Deriving a systolic regular language recognizer

Matteo Vaccari
Department of Computing Science
Università degli Studi di Milano
via Comelico 39
20135 Milano, Italy
vaccari@dsi.unimi.it

Roland Backhouse
Department of Computing Science
Eindhoven University of Technology
PO Box 513
5600 MB Eindhoven, The Netherlands
rolandb@win.tue.nl

December 19, 1996

Abstract

We present a derivation of a regular language recognizing circuit originally developed by Foster and Kung [4]. We make use of point-free relation algebra, in a style combining elements from earlier work in the Eindhoven Mathematics of Program Construction group [0], and from Ruby [6]. First we derive non-systolic recognizers, much in the same way as functional programs are derived. Then we make use of standard circuit transformation techniques, recast in the relation algebra framework, to obtain circuits that are very close to the ones presented by Foster and Kung.

0 Introduction

In 1982, Foster and Kung [4] presented a specialised silicon compiler that constructs recognizers for regular languages. The compiler was presented without formal justification; indeed, they did not present a formal specification of the functionality of the compiler. Their informal description of the functioning left much room for alternative interpretations.

Subsequently, Backhouse [1] verified the correctness of Foster and Kung's compiler. His task amounted primarily to reverse engineering — trying to discover the specification satisfied by the compiler. This resulted in the discovery of an error in Foster and Kung's construction — acknowledged by Foster in his Ph.D. thesis [3]. Otherwise the formal calculations in Backhouse's report were disappointingly complicated and not judged by its author to be worthy of widespread publication.

In this paper we present a formal *derivation* of Foster and Kung's compiler. The complexities of the earlier *verification* have been overcome in two ways: by exploiting (point-free) relation algebra rather than elementary predicate calculus, and by a judicious decomposition of the design task. Our design consists of first deriving a non-systolic implementation, followed by a transformation of this design to a systolic version using standard techniques ("slowing" and "retiming" [6]).

A precise definition of "systolic" can be found in Leiserson's thesis [8]. For our purposes, a circuit is systolic when it can be seen as a network of processing elements interconnected by wires, these wires being interrupted by delays (registers). The presence of the delays on the wires has an important effect on the minimum clock period that can be assigned to a circuit. In fact, the presence of long wires uninterrupted by delays forces the designer to assign a larger clock period to the circuit, and that in turn may have a negative effect on the overall performance.

Foster presents a formal verification of the compiler in his Ph.D. thesis [3]. Both the specification and the implementation have been adapted in order to overcome the error in [4]. We, ourselves, have as yet been unable to understand Foster's arguments, possibly because we do not understand how to describe the non-standard components he uses as stream transducers. In this paper we take the easy way out and avoid the problem rather than overcome it. A further weakness of this paper is that the individual calculations are not tied together into a completely rigorous whole. These weaknesses are pointed out in the relevant places.

A formal derivation of a similar compiler has also been given by Kaldewaij and Zwaan [7], but their implementation is not systolic, in the sense that the minimum clock period that can be assigned to their circuits is a function of the length of the regular expression to be matched. In contrast, the circuits that we generate can be assigned a clock period that is independent of the number of sequence operators in the regular expression, although it does depend on the number of star and choice operators.

The paper is organized as follows: in the next section, we introduce the reader to relation algebra, and our definition of "circuit". In section 2 we show how to specify the problem with relation algebra. Then in section 3 we derive a first, non-systolic version of the recognizers. In section 4 standard circuit transformations are applied to obtain a systolic version. Finally, in section 5 some conclusions are drawn.

1 About circuits and relation algebra

We will write our specifications and our circuits in point-free relation algebra. A brief introduction to our style of relation algebra follows; for a more complete treatment, see [0].

A (binary) relation over a set $\mathcal{U}$ is a set of pairs of elements of $\mathcal{U}$. For x,y in $\mathcal{U}$ and R a relation over $\mathcal{U}$, we write $x\langle R\rangle y$ instead of (x,y) in R. When a relation R satisfies $x\langle R\rangle y \wedge z\langle R\rangle y \Rightarrow x{=}z$ we say that the relation is *deterministic*. In that case it may be considered as a function with domain on the right side and target on the left side; we denote by $R.y$ the unique x such that $x\langle R\rangle y$ holds, if such an x exists. The reason for this name is that we usually interpret relations as programs taking input from the right and producing output on the left. In this way a deterministic relation is interpreted as a deterministic program. We usually use the letters f, g, h to stand for deterministic relations. We use the convention that "." associates to the right so that $f.g.x$ should be parsed as $f.(g.x)$. (This is contrary to the convention used in the lambda calculus.)

Relations are ordered by the usual set inclusion ordering. Hence the set of relations forms a complete lattice. The relation corresponding to the empty set is denoted by $\perp\!\!\!\perp$, and the relation that contains all pairs of elements of $\mathcal{U}$ is denoted by $\top\!\!\!\top$. The *identity relation*, I, is defined by $x\langle I\rangle y \equiv x{=}y$. The composition of two relations R,S is denoted by $R{\circ}S$ and defined by $x\langle R{\circ}S\rangle y \equiv \exists(z :: \ x\langle R\rangle z \wedge z\langle S\rangle y)$. Composition is associative and has unit element I. The converse of a relation R is written $R^{\cup}$ and is defined by $x\langle R^{\cup}\rangle y \equiv y\langle R\rangle x$.

A *monotype* is a relation A such that $A \subseteq I$. An example of a monotype is $\mathbb{N}$, defined by $n\langle\mathbb{N}\rangle m \equiv n{=}m \wedge (n$ is a natural number$)$. There is a clear one-to-one correspondence between the subsets of $\mathcal{U}$ and the monotypes; and this makes it possible to embed set calculus in relation calculus. The *left domain* of relation R, denoted $R<$, is the least monotype A such that $A{\circ}R = R$. As its name suggests, $R<$ represents the set of all x such that x is related by R to some y.

A *left condition* is a relation R such that $R = R{\circ}\top\!\!\!\top$. Clearly, if R is a left condition, then, for all x, $\exists(y :: \ x\langle R\rangle y) \equiv \forall(z :: \ x\langle R\rangle z)$. This suggests that a left condition may also be interpreted as a set, as we may take it to represent the set of values x such that $\exists(y :: \ x\langle R\rangle y)$. We usually abuse notation by writing $x{\in}R$ in place of $\exists(y :: \ x\langle R\rangle y)$, when R is a left condition. A *right condition* is defined analogously, but we will not need to use right conditions in this paper.

There is obviously a 1–1 correspondence between monotypes and left conditions given by the functions $R \mapsto R<$ and $R \mapsto R{\circ}\top\!\!\!\top$. Making the right choice of which to use can simplify calculations a great deal. We use both in this paper.

The relation $R{\vartriangle}S$ (pronounced *R split S*) is defined as the least relation X such that for all x, y and z, $(x,y)\langle X\rangle z \equiv x\langle R\rangle z \wedge y\langle S\rangle z$. Note that the requirement that $R{\vartriangle}S$ be the *least* relation satisfying the above equation in X implies that there is no y such that $x\langle R{\vartriangle}S\rangle y$ when x is not a pair. That

$\cup$ < > * $^{\sigma}$ $^{\varpi}$	all unary operators
.	function application
$\times + \vartriangle$	product, sum, split
$\circ$	relational composition
$\cup \cap$	union, intersection
$= \subseteq$	equality, inclusion
$\wedge \vee$	conjunction, disjunction
$\Rightarrow \Leftarrow$	implication, consequence
$\equiv$	boolean equivalence

Table 0: Precedence of operators, from highest to lowest

is, the left domain of $R \vartriangle S$ is a set of pairs.

Common mathematical practice is not to make explicit that what is being defined is the least solution of a certain equation. "We define the relation $R \vartriangle S$ by $(x, y)\langle R \vartriangle S\rangle z \equiv x\langle R\rangle z \wedge y\langle S\rangle z$" is the more usual way to express the above definition. For brevity we adopt this practice from now on.

Split enjoys the property

$$(0) \qquad R \vartriangle S \circ f = (R \circ f) \vartriangle (S \circ f)$$

if f is a deterministic relation.

We define $R \times S$ (pronounced R *times* S) by $(x, y)\langle R \times S\rangle(z, v) \equiv x\langle R\rangle z \wedge y\langle S\rangle v$. The *projection* relations $\ll$ and $\gg$ are defined by $x\langle\ll\rangle(y, z) \equiv x{=}y$ and $x\langle\gg\rangle(y, z) \equiv x{=}z$. The following properties are easily proved:

$$(1) \qquad \begin{array}{rcl} R \times S \circ T \times U & = & (R \circ T) \times (S \circ U) \\ R \times S \circ T \vartriangle U & = & (R \circ T) \vartriangle (S \circ U) \end{array}$$

The large number of binary operators that we use may make it difficult to parse our expressions; but the precedences were carefully chosen in order to minimise the need for parentheses, and the spacing around operators hints at the way to read a formula. See table 0 for a complete list of precedences.

Following established practice (see [5, 6, 11]) we model a circuit as a relation between arbitrary collections of *streams*, a stream being a total function with domain the integer numbers. Abusing language somewhat, we will use the word "circuit" to mean an actual circuit, or a relation between streams as described above. Context should make clear which one is meant. We usually denote streams by the letters a through e.

As an alternative to our definition, it is possible to define streams as functions on the natural numbers (rather than the integers); but this leads to a more complicated theory, where many equalities no longer hold (see [6] for details). Our definition corresponds in a sense to ignoring initialisation problems. One may see here an analogy with traditional derivation of programs, where one can factor a proof of correctness into a proof of partial correctness together with a proof of termination. What we have instead is a derivation of a circuit that is correct, provided that the circuit can be initialized. This leaves us with the obligation of proving that our circuits can be correctly initialized. We will not devote much space to this latter problem. We trust that the reader will see that our circuits can be initialized, provided that there is a way to set the contents of all boolean delays to *false*, and all character delays to some value different from the encoding of all symbols in $\mathcal{T}$. We assume that some "reset" wire exists in the implementation that performs this function, and we will not give further mention to this issue.

Given a relation R, a relation between streams can be constructed by "lifting": $a\langle\dot{R}\rangle b \equiv \forall(n :: \ a.n\,\langle R\rangle\, b.n)$. Hence for any R, relation $\dot{R}$ is a circuit. Note that, for deterministic relation f, stream a and integer m, $f.a.m = (\dot{f}.a).m$. We refer to this property in our calculations by the hint "lifting". Circuits can be built by relational composition, and product: given R and S, two circuits, the relations $R{\circ}S$ and $R{\times}S$ are also circuits.

A particular relation on streams is the *primitive delay*, denoted by ∂ and defined by $a\langle\partial\rangle b \equiv \forall(n :: \ a.(n+1) = b.n)$. The *delay* relation, written $\lhd$, is a generalisation of primitive delay to arbitrary pairings of streams. It is defined as the least fixed point of function $X \mapsto \partial \cup X{\times}X$:

$$\lhd = \mu(X \mapsto \partial \cup X{\times}X)$$

Delay can be thought of informally as the union of an infinite list of terms $\lhd\ = \partial \cup \partial{\times}\partial \cup \partial{\times}(\partial{\times}\partial) \cup (\partial{\times}\partial){\times}\partial \cup (\partial{\times}\partial){\times}(\partial{\times}\partial) \cup \ldots$. The *antidelay* $\rhd$ is defined to be the converse of delay. In the interpretation as circuits, a delay is a memory element that outputs the contents of memory on the left side, and at every clock tick replaces the contents of memory with the input on its right side. The interpretation of antidelays is the same, with the role of "left" and "right" reversed. Note that both $\lhd$ and $\rhd$ are deterministic.

We define the identity relation for streams in a way that is similar to how we defined delay. The *primitive stream identity* is defined by $a\langle\bar{\iota}\rangle b \equiv \forall(n :: \ a.n = b.n)$. The identity on arbitrary pairings of streams, denoted by ι, is then defined by

$$\iota = \mu(X \mapsto \bar{\iota} \cup X{\times}X)$$

The delay relations are polytypic in the sense that they apply the primitive delay ∂ to a collection of wires, independently of the shape of the collection. Formally,

$$(2) \qquad \lhd^{<} = \lhd^{>} = \iota = \rhd^{<} = \rhd^{>}$$

(Note that this and other properties of delays are proved in appendix A.) A similar domain property that we use frequently is:

(3) $\quad \lhd \circ \iota \times \iota = \lhd \times \lhd \quad$ and $\quad \iota \times \iota \circ \lhd = \lhd \times \lhd$

These equations express the fact that applying $\lhd$ to a pair of (collections of) wires ($\lhd \circ \iota \times \iota$) is the same as $\lhd$ applied to each component of the pair ($\lhd \times \lhd$). From this property, and the corresponding one for $\rhd$, one immediately obtains the following useful distributivity properties: for $\diamond \in \{\lhd, \rhd\}$,

$$(4) \qquad \begin{array}{rcl} \diamond \circ R \times S & = & (\diamond \circ R) \times (\diamond \circ S) \\ R \times S \circ \diamond & = & (R \circ \diamond) \times (S \circ \diamond) \\ \diamond \circ R \vartriangle S & = & (\diamond \circ R) \vartriangle (\diamond \circ S) \end{array}$$

and from (0) and the fact that delays are deterministic, one obtains

(5) $\quad R \vartriangle S \circ \diamond = (R \circ \diamond) \vartriangle (S \circ \diamond)$

Finally, the *feedback* of a circuit R, written R^σ, is defined by $a \langle R^\sigma \rangle b \equiv a \langle R \rangle (b, a)$.

We may now summarize our means of constructing circuits:

0. If R is a relation then $\dot{R}$ is a circuit.
1. The projections $\ll$ and $\gg$ are circuits.
2. If R,S are circuits, then $R \circ S$, $R \times S$, $R \vartriangle S$, and $R \cup$ are circuits.
3. Delays and antidelays are circuits.
4. If R is a circuit, then R^σ is a circuit.

A circuit R is said to be *combinational* if it is defined exclusively by means of the first three items in the above list; i.e., if delay, antidelay and feedback do not appear in its definition.

A circuit term has an interpretation as a picture that is often useful as an aid to understanding how a circuit term is interpreted as a real circuit. A picture shows which "parts" of a circuit are connected; and interconnections are important in order to evaluate the circuit's performance. Figure 0 shows the correspondence between pictures and circuit terms.

The picture interpretation shows the presence of *combinational paths* in the circuit. A picture may be seen as a graph, where combinational elements and delays are nodes, and wires are edges. A combinational path is a path in the picture that does not contain delays. One important parameter for the efficiency of a circuit implementation is the clock speed. Roughly speaking, the shortest clock period that can be assigned to a circuit implementation

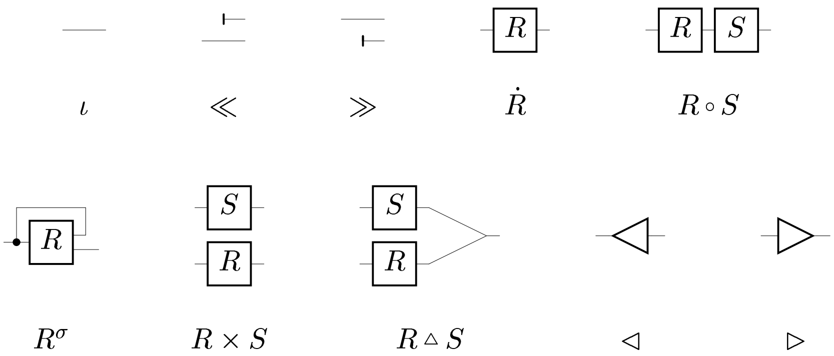


Figure 0: circuits and their pictures

grows with the length of the longest combinational path in the circuit. For this reason it is preferable to design circuits with short combinational paths. A circuit is said to be *systolic* when it is built out of small modules, interconnected by wires interrupted by delays (see [8]).

There are many optimisation techniques that can be used to improve the performance of circuits. Here we will make use of *retiming* and *slowdown*.

Retiming (see [9]) is a transformation that is essentially based on the following laws: given that R is a circuit as defined above,

(6) $\quad \lhd \circ R = R \circ \lhd$

and

(7) $\quad \rhd \circ R = R \circ \rhd$

These laws can be proved by structural induction (see appendix A). Combining (6) with the property that

(8) $\quad \rhd \circ \lhd = \iota = \lhd \circ \rhd$

we obtain the property

(9) $\quad R = \rhd \circ R \circ \lhd \quad ,$

and from the combination (8) and (7), we obtain

(10) $\quad R = \lhd \circ R \circ \rhd \quad ,$

for all circuits R.

(Note that the two retiming laws (9) and (10) break down when the domain of streams is taken to be the natural numbers rather than the integers. Instead of equalities one obtains isomorphisms up to retiming, making calculations more cumbersome.)

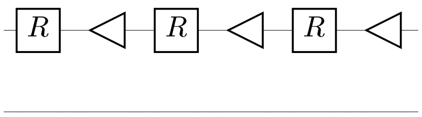
Another optimisation technique is slowdown [12]. Given a circuit R, the circuit $slow.R$ is obtained by replacing every occurrence of $\rhd$ and $\lhd$ in R by $\rhd\circ\rhd$ and $\lhd\circ\lhd$, respectively. The slowed circuit is not equivalent to the original one; it has different timing properties. The reason for implementing a slowed version of a circuit is that the extra delays that are introduced can be shifted around by means of the retiming laws, with the general goal of making the circuit more systolic.

The relation between R and $slow.R$ is formally described in [12, p. 8]. Our use of $slow$ is one of the places we alluded to in the introduction where our account is not completely rigorous.

To illustrate the use of slowing, suppose we are implementing circuit

$$(11) \qquad (\iota\times(\lhd\circ R))^n$$

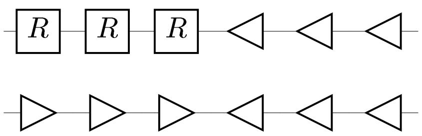
for some $n>0$, where R is combinational. The picture interpretation (for $n = 3$) shows a long combinational path:



By retiming, one obtains:

$$
\begin{array}{ll}
 & (11) \\
= & \quad \{ \quad \text{retiming} \quad \} \\
 & (\rhd\circ\iota\times(\lhd\circ R)\circ\lhd)^n \\
= & \quad \{ \quad \text{delays, (4) and (8)} \quad \} \\
 & (\rhd\times R\circ\lhd)^n \\
= & \quad \{ \quad \text{retiming (6), applied } n-1 \text{ times} \quad \} \\
 & (\rhd\times R)^n\circ\lhd^n
\end{array}
$$

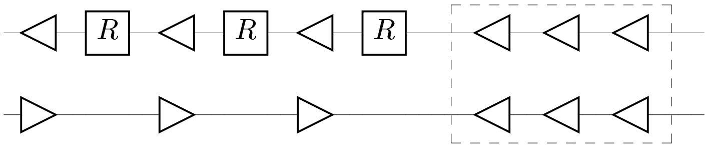
This transformation does not buy us anything, since the resulting circuit still has a long combinational delay (since R is combinational):



If we choose to implement a slowed version of (11) instead, then we have:

$$
\begin{array}{ll}
 & slow.(\iota\times(\lhd\circ R))^n \\
= & \{ \quad \text{definition} \quad \} \\
 & (\iota\times(\lhd\circ\lhd\circ R))^n \\
= & \{ \quad \text{the above derivation, taking } R := \lhd\circ R \quad \} \\
 & (\rhd\times(\lhd\circ R))^n\circ\lhd^n
\end{array}
$$

The last line is a circuit whose interpretation has no long combinational paths:



In fact, the length of the longest combinational path is no longer dependent on n. The $\lhd^n$ term, enclosed in a dashed box in the picture, would not normally be implemented. Rather, it can be thought of as a timing specification of the circuit. It tells us what precisely is the timing difference between the original circuit, and the one that is actually implemented.

Suppose one were to implement the circuit in the last picture, minus the part in the dashed box. In order to obtain a circuit equivalent to $slow.(\iota\times(\lhd\circ R))^n$, one should delay the input on the upper wire by three clock ticks, and "anticipate" the output of the lower wire by the same number of ticks. (Of course, while delaying is certainly an implementable operation, anticipating usually is not.) Note that the placement of delays and antidelays (outside the dashed box) implies that the flow of data through the circuit is both from left to right and from right to left; this is called "contra-flow".

2 The specification

The problem we want to consider is that of formulating a syntax-directed construction of a systolic circuit that (repeatedly) recognizes strings in the language denoted by a regular expression. The syntax of a regular expression is given by the BNF grammar

$$\mathcal{E} \ ::= \ t \,|\, \mathcal{E}{+}\mathcal{E} \,|\, \mathcal{E}\,;\mathcal{E} \,|\, \mathcal{E}^*$$

where t stands for all elements of a given finite alphabet $\mathcal{T}$.

To begin with, we define a mapping from $\mathcal{E}$ to the set of stream transducers $\mathcal{F}$, where

$$\mathcal{F} = Stream(\mathbb{B}) \leftarrow Stream(\mathcal{T}) \times Stream(\mathbb{B})$$

Thus, given a regular expression E, the recognizer for E maps a pair consisting of a stream of characters (elements of $\mathcal{T}$) and a stream of booleans into another stream of booleans. The boolean input is a so-called "enable" signal. A value of *true* for the enable input indicates the start of a new input sequence of characters. For instance, if the expression to be recognized is $t\,;t$, and the set of symbols is $\mathcal{T}=\{t,u\}$, we expect the following behaviour:

output	character (a)	enable bit (e)
0	t	0
0	t	1
0	t	0
1	t	0
0	t	1
0	u	0
0	t	0

As we shall see, a value of *true* for the enable input does not terminate any foregoing sequence of characters. The following is another example of required behaviour:

output	character (a)	enable bit (e)
0	t	0
0	t	1
0	t	1
1	t	0
1	t	0
0	t	0

Usually we use a to range over a stream of input characters and e (for *e*nable bit) to range over a stream of input booleans.

In order to avoid the error in [4] we shall restrict the regular expressions to those expressions not including a subexpression E^* such that the empty word is a member of E. It is well known that this does not reduce the expressive power of regular expressions and that every regular expression can be easily transformed to one of this form.

We denote the isomorphism between strings of booleans and left conditions by tt (standing for "times true") and define it by, for all integers m and n and all streams of booleans e,

$$m\langle tt.e\rangle n \equiv e.m$$

For instance, if the stream e is defined for all n by $e.n \equiv (n \text{ is odd})$, then $tt.e = \{1,3,5,\ldots\}$. We also introduce a binary relation on integers $mem.(E,a)$, for each expression E and each stream of letters a, defined by

$$m\langle mem.(E,a)\rangle n \equiv a[n,m) \in E$$

where $a[n,m)$ denotes the string $a.n \,;\, a.(n{+}1) \,;\, \ldots \,;\, a.(n{+}m{-}1)$. Note the switch in the order of m and n. Returning to the example where $E = t\,;t$, we have that if the stream a is defined by $a.n \equiv t$ for all n, then $m\langle mem.(t\,;t,a)\rangle n \equiv m = n + 2$. Another way to look at mem is as a set transformer. If we compose $mem.(E,a)$ after a left condition, we obtain another left condition:

$$mem.(E,a) \circ (R \circ \top\!\!\top) = (mem.(E,a) \circ R) \circ \top\!\!\top$$

So if $R \circ \top\!\!\top$ can be interpreted as the set $\{0,1,5\}$, then, given a defined as above, $mem.(t\,;t,a) \circ R \circ \top\!\!\top$ could be interpreted as $\{2,3,7\}$.

The following properties of mem are easily verified (see appendix B):

$$(12) \qquad \begin{array}{rcl} m\langle mem.(t,a)\rangle n & \equiv & m = n{+}1 \wedge a.n = t \\ mem.(E{+}F,a) & = & mem.(E,a) \cup mem.(F,a) \\ mem.(E\,;F,a) & = & mem.(F,a) \circ mem.(E,a) \\ mem.(E^*,a) & = & (mem.(E,a))^* \end{array}$$

These properties provide ample justification for choosing to use relation algebra in the formal specification of the recognizer: the function $E \mapsto mem.(E,a)$ is a homomorphism from the algebra of expressions to the algebra of relations.

We say that a circuit f (formally a stream transducer, i.e., an element of $\mathcal{F}$) *recognizes* regular expression E when the following holds, for all $a \in Stream(\mathcal{T})$ and $e \in Stream(\mathbb{B})$:

$$tt.f.(a,e) = mem.(E,a) \circ tt.e$$

A way to read this is: the set of times at which e is $true$, that is $tt.e$, is transformed by mem into a set that must be exactly the same as the set of times at which $f.(a,e)$ is true.

3 A non-systolic recognizer

Once the specification is made clear, deriving a (non-systolic) recognizer is easy. We begin by deriving the recognizer for a single character. We have:

$$
\begin{array}{ll}
 & m \in mem.(t,a) \circ tt.e \\
\equiv & \quad \{ \quad \text{composition} \quad \} \\
 & \exists(n :: \ m\langle mem.(t,a)\rangle n \wedge n \in tt.e) \\
\equiv & \quad \{ \quad mem \quad \} \\
 & \exists(n :: \ n = m-1 \wedge a.n = t \wedge n \in tt.e) \\
\equiv & \quad \{ \quad \text{one-point rule} \quad \} \\
 & a.(m-1) = t \wedge (m-1) \in tt.e \\
\equiv & \quad \{ \quad \text{lifting, } tt \quad \} \\
 & ((\dot{=}\, t).a).(m-1) \wedge e.(m-1) \\
\equiv & \quad \{ \quad \text{lifting} \quad \} \\
 & (e \mathbin{\dot{\wedge}} (\dot{=}\, t).a).(m-1) \\
\equiv & \quad \{ \quad \text{delay} \quad \} \\
 & (\lhd.(e \mathbin{\dot{\wedge}} (\dot{=}\, t).a)).m \\
\equiv & \quad \{ \quad tt \quad \} \\
 & m \in tt.(\lhd.(e \mathbin{\dot{\wedge}} (\dot{=}\, t).a))
\end{array}
$$

From this we obtain:

$$
\begin{array}{ll}
 & f \text{ recognizes letter } t \\
\equiv & \quad \{ \quad \text{definition} \quad \} \\
 & \forall(a,e :: \ tt.f.(a,e) = mem.(t,a) \circ tt.e) \\
\equiv & \quad \{ \quad \text{sets} \quad \} \\
 & \forall(m,a,e :: \ m \in tt.f.(a,e) \equiv m \in mem.(t,a) \circ tt.e) \\
\equiv & \quad \{ \quad \text{above} \quad \} \\
 & \forall(m,a,e :: \ m \in tt.f.(a,e) \equiv m \in tt.(\lhd.(e \mathbin{\dot{\wedge}} (\dot{=}\, t).a))) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & f = \lhd \circ \dot{\wedge} \circ (\dot{=}\, t) \times \iota
\end{array}
$$

Next we consider that the expression has the form $E+F$ for some expressions E and F. Suppose that f, g recognize E, F respectively. Then,

$$
\begin{array}{ll}
 & h \text{ recognizes } E+F \\
\equiv & \quad \{ \quad \text{definition} \quad \} \\
 & \forall(a,e :: \ tt.h.(a,e) = mem.(E+F,a) \circ tt.e)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \quad mem\text{, distributivity} \quad \} \\
& \forall(a,e :: \; tt.h.(a,e) \; = \; mem.(E,a) \circ tt.e \cup mem.(F,a) \circ tt.e) \\
\equiv & \{ \quad \text{hypothesis} \quad \} \\
& \forall(a,e :: \; tt.h.(a,e) = tt.f.(a,e) \cup tt.g.(a,e)) \\
\equiv & \{ \quad tt \text{ and } \cup\text{; } tt \text{ is an isomorphism} \quad \} \\
& \forall(a,e :: \; h.(a,e) = f.(a,e) \mathbin{\dot\vee} g.(a,e)) \\
\equiv & \{ \quad \text{calculus} \quad \} \\
& h = \dot\vee \circ f \triangle g
\end{array}
$$

The next case is an expression of the form $E\,;F$ for some expressions E and F. Suppose again that f, g recognize E, F respectively. Then,

$$
\begin{array}{ll}
& h \text{ recognizes } E\,;F \\
\equiv & \{ \quad \text{definition} \quad \} \\
& \forall(a,e :: \; tt.h.(a,e) = mem.(E\,;F,a) \circ tt.e) \\
\equiv & \{ \quad mem \quad \} \\
& \forall(a,e :: \; tt.h.(a,e) = mem.(F,a) \circ mem.(E,a) \circ tt.e) \\
\equiv & \{ \quad \text{hypothesis} \quad \} \\
& \forall(a,e :: \; tt.h.(a,e) = mem.(F,a) \circ tt.f.(a,e)) \\
\equiv & \{ \quad \text{hypothesis} \quad \} \\
& \forall(a,e :: \; tt.h.(a,e) = tt.g.(a,f.(a,e))) \\
\equiv & \{ \quad tt \text{ is an isomorphism; calculus} \quad \} \\
& h = g \circ \ll \triangle f
\end{array}
$$

The final, and most interesting case, is when the given expression has the form E^* for some E. A circuit containing feedback is clearly needed. This is the case where Foster and Kung's original design contained an error.

The problem occurs because the defining equation of R^*, for any given relation R, does not necessarily have a unique solution. It does have a unique solution if R is so-called "well-founded" [2]. A relation R is said to be *well-founded* whenever, for all relations X, it holds that $X = R \circ X \Rightarrow X = \perp\!\!\perp$. Anticipating the forthcoming calculation somewhat, we determine a condition for the relation $mem.(E,a)$ to be well-founded. For all E and a, we have:

$$X = mem.(E,a) \circ X$$

$$
\begin{array}{ll}
\Rightarrow & \{ \quad \text{Leibniz} \quad \} \\
& X \circ \top\!\top = mem.(E,a) \circ X \circ \top\!\top \\
\equiv & \{ \quad \text{pointwise interpretation; define } S = X \circ \top\!\top, \\
& \quad\ \ \text{a left condition which we regard as a set} \quad \} \\
& \forall(m :: \ m \in S \equiv \exists(n :: \ m\langle mem.(E,a)\rangle n \wedge n \in S)) \\
\equiv & \{ \quad \text{definition of } mem \quad \} \\
& \forall(m :: \ m \in S \equiv \exists(n :: \ a[n,m] \in E \wedge n \in S)) \\
\Rightarrow & \{ \quad \bullet \quad \text{assume } \varepsilon \notin E \text{ and } mem.(E,a)\!> \subseteq \mathbb{N} \\
& \quad\ \ \text{Predicate calculus.} \quad \} \\
& \forall(m : \ m \in \mathbb{N} : \ m \in S \equiv \exists(n : \ n \in \mathbb{N} : \ n < m \wedge n \in S)) \wedge S \subseteq \mathbb{N} \\
\equiv & \{ \quad \text{the natural numbers are well-founded} \quad \} \\
& S = \emptyset \\
\equiv & \{ \quad \text{calculus, } S = X \circ \top\!\top \quad \} \\
& X = \perp\!\perp
\end{array}
$$

We have thus found that the assumptions $\varepsilon \notin E$ and $mem.(E,a)\!> \subseteq \mathbb{N}$ together imply that $mem.(E,a)$ is well-founded. The second of these assumptions is equivalent to postulating that the stream a is such that if a segment of a is a word in E, then this segment is wholly contained in the non-negative "half". Actually, it simplifies matters if we make an even stronger postulate, namely that for all $n < 0$, the value of $a.n$ is some character not appearing in E. This corresponds to asserting that the circuit is fed invalid input until time 0. One may think of time 0 as the moment after the circuit is reset. Given this assumption, we may henceforth just say that $mem.(E,a)$ is well-founded if $\varepsilon \notin E$.

We are now ready to tackle the derivation of the circuit that recognizes E^*. Assume f recognizes E. Assume also that $\varepsilon \notin E$. Then

$$
\begin{array}{ll}
& g \text{ recognizes } E^* \\
\equiv & \{ \quad \text{definition} \quad \} \\
& \forall(a,e,b :: \ b\langle g\rangle(a,e) \ \equiv \ tt.b = mem.(E^*,a) \circ tt.e)
\end{array}
$$

Now,

$$
\begin{array}{ll}
& tt.b = mem.(E^*,a) \circ tt.e \\
\equiv & \{ \quad mem \ (12) \quad \} \\
& tt.b = (mem.(E,a))^* \circ tt.e
\end{array}
$$

$\equiv$ { $\varepsilon \notin E$, so $mem.(E,a)$ is well-founded.
By the u.e.p. for regular expressions [2] }

$tt.b = tt.e \cup mem.(E,a) \circ tt.b$

$\equiv$ { f recognizes E }

$tt.b = tt.e \cup tt.f.(a,b)$

$\equiv$ { tt }

$tt.b = tt.(e \dot{\vee} f.(a,b))$

$\equiv$ { tt is an isomorphism }

$b = e \dot{\vee} f.(a,b)$

$\equiv$ { calculus; define $reorg.((x,y),z) = (y,(x,z))$ }

$b = (\dot{\vee} \circ \iota \times f \circ reorg).((a,e),b)$

Thus

g recognizes E^*

$\equiv$ { above }

$\forall(a,e,b :: \; b\langle g\rangle(a,e) \equiv b = (\dot{\vee} \circ \iota \times f \circ reorg).((a,e),b))$

$\equiv$ { feedback }

$g = (\dot{\vee} \circ \iota \times f \circ reorg)^\sigma$

Summarising the results so far, we have derived a syntax directed translation from $\mathcal{E}$ to $\mathcal{F}$, which we may call τ, defined as follows:

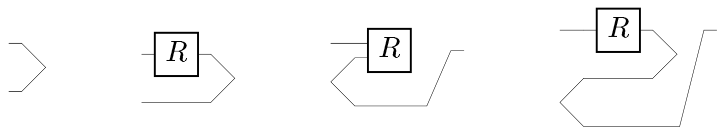
$$
(13) \qquad
\begin{array}{rcll}
\tau.t & = & \lhd \circ \dot{\wedge} \circ (\dot{=}t) \times \iota & \text{for all } t \in \mathcal{T} \\
\tau.(E+F) & = & \dot{\vee} \circ \tau.E \vartriangle \tau.F & \\
\tau.(E\,;F) & = & \tau.F \circ \ll \vartriangle \tau.E & \\
\tau.E^* & = & (\dot{\vee} \circ \iota \times \tau.E \circ reorg)^\sigma &
\end{array}
$$

and such that, for all regular expressions E, $\tau.E$ recognizes E.

4 Making the design systolic

The circuits we have derived so far are not systolic; for instance, if one were to build a recognizer for the string "$t\,;u\,;v$", with t, u and v all elements of $\mathcal{T}$, the resulting circuit would be

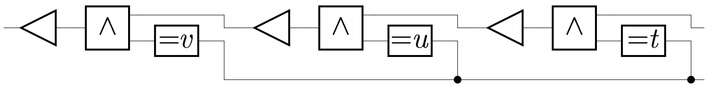
$$(\lhd \circ \dot{\wedge} \circ (\dot{=}v) \times \iota) \circ \ll \vartriangle (\lhd \circ \dot{\wedge} \circ (\dot{=}u) \times \iota) \circ \ll \vartriangle (\lhd \circ \dot{\wedge} \circ (\dot{=}t) \times \iota)$$



term $\quad \iota \times R \circ term \quad$ *str*.R $\quad$ *str*.$(\iota \times R \circ term)$

Figure 1: picture interpretations

it is apparent from the picture interpretation of the above circuit



that there is a path that is not interrupted by delays from one side to the other of the circuit. Such a path, often called a "combinational path", places a constraint on the implementation since the longer the path, the longer it takes for the real circuit to stabilize after a change in the input. This forces the implementer to use a clock with a longer period, with a possible negative effect on the performance of the whole circuit (see, for example, [9]). Even worse is the fact that for every string recognizer, this path grows in length with the length of the string.

In order to apply the optimisation techniques we have introduced in section 1, we should try to transform our circuits so that they exhibit contra-flow. Taking inspiration from Foster and Kung's work, we concentrate on sequence. We begin by considering a general technique for introducing contra-flow in a circuit. Suppose we want to implement a circuit R, taking input on the right side and producing output on the left side. A simple way to introduce contra-flow is to implement

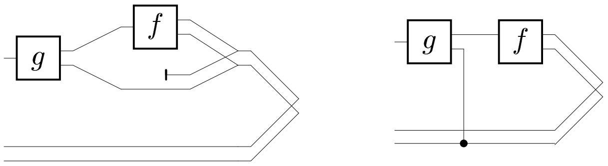
(14) $\quad \iota \triangle R \circ \top\!\!\top$

instead of R, the relation between the two being

$$(a,b)\langle \iota \triangle R \circ \top\!\!\top \rangle c \equiv b \langle R \rangle a$$

(Note the inversion of a and b, and that c does not appear on the right side). By exploiting the fusion laws (1) and defining $term = \iota \triangle \iota \circ \top\!\!\top$, we may rewrite (14) as

$$\iota \times R \circ term$$



$\iota \times g \circ \iota \times (\ll\!\vartriangle\! f) \circ term \quad \iota \times g \circ plumb \circ \iota \times f \circ term$

Figure 2: a simplification

We may define a transformation *str* (from "straighten") by

$$b\langle str.R\rangle a \equiv \exists(c :: \ (a,b)\langle R\rangle c)$$

so that

(15) $\quad str.(\iota\times R \circ term) = R$

(see figure 1). Note that it is possible to define *str* by composition of "wiring" relations:

$$str.R \;=\; \gg \circ\ term^{\cup} \times \iota \circ lsh \circ \iota\times R \circ \ll^{\cup}$$

where *lsh* is defined by $lsh.(a,(b,c)) = ((a,b),c)$.

An interesting property of circuits like (14) is

(16) $\quad \lhd \circ S \circ \top\!\top = S \circ \top\!\top \quad$ and $\quad \rhd \circ S \circ \top\!\top = S \circ \top\!\top$

This is a straightforward consequence of the retiming equations (6) and (7) and the domain equations (2).

Returning now to recognizers, let $g \circ \ll\!\vartriangle\! f$ be the recognizer for $E\,;F$. The picture interpretation for $\iota\times(g \circ \ll\!\vartriangle\! f) \circ term$ suggests a simplification (see figure 2): with *plumb* a wiring relation defined by

$$plumb.((x,y),z) = ((x,y),(x,z))$$

it holds that

(17) $\quad \iota\times(\ll\!\vartriangle\! f) \circ term \;=\; \iota\times(\ll\!\vartriangle\! f) \circ plumb^{<} \circ term$

and

(18) $\quad \iota\times g \circ \iota\times(\ll\!\vartriangle\! f) \circ plumb^{<} \;=\; (\iota\times g \circ plumb^{<}) \circ plumb \circ (\iota\times f \circ plumb^{<})$

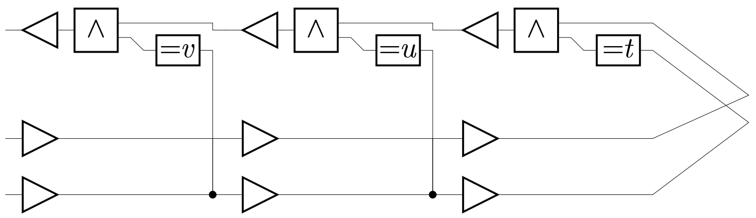


Figure 3: a systolic recognizer for the expression $t\,;u\,;v$

(The proof is omitted. In each case a simple pointwise calculation establishes the property.) Note that $\mathit{plumb}_{<}$ is the monotype that stands for the set of elements of the shape $((x,y),(x,z))$, for some x, y and z. These two properties, applied iteratively, with (17) as the base case and (18) as the body of the iteration, result in a recognizer of the expression $E_1\,;E_2\,;\ldots;E_n$ of the form

$$\iota{\times}f_1 \circ \mathit{plumb} \circ \iota{\times}f_2 \circ \mathit{plumb} \circ \ \ldots\ \circ \iota{\times}f_n \circ \mathit{term}$$

(The expression has to be parsed as $(((E_1\,;E_2)\,;E_3)\,;\ldots)\,;E_n$ to achieve this result.)

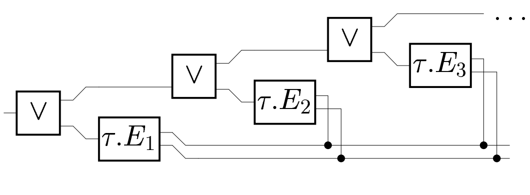
The benefit of this transformation is maximised in the case that the expression to be recognized is a sequence of characters. Before slow-down the constructed circuit has of course a pair of wires stretching across the full breadth of the circuit. After slowing and retiming the circuit is completely systolic (see figure 3). If the regular expression does not have this very special shape the benefit is diminished. Consider for instance expressions of the form

$$E_1 + E_2 + E_3 + \ldots$$

The translation of such expressions has the form

$$\dot{\vee} \circ \tau.E_1 \vartriangle (\dot{\vee} \circ \tau.E_2 \vartriangle (\dot{\vee} \circ \tau.E_3 \vartriangle \ \ldots))$$

and here we have combinational paths whose length depends on the number of occurrences of "+" in the expression:



We are unable for the moment to find a way to avoid this; our current design is well-optimized for expressions that contain many more ";" operators than "+". On the other hand, this problem is also present and is also left unsolved in Foster and Kung's work. (They simply fail to observe the problem.)

In order to exploit the systolic optimisation we have found, we define a new function ρ such that for all regular expressions E,

$$\rho.E = slow.(\iota \times \tau.E \circ term)$$

We aim to find an appropriate recursive definition of ρ; however, the need to append the $(\circ\, term)$ part is problematic, since we would then have to rely on the regular expression being parsed in an appropriate way. To overcome this problem, we define an auxiliary function υ, as follows:

$$\begin{array}{cl}
 & \rho.E \\
= & \{ \quad \text{above definition} \quad \} \\
 & slow.(\iota \times \tau.E \circ term) \\
= & \{ \quad \text{definition of } slow \quad \} \\
 & \iota \times slow.\tau.E \circ term \\
= & \{ \quad \text{equation (17)} \quad \} \\
 & \iota \times slow.\tau.E \circ plumb{<} \circ term \\
= & \{ \quad \bullet \quad \upsilon.E \quad \text{satisfies} \\
 & \qquad \upsilon.E \circ plumb{<} \;=\; \iota \times slow.\tau.E \circ plumb{<} \\
 & \qquad \text{equation (17)} \quad \} \\
 & \upsilon.E \circ term
\end{array}$$

This way we have reduced the problem of finding a recursive definition for ρ to the problem of finding a recursive definition for υ, where the $(\circ\, term)$ part does not occur.

We now proceed by cases. Note first, however, that the requirement on υ

$$\upsilon.E \quad \text{satisfies} \quad \upsilon.E \circ plumb{<} \;=\; \iota \times slow.\tau.E \circ plumb{<}$$

is met if

$$\upsilon.E = \iota \times slow.\tau.E$$

The addition of the context condition $plumb{<}$ is needed for the application of equation (18) which only occurs in the case of an expression of the form $E\,;F$. In all but this case we therefore ensure that the latter equation is satisfied rather than the former. The base case is the single character: for $t \in \mathcal{T}$,

$$
\begin{array}{ll}
 & \iota \times slow.\tau.t \\
= & \{ \quad \text{definitions of } slow \text{ and } \tau \quad \} \\
 & \iota \times (\lhd \circ \lhd \circ \dot{\wedge} \circ (\dot{=} t) \times \iota) \\
= & \{ \quad \text{retiming (9)} \quad \} \\
 & \rhd \times (\lhd \circ \dot{\wedge} \circ (\dot{=} t) \times \iota) \circ \lhd
\end{array}
$$

Appealing to (16) we define

$$\upsilon.t \;=\; \rhd \times (\lhd \circ \dot{\wedge} \circ (\dot{=} t) \times \iota)$$

(Note: This is the most serious transgression of rigour in the paper.)

So much for the character-recognizer. For the choice operator we have:

$$
\begin{array}{ll}
 & \iota \times slow.\tau.(E + F) \\
= & \{ \quad \text{definitions of } \tau \text{ and } slow \quad \} \\
 & \iota \times (\dot{\vee} \circ slow.\tau.E \vartriangle slow.\tau.F) \\
= & \{ \quad \text{property of } str\text{, eq. (15)} \quad \} \\
 & \iota \times (\dot{\vee} \circ str.\rho.E \vartriangle str.\rho.F)
\end{array}
$$

Thus we define

$$\upsilon.(E + F) = \iota \times (\dot{\vee} \circ str.\rho.E \vartriangle str.\rho.F)$$

The reintroduction of ρ in the last formula guarantees that the subexpressions are transformed to systolic circuits in an uniform way. The same trick is used for the star operator:

$$
\begin{array}{ll}
 & \iota \times slow.\tau.E^* \\
= & \{ \quad \text{definitions of } \upsilon,\, \tau \text{ and } slow \quad \} \\
 & \iota \times (\dot{\vee} \circ \iota \times slow.\tau.E \circ reorg)^\sigma \\
= & \{ \quad \text{property of } str \quad \} \\
 & \iota \times (\dot{\vee} \circ \iota \times str.\rho.E \circ reorg)^\sigma
\end{array}
$$

Thus we define

$$\upsilon.E^* \;=\; \iota \times (\dot{\vee} \circ \iota \times str.\rho.E \circ reorg)^\sigma$$

Finally, we get to the sequence operator. As mentioned earlier, this is where the context condition is needed:

$$
\begin{array}{ll}
& \upsilon.(E\,;F)\circ plumb\!< \;=\; \iota\times slow.\tau.(E\,;F)\circ plumb\!< \\
\equiv & \qquad \{ \qquad \text{definitions of } \tau \text{ and } slow \qquad \} \\
& \upsilon.(E\,;F)\circ plumb\!< \;=\; \iota\times(slow.\tau.F \circ \ll\vartriangle slow.\tau.E)\circ plumb\!< \\
\equiv & \qquad \{ \qquad plumb \text{ and sequence: (18)} \qquad \} \\
& \upsilon.(E\,;F)\circ plumb\!< \;=\; \iota\times slow.\tau.F\circ plumb \circ \iota\times slow.\tau.E \circ plumb\!< \\
\Leftarrow & \qquad \{ \qquad \text{Leibniz} \qquad \} \\
& \upsilon.(E\,;F) = \upsilon.F\circ plumb\circ \upsilon.E
\end{array}
$$

This concludes our derivation. Summarizing our results, the translation from regular expressions to systolic circuits is given by the equations:

$$
\begin{array}{rcll}
\rho.E & = & \upsilon.E\circ term & \\
\upsilon.t & = & \rhd\times(\lhd\circ\dot{\wedge}\circ(\dot{=}t)\times\iota) & \text{for all } t\in\mathcal{T} \\
\upsilon.(E{+}F) & = & \iota\times(\dot{\vee}\circ str.\rho.E\vartriangle str.\rho.F) & \\
\upsilon.(E\,;F) & = & \upsilon.F\circ plumb\circ\upsilon.E & \\
\upsilon.(E^{*}) & = & \iota\times(\dot{\vee}\circ\iota\times str.\rho.E\circ reorg)^{\sigma} &
\end{array}
$$

5 Conclusions

In the usual squiggol style, one works with syntactic terms that can be interpreted as both mathematical functions, and computer programs. What one does then is to take a term and transform it according to rules that do not change the functional interpretation, but may — and should — change the efficiency of the term interpreted as a program. What we did in the last section is very similar, except that instead of working with a simple term, we had to improve the efficiency of a term-valued function, τ. This is how functions like ρ come into being. Its characterisation as a function from relations to relations is simple; but it is not as easy to specify formally what we expect ρ to do as a function from syntactical terms to syntactical terms. What we had in mind as we worked is "apply the useful transformations as thoroughly as possible." It could prove fruitful to apply further work to develop notations for cleanly specifying term transformation functions of this kind.

An interesting element of our derivation is its use of the unique extension property (uep) for regular languages in the case of a starred expression. The fact that the subexpression should not include the empty word is a necessary and sufficient condition for application of the uep. This is where the error occurred in Foster and Kung's original paper. The non-uniqueness of solutions to certain equations in relation calculus corresponds to indeterminate behaviour in the corresponding circuits.

At the present stage of our work we are not completely content with the clarity and rigour of our derivation. We are very satisfied with the derivation of the non-systolic recognizer and with the division of the derivation into two phases. Our dissatisfaction is with the formal presentation of retiming and slowdown and, in particular, tying together individual calculations. We are currently endeavouring to eliminate these weaknesses.

A Proof of the delay and retiming laws

This section contains proofs of the properties of delay and, in particular, the retiming laws.

The proof of (2) is as follows:

$$
\begin{array}{ll}
 & \lhd{<} = \iota \\
\equiv & \{ \quad \text{definition of } \lhd,\ \iota \quad \} \\
 & (\mu(X \mapsto \partial \cup X \times X))< = \mu(X \mapsto \bar{\iota} \cup X \times X) \\
\Leftarrow & \{ \quad \text{fusion (see e.g. [10])} \quad \} \\
 & \forall(X :: \ (\partial \cup X \times X)< \ = \ \bar{\iota} \cup X{<} \times X{<}) \\
\Leftarrow & \{ \quad \text{domain calculus} \\
 & \qquad \text{(specifically, } < \text{ distributes through } \cup \text{ and } \times) \quad \} \\
 & \partial{<} = \bar{\iota} \ \wedge \ X{<} \times X{<} = X{<} \times X{<} \\
\equiv & \{ \quad \text{definition of } \partial,\ \bar{\iota} \quad \} \\
 & \mathit{true}
\end{array}
$$

The proof of the first part of (3) is as follows:

$$
\begin{array}{ll}
 & \lhd \circ \iota \times \iota \\
= & \{ \quad \text{definition of } \lhd \quad \} \\
 & (\partial \cup \lhd \times \lhd) \circ \iota \times \iota \\
= & \{ \quad \text{composition distributes over union} \quad \} \\
 & \partial \circ \iota \times \iota \ \cup \ \lhd \times \lhd \circ \iota \times \iota \\
= & \{ \quad \partial \text{ is not defined for pairs, (1)} \quad \} \\
 & \lhd \times \lhd
\end{array}
$$

The proof of the second half, and of the corresponding properties of $\rhd$, are similar.

The next calculation establishes property (8).

$$\rhd \circ \lhd = \iota$$

$$\equiv \quad \{ \quad \text{definition of } \iota \text{ and } \lhd \quad \}$$

$$\rhd \circ \mu(X \mapsto \partial \cup X \times X) = \mu(X \mapsto \bar{\iota} \cup X \times X)$$

$$\Leftarrow \quad \{ \quad \mu\text{-fusion (see, for example, [10])} \quad \}$$

$$\forall(X :: \ \rhd \circ (\partial \cup X \times X) = \bar{\iota} \cup (\rhd \circ X) \times (\rhd \circ X))$$

$$\Leftarrow \quad \{ \quad \text{calculus} \quad \}$$

$$\rhd \circ \partial = \bar{\iota} \wedge \forall(X :: \ \rhd \circ X \times X = (\rhd \circ X) \times (\rhd \circ X))$$

$$\equiv \quad \{ \quad \text{property (4)} \quad \}$$

$$\rhd \circ \partial = \bar{\iota}$$

$$\equiv \quad \{ \quad \rhd = \lhd^{\cup} = (\partial \cup \lhd \times \lhd)^{\cup} = \partial^{\cup} \cup \rhd \times \rhd \quad \}$$

$$(\partial^{\cup} \cup \rhd \times \rhd) \circ \partial = \bar{\iota}$$

$$\equiv \quad \{ \quad \partial \text{ is not defined on pairs} \quad \}$$

$$\partial^{\cup} \circ \partial = \bar{\iota}$$

This last formula can be proved pointwise: for all streams a,b,

$$a \langle \partial^{\cup} \circ \partial \rangle b$$

$$\equiv \quad \{ \quad \text{composition} \quad \}$$

$$\exists(c :: \ a\langle \partial^{\cup} \rangle c \wedge c \langle \partial \rangle b)$$

$$\equiv \quad \{ \quad \text{definition of primitive delay; calculus} \quad \}$$

$$\exists(c :: \ \forall(n :: \ a.n = c.(n+1) \wedge c.(n+1) = b.n))$$

$$\equiv \quad \{ \quad \text{calculus} \quad \}$$

$$\forall(n :: \ a.n = b.n)$$

The second equality in (8) can be proved by means of a very similar proof.

The proof of the retiming laws, equations (6) and (7). It is by structural induction on the "means for constructing circuits" enumerated in section 1.

We begin by proving that (6) holds for any lifted relation $\dot{R}$. For all streams a and b :

$$a \langle \lhd \circ \dot{R} \rangle b$$

$$\equiv \quad \{ \quad \text{composition} \quad \}$$

$$\exists(c :: \ a \langle \lhd \rangle c \wedge c \langle \dot{R} \rangle b)$$

$$\equiv \quad \{ \quad \text{definitions of } \lhd \text{ and } R\text{; calculus} \quad \}$$

$$
\begin{array}{ll}
 & \exists(c ::\ \forall(n ::\ a.(n+1) = c.n \wedge c.n\,\langle R\rangle\, b.n)) \\
\equiv & \qquad \{ \qquad \text{calculus} \qquad \} \\
 & \forall(n ::\ a.(n+1)\,\langle R\rangle\, b.n) \\
\equiv & \qquad \{ \qquad \text{calculus} \qquad \} \\
 & \exists(c ::\ \forall(n ::\ a.(n+1)\,\langle R\rangle\, c.(n+1)\ \wedge\ c.(n+1) = b.n)) \\
\equiv & \qquad \{ \qquad \text{definitions of } \lhd \text{ and } R\text{; calculus} \qquad \} \\
 & \exists(c ::\ a\langle \dot{R}\rangle c \wedge c\langle \lhd\rangle b) \\
\equiv & \qquad \{ \qquad \text{composition} \qquad \} \\
 & a\langle \dot{R} \circ \lhd\rangle b
\end{array}
$$

Next we show that (6) holds for the left projection. First, we need a small lemma:

$$
\begin{array}{ll}
 & (a,b)\langle \lhd\rangle(c,d) \\
\equiv & \qquad \{ \qquad \text{by the definition of } \lhd \qquad \} \\
 & (a,b)\langle \partial \cup \lhd \times \lhd\rangle(c,d) \\
\equiv & \qquad \{ \qquad \text{definition of union} \qquad \} \\
 & (a,b)\langle \partial\rangle(c,d) \vee (a,b)\langle \lhd \times \lhd\rangle(c,d) \\
\equiv & \qquad \{ \qquad \partial \text{ is not defined on pairs; definition of product} \qquad \} \\
 & a\langle \lhd\rangle c \wedge b\langle \lhd\rangle d
\end{array}
$$

Hence, it holds that $(a,b)\langle \lhd\rangle(c,d) \equiv a\langle \lhd\rangle c \wedge b\langle \lhd\rangle d$. We may now proceed: for all streams a,b and c,

$$
\begin{array}{ll}
 & a\langle \lhd \circ \ll\rangle(b,c) \equiv a\langle \ll \circ \lhd\rangle(b,c) \\
\equiv & \qquad \{ \qquad \text{composition} \qquad \} \\
 & \exists(d ::\ a\langle \lhd\rangle d \wedge d\langle \ll\rangle(b,c)) \equiv \exists(d,e ::\ a\langle \ll\rangle(d,e) \wedge (d,e)\langle \lhd\rangle(b,c)) \\
\equiv & \qquad \{ \qquad \text{definition of } \ll\text{, twice, and above lemma:} \qquad \} \\
 & \exists(d ::\ a\langle \lhd\rangle d \wedge d = b) \equiv \exists(d,e ::\ a = d \wedge d\langle \lhd\rangle b \wedge e\langle \lhd\rangle c) \\
\Leftarrow & \qquad \{ \qquad \text{calculus} \qquad \} \\
 & a\langle \lhd\rangle b \equiv a\langle \lhd\rangle b \\
\equiv & \qquad \{ \qquad \text{calculus} \qquad \} \\
 & \mathit{true}
\end{array}
$$

The proof that (6) holds for the right projection is entirely similar. That (6) holds for $R := \lhd$ is trivial; for $R := \rhd$ it is a consequence of (8).

Note that all the proofs we have given until now can be easily modified to prove the corresponding properties for antidelay. We will then assume that the reader is convinced that *both* (6) and (7) hold for lifting, projections, and delays.

Suppose now that both (6) and (7) hold for circuits R and S. We then have:

$$\begin{array}{ll}
& \lhd \circ R \circ S \\
= & \quad \{ \quad \text{hypothesis on } R \quad \} \\
& R \circ \lhd \circ S \\
= & \quad \{ \quad \text{hypothesis on } S \quad \} \\
& R \circ S \circ \lhd
\end{array}$$

So much for composition. For product we have:

$$\begin{array}{ll}
& \lhd \circ R \times S \\
= & \quad \{ \quad \text{equation (4)} \quad \} \\
& (\lhd \circ R) \times (\lhd \circ S) \\
= & \quad \{ \quad \text{hypothesis on } R \text{ and } S \quad \} \\
& (R \circ \lhd) \times (S \circ \lhd) \\
= & \quad \{ \quad \text{equation (4)} \quad \} \\
& R \times S \circ \lhd
\end{array}$$

Similarly, for split:

$$\begin{array}{ll}
& \lhd \circ R \vartriangle S \\
= & \quad \{ \quad \text{equation (4)} \quad \} \\
& (\lhd \circ R) \vartriangle (\lhd \circ S) \\
= & \quad \{ \quad \text{hypothesis on } R \text{ and } S \quad \} \\
& (R \circ \lhd) \vartriangle (S \circ \lhd) \\
= & \quad \{ \quad \text{equation (5)} \quad \} \\
& R \vartriangle S \circ \lhd
\end{array}$$

For converse, we have

$$\begin{array}{ll}
& \lhd \circ R^{\cup} \\
= & \quad \{ \quad \text{converse; } \rhd \text{ is the converse of } \lhd \quad \}
\end{array}$$

$$
\begin{array}{cl}
& (R \circ \rhd)^{\cup} \\
= & \{ \quad \text{by hypothesis (7) holds for } R \quad \} \\
& (\rhd \circ R)^{\cup} \\
= & \{ \quad \text{converse} \quad \} \\
& R^{\cup} \circ \lhd
\end{array}
$$

The proof for feedback is more complicated, and it makes use of definitions and theorems from Frans Rietman's thesis [11, pages 23–25]. We summarise here what we need. The *loop* of a relation R, denoted by R^{ϖ}, is defined by $a\langle R^{\varpi}\rangle b \equiv \exists(c :: \ (a,c)\langle R\rangle(b,c))$. Loop and feedback enjoy the following properties:

$$
\begin{array}{lcll}
R^{\sigma} & = & (\iota \vartriangle \iota \circ R)^{\varpi} & \text{loop-feedback} \\
R \circ S^{\varpi} \circ T & = & (R \times \iota \circ S \circ T \times \iota)^{\varpi} & \text{loop fusion} \\
(\iota \times R \circ S)^{\varpi} & = & (S \circ \iota \times R)^{\varpi} & \text{loop leapfrog}
\end{array}
$$

We are now ready for the last part of the proof:

$$
\begin{array}{cl}
& \lhd \circ R^{\sigma} \\
= & \{ \quad \text{loop-feedback} \quad \} \\
& \lhd \circ (\iota \vartriangle \iota \circ R)^{\varpi} \\
= & \{ \quad \text{loop fusion, eq. (1)} \quad \} \\
& (\lhd \vartriangle \iota \circ R)^{\varpi} \\
= & \{ \quad \text{equations (5) and (8)} \quad \} \\
& (\iota \vartriangle \rhd \circ \lhd \circ R)^{\varpi} \\
= & \{ \quad \text{hypothesis on } R \quad \} \\
& (\iota \vartriangle \rhd \circ R \circ \lhd)^{\varpi} \\
= & \{ \quad \text{equation (1)} \quad \} \\
& (\iota \times \rhd \circ \iota \vartriangle \iota \circ R \circ \lhd)^{\varpi} \\
= & \{ \quad \text{loop leapfrog} \quad \} \\
& (\iota \vartriangle \iota \circ R \circ \lhd \circ \iota \times \rhd)^{\varpi} \\
= & \{ \quad \text{equations (4) and (8)} \quad \} \\
& (\iota \vartriangle \iota \circ R \circ \lhd \times \iota)^{\varpi} \\
= & \{ \quad \text{loop fusion} \quad \} \\
& (\iota \vartriangle \iota \circ R)^{\varpi} \circ \lhd \\
= & \{ \quad \text{loop-feedback} \quad \}
\end{array}
$$

$R^\sigma \circ \lhd$

This concludes the proof of (6) and (7).

B Proof of the properties of *mem*

We prove the laws claimed in section 2 about mem, equations (12). Note that we will write e.g., $E+F$ to mean both the regular expression, and the language it denotes. The context should make clear which one we intend. In the remainder of this section, we let a stand for any stream of characters from $\mathcal{T}$. Letting $t \in \mathcal{T}$, we have

$$
\begin{array}{ll}
 & m\langle mem.(t,a)\rangle n \\
\equiv & \{ \quad \text{definition} \quad \} \\
 & a[n,m) \in t \\
\equiv & \{ \quad \text{here } t \text{ denotes the language } \{t\} \quad \} \\
 & a.n = t \wedge m = n+1
\end{array}
$$

So much for the base case. Now, for the "choice" operator we have:

$$
\begin{array}{ll}
 & m\langle mem.(E+F,a)\rangle n \\
\equiv & \{ \quad \text{definition of } mem \quad \} \\
 & a[n,m) \in E+F \\
\equiv & \{ \quad \text{regular expressions} \quad \} \\
 & a[n,m) \in E \vee a[n,m) \in F \\
\equiv & \{ \quad \text{definition of } mem\text{, twice} \quad \} \\
 & m\langle mem.(E,a)\rangle n \vee m\langle mem.(F,a)\rangle n \\
\equiv & \{ \quad \text{union of relations} \quad \} \\
 & m\langle mem.(E,a) \cup mem.(F,a)\rangle n
\end{array}
$$

Similarly, for composition:

$$
\begin{array}{ll}
 & m\langle mem.(E\,;F,a)\rangle n \\
\equiv & \{ \quad \text{definition of } mem \quad \} \\
 & a[n,m) \in E\,;F \\
\equiv & \{ \quad \text{regular expressions} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & \exists(k :: \ a[n,k)\in E \wedge a[k,m)\in F) \\
\equiv & \{ \quad \text{definition of } mem\text{, twice} \quad \} \\
 & \exists(k :: \ k\langle mem.(E,a)\rangle n \wedge m\langle mem.(F,a)\rangle k) \\
\equiv & \{ \quad \text{composition of relations} \quad \} \\
 & m\langle mem.(F,a) \circ mem.(E,a)\rangle n
\end{array}
$$

Finally, for "star" we have:

$$
\begin{array}{ll}
 & mem.(E^*,a) = mem.(E,a)^* \\
\equiv & \{ \quad \text{definitions of "star" on languages and relations} \quad \} \\
 & mem.(\mu(X \mapsto \varepsilon + X\,;E),a) = \mu(X \mapsto I \cup X \circ mem.(E,a)) \\
\Leftarrow & \{ \quad E \mapsto mem.(E,a) \text{ is universally } \cup\text{-distributive} \\
 & \quad\quad \text{so we may use } \mu\text{-fusion [10]} \quad \} \\
 & \forall(X :: \ mem.(\varepsilon + X\,;E,a) = I \cup mem.(X,a) \circ mem.(E,a)) \\
\equiv & \{ \quad \text{above} \quad \} \\
 & true
\end{array}
$$

This concludes the proof of the properties of *mem*.

References

[0] Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. Available at URL `ftp://ftp.win.tue.nl/pub/math.prog.construction/book.dvi`, December 1992.

[1] Roland C. Backhouse. Specification and proof of a regular language recognizer in synchronous CCS. Technical Report CSM-53, University of Essex, 1983.

[2] Henk Doornbos, Roland Backhouse, and Jaap van der Woude. A calculational approach to mathematical induction. To appear in Theoretical Computer Science, 1997.

[3] Michael J. Foster. *Specialized Silicon Compilers for Language Recognition.* PhD thesis, Computer Science Department, Carnegie Mellon University, 1984.

[4] M.J. Foster and H.T. Kung. Recognize regular languages with programmable building blocks. *Journal of Digital Systems*, 4(6):323–332, 1982.

[5] M.J. Gordon. Why higher-order is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*. North-Holland, 1986.

[6] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørge Staunstrup, editor, *Formal Methods for VLSI Design. IFIP WG 10.5 Lecture Notes*. North-Holland, 1990.

[7] A. Kaldewaij and G. Zwaan. A systolic design for acceptors of regular languages. *Science of Computer Programming*, 15(2):171–183, 1990.

[8] Charles E. Leiserson. *Area-Efficient VLSI Computation*. MIT Press, 1983.

[9] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[10] Eindhoven University of Technology Mathematics of Program Construction Group. Fixed point calculus. *Information Processing Letters*, 53(3):131–136, February 1995.

[11] Frans Rietman. *A Relational Calculus for the Design of Distributed Algorithms*. PhD thesis, University of Utrecht, 1995.

[12] Ole Sandum. Multiple clocks and Ruby. Technical Report ID–TR: 1994-153, Dept. of Computer Science, Technical University of Denmark, August 1994. Available by FTP at `ftp://ftp.it.dtu.dk/pub/Ruby/mcar.ps.Z`.