taylor. Deriving sequential and parallel programs from pure LISP specifications by program transformation. In L. G. L. T. Meertens, editor, *Proceedings of the IFIP TC2/ WG 2.1 working conference on Program Specification and Transformation*, pages 1–19. North-Holland, 1987.
- [18] P. Buneman, M. Fernandez, and D. Suciu. Unql: A query language and algebra for semistructured data based on structural recursion. *VLDB Journal*, 9(1):76–110, 2000.
- [19] J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type analysis and data structure selection. In B. Möller, editor, *Constructing Programs from Specifications*, pages 126–164. North-Holland, 1991.
- [20] J. Cai and R. Paige. Towards increased productivity of algorithm implementation. *ACM Software Engineering Notes*, 18(5):71–78, 1993.
- [21] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, 1992.
- [22] C. Chang and R. Paige. From regular expressions to DFAs using compressed NFAs. *Theoretical Computer Science*, 178(1-2):1–36, 1997.
- [23] W. N. Chin. Fusion and tupling transformations: Synergies and conflicts (invited paper). In *Fuji International Workshp on Functional and Logic Programming*, pages 176–195. World Scientific, 1995.
- [24] M. Codish, B. Demoen, and K. Sagonas. Xsb as the natural habitat for general purpose program analysis. Technical report, KU Leuven, 1996.
- [25] J. R. Cordy, I. H. Carmichael, and R. Halliday. The TXL programming language, version 8. Legasys Corporation, April 1995.

- [26] O. De Moor. Categories, relations and dynamic programming. D.Phil. thesis. Technical Monograph PRG-98, Computing Laboratory, Oxford, UK, 1992.
- [27] O. De Moor. Categories, relations and dynamic programming. *Mathematical Structures in Computing Science*, 4:33–69, 1994.
- [28] H. Doornbos. *Reductivity Arguments and Program Construction*. Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1996.
- [29] H. Doornbos and R. C. Backhouse. Induction and recursion on datatypes. In B. Möller, editor, *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 242–256. Springer-Verlag, 1995.
- [30] S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11(4):452–470, 1967.
- [31] R. E. Faith, L. S. Nyland, and J. F. Prins. KHEPERA: A system for rapid implementation of domain-specific languages. In *Proceedings USENIX Conference on Domain-Specific Languages*, pages 243–255, 1997.
- [32] P. J. Freyd and A. Šcedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.
- [33] D. Hanson, C. W. Fraser, and T. A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, 1992.
- [34] C. A. R. Hoare and J. He. The weakest prespecification, I. *Fundamenta Informaticae*, 9(1):51–84, 1986.
- [35] C. A. R. Hoare and J. He. The weakest prespecification, II. *Fundamenta Informaticae*, 9(2):217–251, 1986.
- [36] P. F. Hoogendijk and R. C. Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Science of Computer Programming*, 22(1-2):67–105, 1994.
- [37] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [38] N. Jones, D. Lacey, and E. Van Wyk. Proving correctness of compiler transformations. In preparation.
- [39] M. Jourdan, D. Parigot, Julié, O. Durin, and C. Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Conference on Programming Languages Design and Implementation*, pages 209–222, 1990. Published as *ACM SIGPLAN Notices*, 25(6).
- [40] J. P. Keller and R. Paige. Program derivation with verified transformations – a case study. *Communications on Pure and Applied Mathematics*, 48(9-10), 1996.

- [41] M. Klein, J. Knoop, D. Koschützski, and B. Steffen. DFA & OPT-METAFrame: a toolkit for program analysis and optimization. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 418–421. Springer, 1996.
- [42] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In *Proceedings Programming Language Implementation and Logic Programming (PLILP), and Algebraic and Logic Programming (ALP)*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 1994.
- [43] P. Lipps, U. Mönke, and R. Wilhelm. OPTRAN – a language/system for the specification of program transformations: system overview and experiences. In *Proceedings 2nd Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65, 1988.
- [44] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.
- [45] T. Milo and D. Suciu. Index structures for path expressions. In *International Conference on Database Theory '99*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.
- [46] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [47] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, 1986.
- [48] R. Paige. Real-time simulation of a set machine on a RAM. In N. Janicki and W. Koczkodaj, editors, *Computing and Information*, volume 2, pages 69–73. Canadian Scholars' Press, 1989.
- [49] R. Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Joint 6th International Conference on Programming Language Implementation and Logic Programming (PLILP) and 4th International conference on Algebraic and Logic Programming (ALP)*, volume 844 of *Lecture Notes in Computer Science*, pages 5–24. Springer, 1991.
- [50] R. Paige. Future directions in program transformations. *Computing Surveys*, 28A(4), 1996.
- [51] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):401–454, 1982.
- [52] R. Paige, R. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical computer science*, 40(1):67–84, 1985.

- [53] H. A. Partsch. Transformational program development in a particular problem domain. *Science of Computer Programming*, 7(2):99–241, 1986.
- [54] A. Pettorossi. Methodologies for transformations and memoing in applicative languages. Ph.D. thesis CST-29-84, University of Edinburgh, Scotland, 1984.
- [55] I. V. Ramakrishnan, Prasad Rao, Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, pages 697–711, 1995.
- [56] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.
- [57] F. J. Rietman. *A relational calculus for the design of distributed algorithms*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, 1995.
- [58] K Saganos. *The XSB System v2.4: Programmers Manual*, July 2001.
- [59] B. Steffen. Data flow analysis as model checking. In *Proceedings of Theoretical Aspects of Computer Science*, pages 346–364, 1991.
- [60] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the Association for Computing Machinery*, 28(3):594–614, 1981.
- [61] S. W. K. Tjiang and J. L. Hennessy. Sharlit — a tool for building optimizers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.
- [62] E. Visser, Z. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming '98*, ACM SigPlan, pages 13–26. ACM Press, 1998.
- [63] D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *ACM Transactions on Programming Languages and Systems*, 19(6):1053–1084, 1997.