3

Deriving a systolic regular language recognizer

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

*Roland Backhouse**
Department of Mathematics and Computing Science, Eindhoven University of Technology
PO Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: rolandb@win.tue.nl

Abstract

We present a derivation of a regular language recognizing circuit originally developed by Foster and Kung. We make use of point-free relation algebra, in a style combining elements from earlier work in the Eindhoven Mathematics of Program Construction group, and from Sheeran and Jones' work on Ruby. 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.

Keywords

Regular languages, circuit derivation, program calculation, systolic circuits, relation algebra, relation calculus, retiming, slowdown, silicon compilation

0 INTRODUCTION

In 1982, Foster and Kung 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 (1983) 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

*Our thanks to Professor Giovanni Degli Antoni and SGS-Thomson for supporting mutual visits which have made this collaboration possible.

discovery of an error in Foster and Kung's construction — acknowledged by Foster in his Ph.D. thesis (1984). 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", Jones and Sheeran 1990).

A precise definition of "systolic" can be found in Leiserson's thesis (1983). 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 (1984). Both the specification and the implementation have been adapted in order to overcome the error in (Foster and Kung 1982). 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 formal derivation of a similar compiler has also been given by Kaldewaij and Zwaan (1990), 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 two sections, we introduce the reader to relation algebra, and our definition of "circuit". In section 3 we show how to specify the problem with relation algebra. Then in section 4 we derive a first, non-systolic version of the recognizers. In section 5 standard circuit transformations are applied to obtain a systolic version. Finally, in section 6 some conclusions are drawn.

*The clock period is the time between two clock ticks.

1 ABOUT 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 (Aarts et al. 1992).

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 $\bot\!\!\!\bot$, 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 $n\langle\mathbb{N}\rangle m \equiv n{=}m \wedge (n \text{ is a natural number})$. 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 $R\triangle S$ (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 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

$$R \vartriangle S \circ f = (R \circ f) \vartriangle (S \circ f) \tag{0}$$

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:

$$\begin{array}{lcl} R \times S \circ T \times U & = & (R \circ T) \times (S \circ U) \\ R \times S \circ T \vartriangle U & = & (R \circ T) \vartriangle (S \circ U). \end{array} \tag{1}$$

To complete this brief survey of relation algebra we must introduce the reflexive, transitive closure of a relation (see e.g. Doornbos, Backhouse and van der Woude 1997), which may be defined for relation R as the least fixed point of the function $X \mapsto I \cup R \circ X$:

$$R^* = \mu(X \mapsto I \cup R \circ X).$$

In (Doornbos et al. 1997) it is proved that the so-called "unique extension property" (uep) of the reflexive, transitive closure:

$$R = S^* \circ T \equiv R = T \cup S \circ R \tag{2}$$

holds if S is *well-founded* and, furthermore, S is well-founded if it enjoys the property $X = S \circ X \Rightarrow X = \bot\!\!\!\bot$ for all relations X.

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.

2 ABOUT CIRCUITS

Following established practice (see Cohn and Gordon 1990, Jones and Sheeran 1990, Rietman 1995) we model a circuit as a relation between arbitrary collections of *streams*, a stream being a total function with domain the integer

$\cup < > ^* {}^\sigma$	all unary operators
.	function application
$\times + \triangle$	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

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 (for details see Jones and Sheeran 1990). 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, $\dot{f}.a.m = (f.a).m$. 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 ar-

bitrary pairings of streams. It is defined as the least fixed point of function $X \mapsto \partial \cup X \times X$:

$$\triangleleft = \mu(X \mapsto \partial \cup X \times X).$$

Delay can be thought of informally as the union of an infinite list of terms $\triangleleft = \partial \cup \partial \times \partial \cup \partial \times (\partial \times \partial) \cup (\partial \times \partial) \times \partial \cup (\partial \times \partial) \times (\partial \times \partial) \cup \ldots$. The *antidelay* $\triangleright$ is defined to be the converse of delay. In the interpretation as circuits, a delay is a memory element that, at every clock tick, outputs the contents of memory on the left side and 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 $\triangleleft$ and $\triangleright$ 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,

$$\triangleleft< = \triangleleft> = \iota = \triangleright< = \triangleright> \tag{3}$$

(Note that this and other properties of delays are proved in a longer version of this paper published as a technical report. Vaccari and Backhouse 1996)

A similar domain property that we use frequently is: for $\diamond \in \{\triangleleft, \triangleright\}$,

$$\diamond \circ \iota \times \iota = \diamond \times \diamond = \iota \times \iota \circ \diamond. \tag{4}$$

These equations express the fact that applying delay or antidelay to a pair of (collections of) wires ($\diamond \circ \iota \times \iota$) is the same as applying it to each component of the pair ($\diamond \times \diamond$). From this property, one immediately obtains the following useful distributivity properties: for $\diamond \in \{\triangleleft, \triangleright\}$,

$$\begin{array}{lcl} \diamond \circ R \times S & = & (\diamond \circ R) \times (\diamond \circ S) \\ R \times S \circ \diamond & = & (R \circ \diamond) \times (S \circ \diamond) \\ \diamond \circ R \vartriangle S & = & (\diamond \circ R) \vartriangle (\diamond \circ S) \end{array} \tag{5}$$

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

$$R \triangle S \circ \Diamond = (R \circ \Diamond) \triangle (S \circ \Diamond).$$

A particular wiring relation is *term* (for "terminator"), defined by

$$term = \iota \triangle \iota \circ \top\top.$$

It is easy to calculate that *term* satisfies, for all a, b and c,

$$(a, b)\langle term \rangle c \equiv a{=}b.$$

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. *term* is a circuit.
3. If R,S are circuits, then $R \circ S$, $R \times S$, $R \triangle S$, and $R^\cup$ are circuits.
4. Delays and antidelays are circuits.
5. 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 four 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. Figure 0 shows the correspondence between pictures and circuit terms. A picture shows which "parts" of a circuit are connected; and interconnections are important in order to evaluate the circuit's performance. In fact, 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 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 (Leiserson 1983).

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

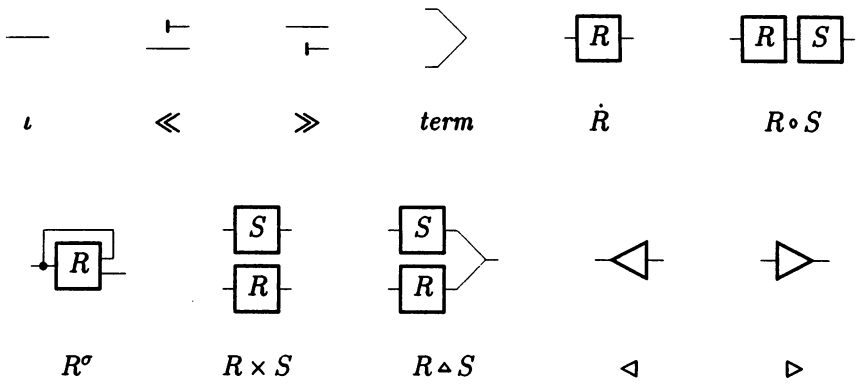


Figure 0 Circuits and their pictures

Retiming (Leiserson and Saxe 1991) is a transformation that is essentially based on the following laws: given that R is a circuit as defined above,

$$\triangleleft \circ R = R \circ \triangleleft \quad \text{and} \quad \triangleright \circ R = R \circ \triangleright. \tag{6}$$

These laws can be proved by structural induction (see the longer version of this paper, Vaccari and Backhouse 1996). Combining (6) with the property that

$$\triangleright \circ \triangleleft = \iota = \triangleleft \circ \triangleright \tag{7}$$

we obtain the properties

$$R = \triangleright \circ R \circ \triangleleft \quad \text{and} \quad R = \triangleleft \circ R \circ \triangleright \tag{8}$$

for all circuits R.

(Note that the retiming law (8) breaks 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.)

A useful property of *term* is that, for all circuits R and $\diamond \in \{\triangleleft, \triangleright\}$,

$$\diamond \circ R \circ term = R \circ term. \tag{9}$$

This is a straightforward consequence of the retiming equations (6) and the domain equations (3). An intuitive understanding of why this law is true can be gained by comparing the picture interpretation of (9), for $R = \iota$, with the

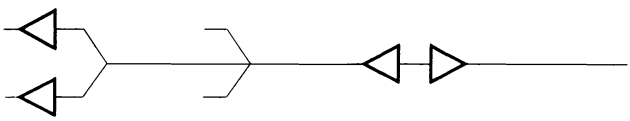


Figure 1 Comparing instances of (9) and (8)

picture interpretation (8), also for $R = \iota$ (see figure 1): the pictures are the same, modulo the orientation of the wires.

Another optimisation technique is slowdown (Jones and Sheeran 1992). Given a circuit R, the circuit $slow.R$ is defined by

$$slow.R = \mathcal{B} \circ R \times R \circ \mathcal{B}\cup,$$

where $\mathcal{B}$, pronounced "bundle", is defined by

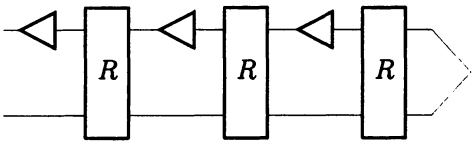
$$a\langle\mathcal{B}\rangle(b,c) \equiv \forall(n :: b.n = a.(2n) \wedge c.n = a.(2n+1)).$$

It can be shown by structural induction that, for any circuit R, the circuit $slow.R$ is equal to the one obtained by replacing every occurrence of $\triangleright$ and $\triangleleft$ in R by $\triangleright\circ\triangleright$ and $\triangleleft\circ\triangleleft$, 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.

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

$$(\iota \times \triangleleft \circ R)^n \circ term \tag{10}$$

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

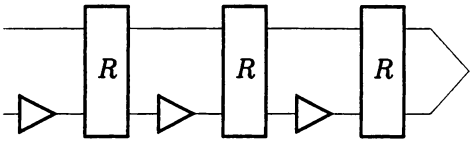


By retiming, one obtains:

(10)

$$
\begin{array}{ll}
= & \{ \quad \text{retiming (8)} \quad \} \\
& (\triangleright \circ \iota \times \triangleleft \circ R \circ \triangleleft)^n \circ \mathit{term} \\
= & \{ \quad \text{delays, (5) and (7)} \quad \} \\
& (\triangleright \times i \circ R \circ \triangleleft)^n \circ \mathit{term} \\
= & \{ \quad \text{retiming (6)} \quad \} \\
& (\triangleright \times \iota \circ R)^n \circ \triangleleft^n \circ \mathit{term} \\
= & \{ \quad \text{equation (9)} \quad \} \\
& (\triangleright \times \iota \circ R)^n \circ \mathit{term}.
\end{array}
$$

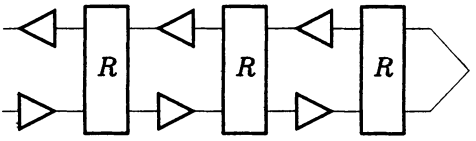
This transformation does not buy us anything, since the resulting circuit still has a long combinational delay:



But, if we choose to implement a slowed version of (10) instead, we have:

$$
\begin{array}{ll}
& \mathit{slow}.((\iota \times \triangleleft \circ R)^n \circ \mathit{term}) \\
= & \{ \quad \text{slowing is the same as doubling the delays;} \\
& \quad \text{assume } R \text{ is combinational} \quad \} \\
& (\iota \times (\triangleleft \circ \triangleleft) \circ R)^n \circ \mathit{term} \\
= & \{ \quad \text{fusion (1)} \quad \} \\
& (\iota \times \triangleleft \circ \iota \times \triangleleft \circ R)^n \circ \mathit{term} \\
= & \{ \quad \text{the above derivation,} \\
& \quad \text{taking } R := \iota \times \triangleleft \circ R \quad \} \\
& (\triangleright \times \iota \circ \iota \times \triangleleft \circ R)^n \circ \mathit{term} \\
= & \{ \quad \text{fusion (1)} \quad \} \\
& (\triangleright \times \triangleleft \circ R)^n \circ \mathit{term}.
\end{array}
$$

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. Note that the placement of delays and antidelays 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".

We conclude this section with a remark on the realizability of the circuits we derive. In the algebra of relations there is no notion of "input" or "output". When one wishes to implement a relation algebra term as an actual circuit, input and output directions must be assigned to each wire. But not all choices yield an implementable circuit. For instance, if *add* is a relation such that $a\langle add\rangle(b,c) \equiv a = b+c$, then choosing a as input and b, c as output would yield a non-deterministic circuit. One must pay particular attention to delays and antidelays, since they can be implemented in just one way: every delay must have input on the right side, and output on the left side; and the other way around for antidelays. If there is no way to choose inputs and outputs such that every delay and antidelay is driven in the correct direction, then the circuit is not implementable. A more formal and thorough discussion of implementability is given in Graham Hutton's thesis (1993).

3 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

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

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

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

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 enable bit) to range over a stream of input booleans.

In order to avoid the error in (Foster and Kung 1982) 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 $e.n \equiv n > 0 \wedge (n \text{ is odd})$, then $tt.e = \{1, 3, 5, \ldots\}$. We also introduce a relation, $mem.(E, a)$, on integers 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 $a.n \ ; \ a.(n+1) \ ; \ \ldots \ ; \ a.(m-1)$. 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 $a.n = t$ 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 Vaccari and Backhouse 1996):

$$\begin{array}{rcll} m\langle mem.(t, a)\rangle n & \equiv & m = n+1 \ \wedge \ a.n = t & \text{for all } t \in \mathcal{T} \\ mem.(E+F, a) & = & mem.(E, a) \cup mem.(F, a) & \\ mem.(E\,;F, a) & = & mem.(F, a) \circ mem.(E, a) & \\ mem.(E^*, a) & = & (mem.(E, a))^*. & \end{array} \tag{11}$$

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 $e \in Stream(\mathbf{B})$:

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

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.

4 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:

$$m \in mem.(t, a) \circ tt.e$$

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

From this we obtain:

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

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,

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

$$
\begin{array}{ll}
 & \forall(a,e :: h.(a,e) = f.(a,e)\ \dot{\vee}\ g.(a,e)) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & h = \dot{\vee} \circ f \vartriangle g.
\end{array}
$$

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,

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

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 well-founded. 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:

$$
\begin{array}{ll}
 & X = mem.(E,a) \circ X \\
\Rightarrow & \quad \{ \quad \text{Leibniz} \quad \} \\
 & X \circ \top\!\!\top = mem.(E,a) \circ X \circ \top\!\!\top \\
\equiv & \quad \{ \quad \text{pointwise interpretation; define } S = X \circ \top\!\!\top, \\
 & \qquad \text{a left condition which we regard as a set} \quad \} \\
 & \forall(m :: m \in S \equiv \exists(n :: m\langle mem.(E,a)\rangle n \wedge n \in S)) \\
\equiv & \quad \{ \quad \bullet \text{ assume } mem.(E,a) > \subseteq \mathbf{N} \quad \} \\
 & \forall(m :: m \in S \equiv \exists(n : n \in \mathbf{N} : m\langle mem.(E,a)\rangle n \wedge n \in S)) \\
\equiv & \quad \{ \quad \text{definition of } mem \quad \} \\
 & \forall(m :: m \in S \equiv \exists(n : n \in \mathbf{N} : a[n,m) \in E \wedge n \in S)) \\
\Rightarrow & \quad \{ \quad \bullet \text{ assume } \varepsilon \notin E \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & \forall(m ::\ m \in S \equiv \exists(n :\ n \in \mathbf{N} :\ n < m \wedge n \in S)) \\
\Rightarrow & \quad \{ \quad \text{predicate calculus} \quad \} \\
 & \forall(m :\ m \in \mathbf{N} :\ m \in S \equiv \exists(n :\ n \in \mathbf{N} :\ n < m \wedge n \in S)) \\
 & \wedge\ S \subseteq \mathbf{N} \\
\equiv & \quad \{ \quad \text{the natural numbers are well-founded} \quad \} \\
 & S = \perp\!\!\!\perp \\
\equiv & \quad \{ \quad \text{calculus, } S = X \circ \top\!\!\!\top \quad \} \\
 & X = \perp\!\!\!\perp .
\end{array}
$$

We have thus found that the assumptions $\varepsilon \notin E$ and $mem.(E,a) > \subseteq \mathbf{N}$ 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

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

Now,

$$
\begin{array}{ll}
 & tt.b = mem.(E^*,a) \circ tt.e \\
\equiv & \quad \{ \quad mem\ (11) \quad \} \\
 & tt.b = (mem.(E,a))^* \circ tt.e \\
\equiv & \quad \{ \quad \varepsilon \notin E, \text{ so } mem.(E,a) \text{ is well-founded.} \\
 & \qquad \text{Unique extension property (2).} \quad \} \\
 & tt.b \ = \ tt.e \cup mem.(E,a) \circ tt.b \\
\equiv & \quad \{ \quad f \text{ recognizes } E \quad \} \\
 & tt.b = tt.e \cup tt.f.(a,b) \\
\equiv & \quad \{ \quad tt \quad \} \\
 & tt.b = tt.(e \dot{\vee} f.(a,b)) \\
\equiv & \quad \{ \quad tt \text{ is an isomorphism} \quad \} \\
 & b = e \dot{\vee} f.(a,b)
\end{array}
$$

$$\equiv \quad \{ \quad \text{calculus; define } reorg.((x,y),z) = (y,(x,z)) \quad \}$$
$$b = (\dot{\vee} \circ \iota \times f \circ reorg).((a,e),b).$$

Thus

$$\begin{array}{ll} & g \text{ recognizes } E^* \\ \equiv & \{ \quad \text{above} \quad \} \\ & \forall (a,e,b :: \ b\langle g\rangle(a,e) \ \equiv \ b = (\dot{\vee} \circ \iota \times f \circ reorg).((a,e),b)) \\ \equiv & \{ \quad \text{feedback} \quad \} \\ & g = (\dot{\vee} \circ \iota \times f \circ reorg)^{\sigma}. \end{array}$$

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:

$$\begin{array}{rcll} \tau.t & = & \triangleleft \circ \dot{\wedge} \circ (\dot{=} t) \times \iota & \text{for all } t \in \mathcal{T} \\ \tau.(E+F) & = & \dot{\vee} \circ \tau.E \vartriangle \tau.F & \\ \tau.(E\,;F) & = & \tau.F \circ \ll \vartriangle \tau.E & \\ \tau.E^* & = & (\dot{\vee} \circ \iota \times \tau.E \circ reorg)^{\sigma} & \end{array} \tag{12}$$

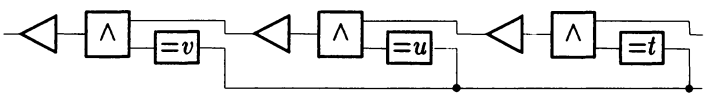
and such that, for all regular expressions E, $\tau.E$ recognizes E.

5 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

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

It is apparent from the picture interpretation of the above circuit



that there is a combinational path from one side to the other of the circuit. Even worse is the fact that for every string recognizer, this path grows in length with the length of the string.

Our strategy for making the design systolic is in three steps. First we rearrange the wires, in order to introduce contra-flow in the circuits (see section 2). Then we make the design modular, by designing a separate cell for each operator (except for sequence, which we regard as the "basic" oper-

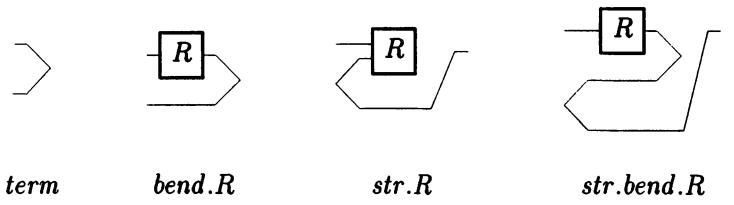


Figure 2 Picture interpretations

ator). Finally we apply retiming and slowdown, in order to make the design as systolic as possible.

As we saw in the example at the end of section 2, a standard way to make a circuit systolic is to apply slowdown and retiming. However, the transformation shown in that example only works when the circuit has contra-flow; that is, when there are two parallel wires where data travel in opposite directions. Given a circuit R, a simple way to introduce contra-flow in it is to implement $bend.R$ instead of R, defined by

$$bend.R = \iota \times R \circ term.$$

It is easy to calculate that, for all a, b and c,

$$(a,b)\langle bend.R\rangle c \;\equiv\; b\langle R\rangle a$$

(Note the inversion of a and b, and that c does not appear on the right side.) 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

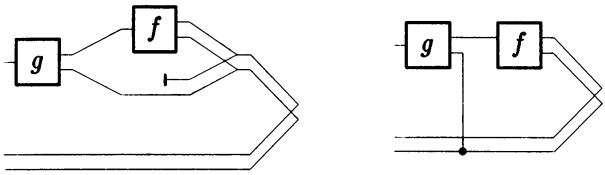
$$str.bend.R = R \tag{13}$$

(see figure 2). Since $bend$ is defined without mentioning delays, we have that for all R,

$$slow.bend.R = bend.slow.R. \tag{14}$$

Given the above discussion, we propose to change the specification so that instead of implementing $\tau.E$ we decide to implement $\rho.E$ instead, defined by

$$\rho.E = slow.bend.\tau.E.$$



$\iota \times g \circ \iota \times (\ll \triangle f) \circ term$ $\qquad$ $\iota \times g \circ plumb \circ \iota \times f \circ term$

Figure 3 A simplification

Note that in the definition of ρ we foresee the need to apply slowdown; hence the occurrence of *slow*. So much for the introduction of contra-flow.

Another strategy for systolic design is to break the design down into small modules or "cells" of the same "shape", connected in some regular way. Taking inspiration from Foster and Kung, we concentrate on sequence. Consider the expression $\rho.(E\,;F)$. Its picture interpretation suggests a simplification: see figure 3. Let *plumb* be a wiring relation defined by

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

It holds that

$$\iota \times (g \circ \ll \triangle f) \circ term \;=\; \iota \times g \circ plumb \circ \iota \times f \circ term \tag{15}$$

(The proof is omitted. A simple pointwise calculation establishes the property.) By defining a new function $v.E = \iota \times slow.\tau.E$, equation (15) may be rewritten as

$$\rho.(E\,;F) \;=\; v.F \circ plumb \circ \rho.E. \tag{16}$$

This result provides the insight for our next step in the process of making the design systolic, i.e. the breaking down of the design into cells. We choose sequence as the "privileged" operator in our design, and implement it by means of equation (16). Every regular expression E other than a sequence will be implemented by a "cell" equal to $v.E$. For instance, the interconnection of the cells of the recognizer for the expression $t^*\ ;\ v\ ;(t+(u\,;u))$ would have the shape shown in figure 4.

What we are left to do is to come up with designs for the cells $v.(E+F)$, $v.E^*$ and $v.t$, for all $t \in \mathcal{T}$. Let E, F be regular expressions; we calculate for choice:

$$v.(E+F)$$

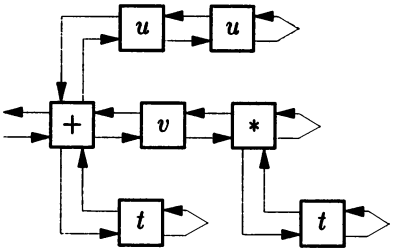


Figure 4 Cells layout for the recognizer of $t^*\ ;\ v\ ;(t+(u\,;u))$

$$
\begin{array}{ll}
= & \{\quad \text{definition of } v \quad\} \\
& \iota \times \mathit{slow}.\tau.(E+F) \\
= & \{\quad \text{definition of } \tau \quad\} \\
& \iota \times \mathit{slow}.(\dot{\vee} \circ \tau.E \vartriangle \tau.F) \\
= & \{\quad \text{slowing is the same as doubling delays} \quad\} \\
& \iota\times(\dot{\vee} \circ \mathit{slow}.\tau.E \vartriangle \mathit{slow}.\tau.F) \\
= & \{\quad \text{equation (13)} \quad\} \\
& \iota\times(\dot{\vee} \circ \mathit{str.bend}.\mathit{slow}.\tau.E \vartriangle \mathit{str.bend}.\mathit{slow}.\tau.F) \\
= & \{\quad \text{slow commutes with bend (14)} \quad\} \\
& \iota\times(\dot{\vee} \circ \mathit{str.slow.bend}.\tau.E \vartriangle \mathit{str.slow.bend}.\tau.F) \\
= & \{\quad \text{definition of } \rho \quad\} \\
& \iota\times(\dot{\vee} \circ \mathit{str}.\rho.E \vartriangle \mathit{str}.\rho.F)
\end{array}
$$

Note that in the above derivation we have exploited the property $\mathit{slow}.\tau.E = \mathit{str}.\rho.E$ in order to have ρ reappear. By doing so we ensure that the systolic optimizations can be applied recursively.

We have a similar derivation for *star*:

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

Finally, for any $t \in \mathcal{T}$, we have

$$
\begin{array}{ll}
& v.t \\
= & \{\quad \text{definitions of } \rho,\ \tau \text{ and } \mathit{slow} \quad\}
\end{array}
$$

$$
\begin{array}{ll}
 & \iota\times(\lhd\circ\lhd\circ\dot{\wedge}\circ(\dot{=}t)\times\iota) \\
= & \quad \{ \quad \text{retiming (6) and delays (7), (5)} \quad \} \\
 & \rhd\times(\lhd\circ\dot{\wedge}\circ(\dot{=}t)\times\iota)\circ\lhd.
\end{array}
$$

Here we'd like to get rid of the rightmost $\lhd$, but with the current definition of v we can't. However, the circuits we generate are always of the shape $v.E \circ \ldots \circ \rho.F$, and by the definition of ρ and equation (9) we have $\lhd \circ \rho.F = \rho.F$. So it's easy to see that dropping the delay is harmless. Formally, all we have to do is to weaken the defining property of v to

$$\forall(R :: v.E \circ R \circ \top\!\!\top = \iota \times \mathit{slow}.\tau.E \circ R \circ \top\!\!\top).$$

By this definition, the expressions for $v.(E+F)$ and $v.E^*$ that we previously derived are still valid, as well as equation (16); and the derivation of $v.t$ becomes:

$$
\begin{array}{ll}
 & v.t \circ R \circ \top\!\!\top \\
= & \quad \{ \quad \text{above} \quad \} \\
 & \rhd\times(\lhd\circ\dot{\wedge}\circ(\dot{=}t)\times\iota)\circ\lhd\circ R\circ\top\!\!\top \\
= & \quad \{ \quad \text{equation (9)} \quad \} \\
 & \rhd\times(\lhd\circ\dot{\wedge}\circ(\dot{=}t)\times\iota)\circ R\circ\top\!\!\top.
\end{array}
$$

Summarizing our results, we have

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

These equations can be interpreted as a functional program that translates regular expressions to systolic recognizers.

6 CONCLUSIONS

The transformation ρ that we derived in the last section generates circuits that are very similar to the ones presented by Foster and Kung. If the regular expression to be recognized does not contain choice or star, i.e. it is just a string of characters, then ρ generates a fully systolic string recognizer. The picture interpretation of the recognizer for the string $t\,;u\,;v$ can be seen in figure 5.

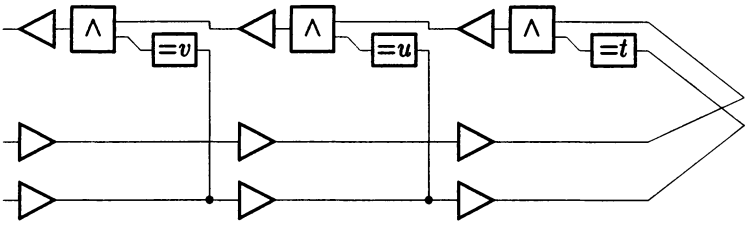


Figure 5 A systolic recognizer for the expression $t\,;u\,;v$

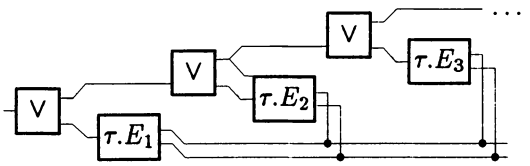
If the regular expression does not have this very special shape the benefit is diminished. Consider for instance expressions of the form

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

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:



The design presented here is well-optimized for expressions that contain many more ";" operators than "+". (This problem is also present in Foster and Kung's work; they simply fail to observe it.) The first author is currently investigating using a different transformation technique called "pipelining" to overcome this problem.

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.

REFERENCES

Aarts, C., Backhouse, R., Hoogendijk, P., Voermans, E. and van der Woude, J.: 1992, A relational theory of datatypes. Available at URL `ftp://ftp.win.tue.nl/pub/math.prog.construction/book.dvi`.

Backhouse, R. C.: 1983, Specification and proof of a regular language recognizer in synchronous CCS, *Technical Report CSM-53*, University of Essex.

Cohn, A. and Gordon, M.: 1990, A mechanized proof of correctness of a simple counter, *in* K. McEvoy and J. Tucker (eds), *Theoretical Foundations of VLSI Design*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press.

Doornbos, H., Backhouse, R. and van der Woude, J.: 1997, A calculational approach to mathematical induction. To appear in Theoretical Computer Science.

Foster, M. J.: 1984, *Specialized Silicon Compilers for Language Recognition*, PhD thesis, Computer Science Department, Carnegie Mellon University.

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

Hutton, G.: 1993, *Between Functions and Relations in Calculating Programs*, PhD thesis, Dept. of Computer Science, University of Glasgow.

Jones, G. and Sheeran, M.: 1990, Circuit design in Ruby, *in* J. Staunstrup (ed.), *Formal Methods for VLSI Design. IFIP WG 10.5 Lecture Notes*, North-Holland.

Jones, G. and Sheeran, M.: 1992, Deriving bit-serial circuits in Ruby, *in* A. Halaas and P. B. Denyer (eds), *IFIP Transactions A-1, VLSI 91*, North-Holland. Available at `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications/PRG-TR-3-91.ps.Z`.

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

Leiserson, C. E.: 1983, *Area-Efficient VLSI Computation*, MIT Press.

Leiserson, C. E. and Saxe, J. B.: 1991, Retiming synchronous circuitry, *Algorithmica* **6**, 5–35.

Rietman, F.: 1995, *A Relational Calculus for the Design of Distributed Algorithms*, PhD thesis, University of Utrecht.

Vaccari, M. and Backhouse, R.: 1996, Deriving a systolic regular language recognizer, *Technical report*, Eindhoven University of Technology. Available at `ftp://ftp.win.tue.nl/pub/math.prog.construction/regpaper.dvi.Z`.

7 BIOGRAPHIES

Matteo Vaccari is a Ph.D. student at the University of Milano. He has written statistics software for medical research, and was involved in the development of a model checker for CSP. He did work on the application of automated reasoning to hardware verification. His main research interests are in the practical application of circuit and program derivation techniques.

Roland Backhouse is Professor in Computing Science at Eindhoven University of Technology. His research area is the mathematics of program construction, in particular the calculational derivation of programs. Since 1990 his main interest has been the development of a generic theory of datatypes based on the point-free relational calculus.

Discussion Session
Monday morning

Lambert Meertens: The question is often asked after the presentation of a calculational technique: Did you find something new this way? Almost always the answer is: No, that was not our aim. I am starting to wonder why it is so rare to find something new using such techniques.

Roland Backhouse: In the case of the systolic regular language recognizer, something new was discovered, though not when the present design was done. The design that Foster and Kung presented was not correct for all regular expressions because of an indeterminacy in the functioning of the circuit for expressions of the form E^{**}. The current derivation makes it very clear where the error was. That is one example where one can actually discover something by algebraic calculation.

Dave Wile: Surely, the whole point of a calculus is to coalesce what is not new. Then you can go on and build something that couldn't have been considered before. For example, without the differential and integral calculus, one would never have considered something like boundary-value problems.

Jim Boyle: But then, why don't we see examples that correspond to boundary value problems?

Michel Sintzoff: A number of people do claim that they have found new algorithms using calculational methods. The problem is that these inventions seem not to be repeatable by others: calculational technology is not transferable. For example, I am still looking for people other than Doug Smith to use his algorithmic schemes to derive new algorithms.

Doug Smith: Eleven years or so ago, we recapitulated a lot of known backtrack algorithms and developed the Kids system to codify them. Five years ago we repeated the exercise for scheduling algorithms, which required elaboration of constraint propagation and backtrack augmentation techniques. Now we have derived dozens of schedulers for a variety of different problems. Are these new algorithms? In a sense they are not, because they can be regarded as a codification and abstraction of existing techniques using state-of-the-art technology. But, as a consequence, the technology has led to algorithms that people hadn't written down before. The real question is, at what point do such abstractions become economical? I think that mechanization is necessary before the economics show up. We are under contract to provide the next generation of schedulers for a couple of government sites. The agencies do not care whether or not the code has been synthesized by machines using a calculational approach, they are interested only in whether they are getting competitive software.

Peter Pepper: There is no clear definition of what 'new' means. For in-

stance, Mercedes, Volkswagen, and so on, build new cars all the time, but these manufacturers never re-invent the principles of cars and engines. Are they doing something new or just applying car-building principles? What we are doing, so to speak, is providing better engineering techniques for building the next car. I once read about a PhD student in Physics who spent four years parallelizing a single algorithm. Surely, we should be providing techniques to help prevent such situations arising?

Wolf Zimmermann: I disagree that we don't have a precise notion of what a new algorithm is. For me, a new algorithm is something that has a better time or space complexity, or a simpler implementation than any previous algorithm.

Tom Maibaum: There is a general hypothesis in Literature that there is something like seven basic stories or themes, and that all literary work is a reiteration and combination of these themes. Algorithmics is similar in that algorithms are generally quite basic, but their combination, programming in the large, is a much more difficult story that is more akin to literary efforts. How can one combine these basic algorithmic themes to get something new? We know next to nothing about how to do this systematically.

Bob Paige: I have used my transformational methodology to invent new algorithms. I cannot answer Sintzoff's observation that I am the only one who can do that – I am not sure that anyone else has tried. And it is not just Doug Smith and I who have been doing this sort of thing. Johan Jeuring has invented a new palindrome algorithm, though I don't know whether the new algorithm was only expressed using the Bird–Meertens formalism, or whether it actually helped the invention. Meertens has also invented some algorithms that he presented to this group at previous meetings. But the question is not that important, if you can demonstrate that a refinement calculus can help, then that is sufficient. Part of the reason that I try to invent new algorithms is that it makes my papers easier to publish. It is a defensive manoeuvre, because anyone trying to argue that my methodology is not viable has a harder task if I can invent five or six of the fastest algorithms around.

Michel Sintzoff: I think we should distinguish between new algorithms and new problems. New problems should be solved using known algorithmic schemes and the paradigms of reuse and scaling-up. To do this work we still have one challenge: to elaborate a common shared calculus. What we have now is as many calculi as researchers.

Jim Boyle: In some sense, the more we abstract the more we defeat the ability to find something new because people then say, oh yes, this is just an instance of that abstraction. The tragedy of Pepper's four year PhD example, I guess, is that none of the result was captured in a form that could be applied to another program.

Alberto Pettorossi: I think we should continue to seek out new languages,

new formalisms and new calculi, provided that they are adapted to different needs. There are many formalisms and calculi in mathematics. Calculi constantly improve; for instance, 20 years ago we used Hoare triples to prove correctness, now we use relational calculus. The problem for the future is to go from small-scale verification to large-scale verification, where we have problems of concurrency, distributivity, real-time constraints, and so on.

Peter Pepper: One of the big issues is that we encounter problems that are amalgamations of other problems, and the hard thing to do is to figure out what the subproblems are and how they are interrelated. This is where systems like Kids can help. The other issue is that of languages. There is a desperate need, not for more universal languages, but for application-oriented and domain-specific languages. We should apply our knowledge about calculi and languages to specific problem domains.

Tom Maibaum: I want to make a comment about inventing more and more calculi versus a universal language. In science and engineering one could say, a bit simplistically, that the ubiquitous tool is the differential equation. For me, the analogous tool in computing is not a theory within a single logic, but different logics or, what is the same thing, consequence relations. Such relations characterize the problem domain more accurately than a theory within some fixed universal logic. If you take the view that consequence relations are the equivalent of differential equations in science and engineering, then inventing new calculi is exactly what we should be doing to characterize different problem domains more completely.

Armando Haeberer: I agree with Sintzoff that for the same fundamental relational calculus we have seen about 50 versions, or something like that. I think we need to do some work on standardization.

Jim Boyle: I think we have not adequately conveyed outside our group the fact that our techniques can address the possibility of creating and implementing domain specific languages much more economically. We can write transformations to calculate the implementation of such languages quite easily, and that is something we should convey.