Università degli Studi di Milano
Dipartimento di Informatica
Dottorato di Ricerca in Informatica
Ph.D. Thesis

Calculational Derivation of Circuits

Matteo Vaccari

Advisors:
Prof. Roland Backhouse
Prof. Pierangelo Miglioli

Matteo Vaccari
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39, 20135 Milano–Italy
vaccari@dsi.unimi.it
http://eolo.usr.dsi.unimi.it/~matteo

Last Revision: January 19, 2015: recompiled with a modern distribution of TEX so that the resulting pdf is searchable

Abstract

This work is about the calculational construction of circuits. By "construction" we mean obtaining a design by means of formal manipulation of a formal specification. "Calculational" refers to the style which we employ, that is to try to reduce all program derivations to straightforward calculation.

In other words, in this thesis we aim to develop a style of calculus that aids the designer in the derivation (and the presentation) of circuits.

We start from an established notation called *Ruby*, developed by Sheeran and Jones, based on the algebra of binary relations. We then develop our own style of Ruby, through a number of derivation case studies.

The main results in this thesis concerns the derivation of regular language recognizing circuits. We show how an existing design, due to Foster and Kung can be derived and explained; we point out a limitation in Foster and Kung's design and show how it can be partially overcome.

Other case studies we present are a new derivation of a circuit that solves the carré problem, and a derivation of a round-robin scheduler.

We then show how our designs can be implemented in practice. First we show how to simulate the circuits using Hutton's Ruby simulator; then we go all the way to implement a compiler from Ruby circuits to a CSP-like notation called Tangram.

Additional tool support is provided by proof checkers. Even though our method is supposed to be mainly used with pencil and paper, it is possible and sometimes useful to check the proofs by machine. We use PVS to verify part of our theory and one of the case studies.

Sommario

Questo lavoro tratta della costruzione per mezzo di calcolo di circuiti. Per "costruzione" intendiamo ottenere un circuito per mezzo di manipolazioni formali di una specifica formale. "Per mezzo di calcolo" si riferisce allo stile che usiamo, che consiste nel cercare di ridurre tutte le derivazione di programmi a calcolo.

In altre parole, in questa tesi si aspira a sviluppare uno stile di calcolo che aiuti il progettista nella derivazione (e l'esposizione) di circuiti.

Il punto di partenza è un calcolo preesistente chiamato *Ruby*, sviluppato da Sheeran e Jones, basato sull'algebra delle relazioni binarie. Di qui sviluppiamo il nostro stile di Ruby, attraverso una serie di derivazioni.

I risultati principali in questa tesi riguardano la derivazione di circuiti che riconoscono linguaggi regolari. Mostriamo come una classe di circuiti progettati da Foster e Kung possano essere derivati e spiegati; sottolineiamo una limitazione di questi circuiti e mostriamo come si possa superarla, almeno in parte.

Altri casi sviluppati sono una nuova derivazione di un circuito che risolve il problema del carré, e un algoritmo di mutua esclusione.

Mostriamo poi come i nostri circuiti possano essere praticamente realizzati. Prima mostriamo come simularli per mezzo dell'interprete Ruby di Hutton; quindi produciamo un compilatore che traduce circuiti Ruby in una notazione simile a CSP chiamata Tangram.

I programmi che verificano dimostrazioni possono essere utili strumenti di supporto alla derivazione. Anche se il nostro metodo è pensato per essere usato con carta e matita, è possibile e a volte utile verificare le dimostrazioni meccanicamente. Usiamo il sistema PVS per verificate parte della nostra teoria, e di una derivazione.

Contents

Chapter 0

Introduction

> Very few tastes are universal. A taste for programs that can visibly be seen to be correct is certainly not one of them. A great deal of programmers are very bad at what they do. We *cannot* measure the usefulness of a technique by counting noses.
> Richard O'Keefe <5kph1v$74l$1@goanna.cs.rmit.edu.au>

This work is about the calculational construction of circuits. By "construction" we mean obtaining a design by means of formal manipulation of a formal specification. "Calculational" refers to the style which we employ, that is to try to reduce all program derivations to straightforward calculation.

In other words, in this thesis we aim to develop a style of calculus that aids the designer in the derivation (and the presentation) of circuits.

The ultimate goal of this research is to give the individual programmer, or circuit designer, a calculus that allows the derivation of programs and circuits that are better understood, better documented, and behave as expected. We stress *individual* in the above sentence, since the techniques presented here make sense for the individual even if they are not part of the design methodology embraced by the organization the individual works in.

We start from an established notation called *Ruby*, developed by Sheeran and Jones [25]. Ruby is a language based on (point-free) the algebra of binary relations. We then develop our own style of Ruby, through a number of derivation case studies. We find that a judicious mixture of pointwise and point-free reasoning is needed to attain a concise exposition.

The main results in this thesis concerns the derivation of regular language recognizing circuits. We show how an existing design, due to Foster and

Kung [18] can be derived and explained; we point out a limitation in Foster and Kung's design and show how it can be partially overcome.

Other case studies we present are a new derivation of a circuit that solves the carré problem, which is presented and solved by Rem in [45], and a derivation of a round-robin scheduler, from a specification found in a collection of benchmark circuits for verification by Kropf [29].

We then show how our designs can be implemented in practice. First we show how to simulate the circuits using Hutton's Ruby simulator [23]; then we go all the way to implement a compiler from Ruby circuits to a CSP-like notation called Tangram, developed by Van Berkel [51]. Finally we test our generated circuits with the Tangram simulator.

Additional tool support is provided by proof checkers. Even though our method is supposed to be mainly used with pencil and paper, it is possible and sometimes useful to check the proofs by machine. We use PVS [40] (a proof checker developed at SRI) to verify part of our theory and one of the case studies.

In the remainder of this chapter we have a short introduction to the calculational method and Ruby, followed by a brief summary of the thesis' contents.

0.0 The calculational method and Ruby

The calculational method strives to combine concision with precision. Ultimately we hope to be able to derive a program as a solution of a formal specification, much in the way the integral calculus is used in deriving a solution to an integral.

The relevance of mathematical logic to programming was made prominent by Hoare with the invention of (what is now called) Hoare logic. However, his work was aimed at *verifying* that a program satisfies a specification, rather than *deriving* a program that satisfies a given specification. An important step forward came with the calculus of weakest preconditions, by Dijkstra [11]; this enabled the programmer to derive a program from a goal assertion that must be established. To this day, this is the most effective way of deriving imperative programs operating on arrays.

A second step forward was the style of constructive functional programming introduced by Bird and Meertens [6], and known as the Bird/Meertens formalism. This allows one to derive functional programs operating on lists.

A third step forward was the generalisation of Bird/Meertens calculus to arbitrary (tree-like) datatypes, by Malcolm [36]. This opened the way to programs that are parameterised by type *constructors* (so-called polytypic programs) rather than simply parameterised by types.

The fourth step was the extension of the Bird/Meertens formalism to relations rather than simply functions. The application of (binary) relation algebra to computing is the subject of the work of a number of researchers worldwide [9]. Bird and De Moor derive dynamic programming algorithms in a relational style [7]; the Mathematics of Program Construction group in Eindhoven extended polytypic programming to relations [0]; and Sheeran and Jones invented Ruby, a relational language of circuits [25].

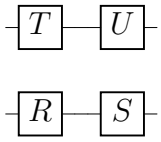
In this thesis we combine techniques from all of these developments, but the work on Ruby is closest to our goals.

In Ruby a circuit is a (binary) relation between streams, a stream being a function from time (i.e., the integers) to some value domain (e.g., the booleans). A simple example of Ruby circuit is the *and* gate that relates the stream of pairs $\{(a_t, b_t)\}_{t\in\mathbb{Z}}$ with the stream $\{a_t \wedge b_t\}_{t\in\mathbb{Z}}$.

Circuits can be composed with a number of combinators such as relational composition $(R \circ S)$ and relational product $(R \times S)$. The combinators enjoy a number of useful rules, like the so-called fusion rule:

$$(R \circ S) \times (T \circ U) \;=\; (R \times T) \circ (S \times U)$$

Ruby circuits have an interpretation as pictures, which is useful for the presentation of circuit design, as well as for the illustration of Ruby laws. For instance, both sides of the fusion rule have the same picture interpretation:



0.1 Foster and Kung's recognizers

In 1982, Foster and Kung presented a specialised silicon compiler (i.e., a program that produces the description of a circuit from a specification of the circuit's behaviour) that constructs recognizers for regular languages [18]. The compiler was presented without formal justification; indeed, they did not even present a formal specification of the functionality of the compiler.

Their informal description of the functioning left much room for alternative interpretations.

In this thesis we present a formal *derivation* of Foster and Kung's compiler. The complexity of the task is 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, which is essentially a functional program, followed by a transformation of this design into two different systolic versions, using standard techniques of "slowdown", "retiming" and "pipelining" [25].

The technique used in specifying the problem involves mapping the problem from streams to sets, in a way that is novel in Ruby literature. Part of this work was published in the proceedings of an international conference [50].

The goal of Foster and Kung's work was to obtain recognizers that are fully systolic; i.e., circuits such that no long propagation path exists. Their design fell short of this goal, since circuits corresponding to regular expressions where the "choice" operator occurs many times can have long propagation paths. We present an alternative to Foster and Kung's design that partially overcomes this problem, by using the standard technique of "pipelining".

0.2 Implementations

The designs we obtain are detailed enough to be implemented. We demonstrate this first by showing how to simulate our circuits by means of Hutton's Ruby interpreter [23]. Subsequently, we present a compiler from Ruby circuits to Tangram. Tangram is a CSP-like [21] imperative programming language developed by Van Berkel [51]. Tangram programs can be compiled to asynchronous circuits in silicon. We present the result of simulating our circuits using the Tangram simulator.

These implementations contain no new technical detail; the purpose is to demonstrate that the designs are in fact implementable, and to show how it would be possible to obtain working circuits in practice.

0.3 Machine checked proofs

The derivations we present are meant to be done with paper and pencil; in fact, care has been taken to keep calculations as short and as simple as possible. The goal of this research is to develop a viable calculus.

However, converting a paper-and-pencil proof to a format that can be proved by machine is a valuable learning experience. The proof checker forces one to be explicit about every minute detail, and uncovers every unstated assumption. The specification language afforded by the proof checker cannot be as flexible as paper-and-pencil mathematics, and this sometimes forces one to find alternative, more elementary definitions; which is also useful since it makes one think of a subject from different points of view.

The proof checker we use here is the Prototype Verification System (PVS), developed by Owre, Rushby, Shankar and others at Stanford Research Institute [40]. The main advantage of this program with respect to others such as Isabelle [42], HOL [10] or NQTHM [8] is the ease of use and the powerful proof language. What we have done is to formalise and prove a part of the Ruby theory in PVS, and then check part of the carré case study of chapter 5.

0.4 Outline

Chapter 1 gives an introduction to the mathematical style and notation we employ. Chapter 2 introduces the algebra of binary relations. Chapter 3 shows how binary relations can be used to reason about circuits, and introduces our variant of Ruby. Chapter 4 continues the exposition of Ruby by generalising pairs to n-tuples. Chapter 5 presents the first case study, the Carré problem. Chapter 6 contains the second case study, the round-robin scheduler. Chapter 7 is about the third case study, the regular language recognizers. Chapter 8 shows how to execute the recognizers with the Ruby interpreter. In chapter 9 a compiler from Ruby to Tangram is presented, and its use is demonstrated on the regular language recognizers. Chapter 10 contains an encoding of Ruby and the Carré case study in PVS. Finally, chapter 11 discusses the work presented in this thesis and provides some conclusions.

Acknowledgements

I must be ever so grateful to Roland Backhouse for accepting me as a "telecommuting" student, and to his family for accepting me in their house. I thank Giovanni Degli Antoni for accepting me in his laboratory, and organizing the seminar which got me involved in program derivation; thanks to Degli Antoni and Milano Ricerche for providing funding for travel and conferences. Thanks to Henk Doornbos and Paul Hoogendijk for help and

interesting discussion. Thanks to Oege de Moor for encouraging me. My thanks to Philips Electronics and Kees van Berkel for allowing me to use the Tangram tools; thanks to Hans van Gageldonk for instructing me in their use. Thanks to Graham Hutton for thorough proof-reading and many suggestions for improvement (although any remaining inaccuracies are solely the fault of the author). Thanks to Richard Verhoeven for proof-reading. Thanks to all the friends of the PhD lab in Milano, and especially Marco Benini for sharing my interests. This work wouldn't exist without the help and encouragement from my parents, Francesco and Rosanna. Finally, and most importantly, thanks to Paola and Anna for putting up with all this all this time.

TEXnical details

This thesis was written on Linux[0]-powered computers.

Most of the text and the calculations were typeset with Mathſpad[1], Richard Verhoeven and Roland Backhouse's mathematical editor that is particularly well suited for this kind of presentation.

The remaining text-editing work was done with Gnu Emacs[2].

The output of Mathſpad for this thesis consists of a number of LATEX files.

Circuit pictures were coded in `pic`, Brian W. Kernighan's language for typesetting pictures[3], and produced with the Gnu implementation of `pic`.

[0] `http://www.linux.org`

[1] `http://www.win.tue.nl/cs/wp/mathspad/`

[2] `http://www.gnu.org`

[3] See Jon Bentley, *More Programming Pearls* for an introduction; for the manual send an e-mail message to `netlib@research.att.com` with a body of 'send 116 from research/cstr'

Chapter 1

The calculational style

In this chapter we give an introduction to the notational conventions we use, along with an introduction to the calculational style, with some examples to show why we find it an effective style of reasoning and exposition.

The calculational style strives to reduce complex proofs to straightforward calculation. Quoting R. Backhouse [2]:

> Whereas Mathematics is principally concerned with amassing a collection of results, Computing Science is about method: *how to construct* programs and systems that are reliable, ergonomic and efficient. The *calculational method* is about enhancing the human being's innate abilities by reducing as much as possible of the construction process to elementary syntactic calculation.

A major problem in the derivation of hardware (and software as well) is in keeping the size of the derivations manageable, while retaining precision:

> The calculational method ... aims to combine precision with concision. [2]

Terms representing circuits can easily become very large. While proof-checking programs like HOL [10] or PVS [40] may help in managing the complexity, they offer little help in the way of understanding. In order for human beings to be able to understand derivations we must find ways to keep the size and complexity under control. Calculational methodists have found a number of ways to do so. Good introductions to the calculational method include [0, 1, 2, 12, 13, 53]. Some of the works that advocate the use of calculational techniques in programming are [4, 5, 7, 20, 37]. Graham et al. [19]

propose a more calculational approach to mathematics, although the techniques they use are more useful in the analysis rather than the derivation of algorithms.

In the next section the proof format that we use is explained; then an example proof in calculational style is shown. Finally, section 1.2 deals with rules for effective reasoning with fixed points.

1.0 Proof format

We will structure most proofs as a chain of equalities (or inequalities). The proof that $P \equiv Q$ because $P \equiv R$ and $R \equiv Q$ would be written as

$$\begin{array}{ll} & P \\ \equiv & \quad \{ \quad \text{hint why } P \equiv R \quad \} \\ & R \\ \equiv & \quad \{ \quad \text{hint why } R \equiv Q \quad \} \\ & Q \end{array}$$

(The symbol $\equiv$ stands for logical equivalence, which is just equality on booleans.) An important variation on this theme is a proof of the form $P \Rightarrow x = y$, that can be written as

$$\begin{array}{ll} & x \\ = & \quad \{ \quad \text{assuming } P \quad \} \\ & y \end{array}$$

This is often used when trying to derive a condition for $x = y$ to hold. This style of proof presentation is explained in detail in van Gasteren [53].

1.1 An example proof about *max*

As an example of what a calculational proof is, consider the following: we wish to prove that multiplication distributes through the maximum of two numbers: for any $a \geq 0$,

$$a{\cdot}x \uparrow a{\cdot}y = a{\cdot}(x \uparrow y), \tag{1.0}$$

where $r \uparrow s$ is the maximum of two numbers r and s. The usual definition of $\uparrow$ is

$$x \uparrow y = \begin{cases} x & \text{if } x \geq y \\ y & \text{if } y \geq x. \end{cases}$$

A problem with this is that it forces you to reason by cases whenever you want to prove *anything* that mentions $\uparrow$. Consider instead the following characterization, discussed by Feijen and Bijlsma in [16]:

$$x \uparrow y \leq z \;\equiv\; x \leq z \;\wedge\; y \leq z.$$

This equivalence is clearly valid, according to the old definition of $\uparrow$. It is possible to show that there is just one function that satisfies this equation; hence it may be taken as an alternative definition of $\uparrow$, one that is to be greatly preferred, because it avoids case analysis.

We want to prove (1.0). For $a = 0$ it clearly holds, so we only need to prove it for $a > 0$. We could do it by showing the two inclusions $a{\cdot}x \uparrow a{\cdot}y \leq a{\cdot}(x \uparrow y)$ and $a{\cdot}(x \uparrow y) \leq a{\cdot}x \uparrow a{\cdot}y$; but this would involve two separate arguments. We'd be eliminating a reasoning by cases, only to replace it by a two-part proof. The challenge is: can we produce a proof in one go?

There is a general law about partial orderings that can help us:

$$x{=}y \;\equiv\; \forall(z ::\; x \leq z \;\equiv\; y \leq z)$$

It is sometimes called the law of "indirect equality". Its usefulness is in making it easy to apply laws like the $\uparrow$ characterization. Returning to our problem, we calculate as follows: for any z,

$$\begin{array}{ll} & a{\cdot}x \uparrow a{\cdot}y \leq z \\ \equiv & \quad \{ \quad \text{characterization} \quad \} \\ & a{\cdot}x \leq z \;\wedge\; a{\cdot}y \leq z \\ \equiv & \quad \{ \quad a \text{ is positive} \quad \} \\ & x \leq z/a \;\wedge\; y \leq z/a \\ \equiv & \quad \{ \quad \text{characterization} \quad \} \\ & x \uparrow y \leq z/a \\ \equiv & \quad \{ \quad a \text{ is positive} \quad \} \\ & a{\cdot}(x \uparrow y) \leq z. \end{array}$$

The hint "a is positive" refers to the use of the equivalence

$$(a{\cdot}x \leq y \;\equiv\; x \leq y/a) \Leftarrow a > 0,$$

which is similar in shape to the $\uparrow$ characterization, but is much more widely known. Note that no real "reasoning" is required by the above proof; it is a simple calculation. With practice, it requires very little effort. The practical benefits are great. Proofs that required substantial effort are reduced to formal manipulation of symbols.

1.2 The fixed point calculus

The Eindhoven Mathematics of Program Construction group has collected a set of rules that make it easier to calculate with fixed points [39]. We give here a brief account of the fixpoint rules.

In this section, the letters f,g,h will stand for monotone functions over a complete lattice. Recall that a complete lattice is a lattice where meets and joins of arbitrary sets of elements exist. Let A be a complete lattice, and $\leq$ be the associated order relation. An element $a{\in}A$ is said to be a *prefix point* of function f iff $f.a \leq a$. We'll denote the least prefix point of f by μf. Hence the following *induction rule* holds:

$$\mu f \leq a \Leftarrow f.a \leq a \quad .$$

An element a is said to be a *fixed point* of f iff $f.a = a$. The main tool that is used in what follows is the Knaster-Tarski theorem:

Theorem 1.1 **(Knaster-Tarski)** if f is a monotone function over a complete lattice, then it has a least fixed point, that coincides with the least of its prefix points.

□

This theorem proves the following *computation rule*:

$$f.\mu f = \mu f \quad .$$

The rest of the fixed-point calculus rules can be derived from the first two.

Theorem 1.2 **(rolling rule)** $\mu(f \circ g) = f.\mu(g \circ f)$

□

Let the *pointwise ordering* between functions be defined by

$$f \sqsubseteq g \equiv \forall(x :: f.x \leq g.x) \ .$$

We then have the following:

Theorem 1.3 **(simple μ-fusion)** $\mu f \leq g.\mu h \Leftarrow f \circ g \sqsubseteq g \circ h$

□

Theorem 1.4 **(diagonal rule)** Let $\oplus$ be a binary operator that is monotone in both arguments. We then have that

$$\mu(a \mapsto a \oplus a) = \mu(a \mapsto \mu(b \mapsto a \oplus b)) \ .$$

□

The last rule below is proved in [39]:

Theorem 1.5 **(μ-fusion)** If f distributes over arbitrary joins, then we have:

$$f.\mu g \leq \mu h \Leftarrow f \circ g \sqsubseteq h \circ f \ .$$

□

Chapter 2

On the relation calculus

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].

Conducting calculations in a point-free style is advantageous, because many properties can be stated in a very concise form; for instance, the property "R is an injective relation" can be simply stated as $R^{\cup} \circ R \subseteq I$. However, not all calculations are possible or convenient to do in a point-free manner.

> As we have tried to demonstrate, relation algebra is an excellent vehicle for concise expression of fundamental notions in computing. It should not be supposed, however, that it will ever completely replace the pointwise predicate calculus. A degree of good taste is essential to deciding where and when point-free reasoning is to be preferred.
>
> Doornbos, van Gasteren, Backhouse [15]

2.0 Basic definitions and properties

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*. 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 program with deterministic behaviour. The reason for the choice of interpreting the right domain as input, is to follow the convention used in functional programming, where the term $f \bullet g$ is usually interpreted as a program that first applies g to the input, and then applies f to the result. Programming with relations is then an extension of functional programming.

If R is deterministic, then it may be considered as a function with domain on the right side and range 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. 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 over a given set $\mathcal{U}$ 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:

$$\begin{array}{rcl} R \circ I & = & I \circ R \;=\; R \\ (R \circ S) \circ T & = & R \circ (S \circ T). \end{array}$$

Furthermore, composition is monotonic in both arguments:

$$\begin{array}{rcl} R \circ S \subseteq R \circ T & \Leftarrow & S \subseteq T \\ R \circ S \subseteq T \circ S & \Leftarrow & R \subseteq T. \end{array}$$

Repeated composition of a relation R can be denoted by R^n:

$$\begin{array}{rcl} R^0 & = & I \\ R^{n+1} & = & R^n \circ R \quad \text{for } n \geq 0. \end{array}$$

The converse of a relation R is written $R^{\cup}$ (pronounced R *wok*) and is defined by

$$x\langle R^{\cup}\rangle y \;\equiv\; y\langle R\rangle x.$$

Converse satisfies the properties

$$\begin{array}{rcl} R^{\cup\cup} & = & R \\ (R \circ S)^{\cup} & = & S^{\cup} \circ R^{\cup}. \end{array}$$

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. The *right domain* of relation R, denoted by $R^{>}$, is defined in a similar way as the least monotype A such that $R \circ A = R$.

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 thesis.

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 thesis.

2.1 Pairs and product

From this point onwards we will assume that the set $\mathcal{U}$ is closed under pairing: given x and y chosen arbitrarily among the elements of $\mathcal{U}$, we assume that the pair (x, y) is also in $\mathcal{U}$.

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

It is common practice in mathematics 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

$$(2.0) \qquad (R \vartriangle S) \circ T \;=\; (R \circ T) \vartriangle (S \circ T) \quad\Leftarrow\quad S \circ T \circ T^{\cup} \;\subseteq\; S.$$

The antecedent holds, for example, when T is a deterministic relation (since then $T \circ T^{\cup} \subseteq I$). It also holds if S is a left condition.

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

$$x\langle \ll\rangle (y,z) \;\equiv\; x{=}y \qquad \text{and} \qquad x\langle \gg\rangle (y,z) \;\equiv\; x{=}z.$$

The following properties are easily proved:

$$(2.1) \qquad \begin{array}{lcl} R \times S \circ T \times U & = & (R \circ T) \times (S \circ U) \\ R \times S \circ T \vartriangle U & = & (R \circ T) \vartriangle (S \circ U). \end{array}$$

These laws are used most frequently in calculations. We call them *fusion* laws, because they allow to "fuse" two products into one (or a product and a split into a split). The fusion laws will allow us in chapter 3 to draw a picture of, say, $R \times S \circ T \times U$ without ambiguity.

To complete this brief survey of relation algebra we must introduce the reflexive, transitive closure of a relation (see e.g. [14]), which may be defined for relation R as a least fixed point of a certain function:

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

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

Table 2.0: Precedence of operators, from highest to lowest

In [14] it is proved that the "unique extension property" (uep) of the reflexive, transitive closure:

$$R \ = \ S^* \circ T \quad \equiv \quad R \ = \ T \cup S \circ R \tag{2.2}$$

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 2.0 for a complete list of precedences.

2.2 An example: bisimulations

The following example is to show the power and concision of the relational calculus.

Milner defines a theory of processes in his Calculus of Communicating Systems [38], where a process is, roughly, a graph with labelled arches. Such a graph can be modelled using a collection of relations, one relation for each label. Let's call A the set of labels; for every $a \ \in \ A$ we'll write $\stackrel{a}{\rightarrow}$ for the relation corresponding to the label a. Thus two vertices are connected by an a-labelled arch if and only if the relation $\stackrel{a}{\rightarrow}$ holds between those two vertices.

A *bisimulation* is a relation R such that, for all $a \ \in \ A$:

$$\begin{array}{lll} p\langle R\rangle q & \Rightarrow & \\ & (i) & \text{if } p\langle \stackrel{a}{\rightarrow}\rangle p', \text{ then } q\langle \stackrel{a}{\rightarrow}\rangle q' \text{ for some } q', \text{ and } p'\langle R\rangle q' \\ & (ii) & \text{if } q\langle \stackrel{a}{\rightarrow}\rangle q', \text{ then } p\langle \stackrel{a}{\rightarrow}\rangle p' \text{ for some } p', \text{ and } p'\langle R\rangle q' \ . \end{array} \tag{2.3}$$

This definition is bothersome to look at, not to mention to reason with. A much shorter definition of bisimilarity is the following:

$$(2.4) \qquad R \text{ is a simulation } \equiv \forall(a :\ a \in A :\ R^{\cup} \circ \overset{a}{\rightarrow} \subseteq \overset{a}{\rightarrow} \circ R^{\cup})$$

and R is a bisimulation if and only if both R and $R^{\cup}$ are simulations.

Aside How does one arrive at a definition such as (2.4)? Here is a derivation: first note that (2.3) equivales, for all p,q,p' and q',

$$(p\langle R\rangle q \Rightarrow (i)) \ \wedge \ (p\langle R\rangle q \Rightarrow (ii)).$$

Now we manipulate the first conjunct: for all $a \in A$, p, q and p',

$$\begin{array}{ll} & p\langle R\rangle q \Rightarrow (i) \\ \equiv & \quad \{ \quad \text{rewrite with existential quantifier} \quad \} \\ & p\langle R\rangle q \Rightarrow (p\langle \overset{a}{\rightarrow}\rangle p' \Rightarrow \exists(q' ::\ q\langle \overset{a}{\rightarrow}\rangle q' \ \wedge \ p'\langle R\rangle q')) \\ \equiv & \quad \{ \quad \text{propositional calculus} \quad \} \\ & p\langle R\rangle q \ \wedge \ p\langle \overset{a}{\rightarrow}\rangle p' \ \Rightarrow \ \exists(q' ::\ q\langle \overset{a}{\rightarrow}\rangle q' \ \wedge \ p'\langle R\rangle q') \\ \equiv & \quad \{ \quad \text{definitions of converse and composition} \quad \} \\ & q\langle R^{\cup} \circ \overset{a}{\rightarrow}\rangle p' \Rightarrow q\langle \overset{a}{\rightarrow} \circ R^{\cup}\rangle p' \\ \equiv & \quad \{ \quad \text{definition of inclusion} \quad \} \\ & q\langle R^{\cup} \circ \overset{a}{\rightarrow} \subseteq \overset{a}{\rightarrow} \circ R^{\cup}\rangle p' \end{array}$$

Hence we have proved

$$\forall(p, q, p' ::\ p\langle R\rangle q \Rightarrow (i)) \quad \equiv \quad R^{\cup} \circ \overset{a}{\rightarrow} \subseteq \overset{a}{\rightarrow} \circ R^{\cup}.$$

and similarly we can prove

$$\forall(p, q, q' ::\ p\langle R\rangle q \Rightarrow (ii)) \quad \equiv \quad R \circ \overset{a}{\rightarrow} \subseteq \overset{a}{\rightarrow} \circ R.$$

End aside.

Given this shorter definition, it becomes easy to prove things about bisimulations:

Fact: if R, S are bisimulations, then all of

$$\begin{array}{llll} (i) & I & (iii) & R \circ S \\ (ii) & R^{\cup} & (iv) & R \cup S \end{array}$$

are bisimulations too.

(i)

$$
\begin{array}{ll}
 & I^{\cup} \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ I^{\cup} \\
\equiv & \qquad \{\ I \text{ is symmetric } \} \\
 & I \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ I \\
\equiv & \qquad \{\ I \text{ is the unit of } \circ \ \} \\
 & \xrightarrow{a} \ \subseteq \ \xrightarrow{a}
\end{array}
$$

(ii) follows from $R^{\cup\cup} = R$

(iii)

$$
\begin{array}{ll}
 & (R \circ S)^{\cup} \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ (R \circ S)^{\cup} \\
\equiv & \qquad \{\ \cup \text{ distributes over } \circ \ \} \\
 & S^{\cup} \circ R^{\cup} \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ S^{\cup} \circ R^{\cup} \\
\Leftarrow & \qquad \{\ R \text{ is a bisimulation; } \circ \text{ is monotonic; transitivity } \} \\
 & S^{\cup} \circ \xrightarrow{a} \circ R^{\cup} \ \subseteq \ \xrightarrow{a} \circ S^{\cup} \circ R^{\cup} \\
\Leftarrow & \qquad \{\ \text{monotonicity of } \circ \ \} \\
 & S^{\cup} \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ S^{\cup} \\
\equiv & \qquad \{\ S \text{ is a bisimulation } \} \\
 & \mathit{true}
\end{array}
$$

(iv)

$$
\begin{array}{ll}
 & (R \cup S)^{\cup} \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ (R \cup S)^{\cup} \\
\equiv & \qquad \{\ \cup \text{ distributes over } \cup \ \} \\
 & (R^{\cup} \cup S^{\cup}) \circ \xrightarrow{a} \ \subseteq \ \xrightarrow{a} \circ (R^{\cup} \cup S^{\cup}) \\
\equiv & \qquad \{\ \circ \text{ distributes over } \cup \ \} \\
 & (R^{\cup} \circ \xrightarrow{a}) \cup (S^{\cup} \circ \xrightarrow{a}) \ \subseteq \ (\xrightarrow{a} \circ R^{\cup}) \cup (\xrightarrow{a} \circ S^{\cup}) \\
\Leftarrow & \qquad \{\ R, S \text{ are bisimulations; } \cup \text{ is monotonic; transitivity } \} \\
 & (\xrightarrow{a} \circ R^{\cup}) \cup (\xrightarrow{a} \circ S^{\cup}) \ \subseteq \ (\xrightarrow{a} \circ R^{\cup}) \cup (\xrightarrow{a} \circ S^{\cup}) \\
\equiv & \qquad \{\ X \subseteq X \ \} \\
 & \mathit{true}
\end{array}
$$

Chapter 3

About circuits

This chapters describes a calculus of circuits based on the previous chapter's calculus of relations. In particular we are going to specify that the universe $\mathcal{U}$ on which the relations are defined is a set of *streams*, that is, functions from the integers. We then define *lifted* relations, which model combinational circuits, and *delay* relations, which model memory elements, and other primitive circuits, as well as ways to build larger circuits by combining circuits together. A formal definition of what we mean by "circuit" is given. Finally, we give a number of rules for circuit transformation. Many properties of circuits are given; most of the proofs are to be found in the appendices.

The formalization of circuits we give in this chapter is along the lines of the Ruby language of Mary Sheeran and Geraint Jones. Good Ruby tutorials include [25, 24, 49]. Our version of Ruby uses a different syntax, and a different interpretation for product. The reason for our choosing a different syntax is to be "compatible" with previous work on relation algebra in Eindhoven, and to ease legibility. In fact, we feel that expressions like the Ruby

(3.0) $\quad [R; S, T; V]$

are much less legible than the alternative

$$(R \circ S) \times (T \circ V)$$

because "," is smaller than ";", so many people incorrectly read (3.0) as

$$[R; (S, T); V].$$

In standard Ruby the term $R \times S$ is interpreted as a relation between pair-valued streams (see below). In this thesis we interpret $R \times S$ as a relation between *pairs of streams*. A discussion of this choice is in chapter 11.

A further difference between our calculus and standard Ruby is that we generally think of right-to-left as the default direction of travel of data. This corresponds to the convention used in functional programming, where $f \bullet g$ is the program executed by first evaluating g, then f. In Ruby the opposite convention is used.

3.0 Circuits and lifted relations

Following established practice (see [10, 25, 46]) 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 (collections of) 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 some equalities no longer hold (for details see [24]). 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 appropriate values. 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. Other combining forms exist; a formal definition is in page 34.

3.1 Delays and stream identity

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, 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. (Operationally, one should take care to interpret the right hand side of a delay as the input, and the left hand side as the output; the opposite holds for the antidelay. See section 3.6 for further details on the operational interpretation of delays.) 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

$$\iota \;=\; \mu(X \;\mapsto\; \bar{\iota} \cup X\times X). \tag{3.1}$$

The delay relations apply the primitive delay ∂ to a collection of wires, independently of the shape of the collection: for $\diamond \in \{\lhd, \rhd\}$,

$$\diamond \circ \iota\times\iota \;=\; \diamond\times\diamond \;=\; \iota\times\iota \circ \diamond. \tag{3.2}$$

(see the proof in section A). 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 $\diamond \in \{\lhd, \rhd\}$,

$$\begin{array}{rcl} \diamond \circ R\times S & = & (\diamond \circ R)\times(\diamond \circ S) \\ R\times S \circ \diamond & = & (R \circ \diamond)\times(S \circ \diamond) \\ \diamond \circ R \vartriangle S & = & (\diamond \circ R) \vartriangle (\diamond \circ S). \end{array} \tag{3.3}$$

A domain property that we use frequently is

$$\lhd^{<} \;=\; \lhd^{>} \;=\; \iota \;=\; \rhd^{<} \;=\; \rhd^{>}. \tag{3.4}$$

From (2.0) and the fact that delays are deterministic, one obtains

$$R \vartriangle S \circ \diamond \;=\; (R \circ \diamond) \vartriangle (S \circ \diamond). \tag{3.5}$$

When reasoning pointwise, it is useful to remember the *delay introduction* rules: for all streams a and integers n,

$$a.n \;=\; (\lhd.a).(n+1) \qquad \text{and} \qquad a.(n+1) \;=\; (\rhd.a).n$$

Other important properties of delays are introduced in section 3.5.0.

3.2 Other circuit primitives

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

$$term \;=\; \iota \vartriangle \iota \circ \top\!\!\top.$$

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

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

We write K_x for a "constant circuit", actually a left condition, defined by

$$a\langle K_x\rangle b \;\equiv\; \forall(n ::\; a.n \;=\; x).$$

An obvious property of constant circuits is

$$K_x \circ R \;=\; K_x \circ R{>}$$

for all R.

Note that K_x is a lifted relation, with $K_x = \dot{f}$ and $f.y = x$ for all y.

The *feedback* of a circuit R, written R^σ, is defined by

$$a\langle R^\sigma\rangle b \;\equiv\; a\langle R\rangle(b,a); \tag{3.6}$$

for f a deterministic relation, the above can be written

$$a \;=\; f^\sigma.b \quad\equiv\quad a \;=\; f.(b,a)\;\;. \tag{3.7}$$

Loop and feedback are described, e.g., in Rietman [46, pages 23–25]. The feedback can be defined within the algebra, since the following

$$R^\sigma \;=\; (R{\cap}{\ll}) \circ \iota \vartriangle \top\!\top$$

is equivalent to (3.6). 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:

$$\begin{array}{lcll}
R^\sigma & = & (\iota \vartriangle \iota \circ R)^\varpi & \text{loop-feedback} \\
R \circ S^\varpi \circ T & = & (R{\times}\iota \circ S \circ T{\times}\iota)^\varpi & \text{loop fusion} \\
(\iota{\times}R \circ S)^\varpi & = & (S \circ \iota{\times}R)^\varpi & \text{loop leapfrog} \\
R^\sigma & = & (R{\cap}{\ll}) \circ \iota \vartriangle R^\sigma & \text{feedback unfolding} \\
(R{\times}S)^\varpi & = & R & \text{if } S \neq \perp\!\!\perp \\
R^\varpi \times S^\varpi & = & (zip \circ R{\times}S \circ zip)^\varpi & \\
R^\varpi \circ S^\varpi & = & (rsh \circ R{\times}\iota \circ zip \circ S{\times}\iota \circ lsh)^\varpi. &
\end{array} \tag{3.8}$$

For any two lifted relations $\dot{R}$ and $\dot{S}$, define $\dot{R}{\triangledown}\dot{S}$ (pronounced "junc") by

$$a\langle \dot{R}{\triangledown}\dot{S}\rangle(b,c) \;\equiv\; \forall(t:\; \neg(b.t):\; a.t\;\langle R\rangle\; c.t) \;\wedge\; \forall(t:\; b.t:\; a.t\;\langle S\rangle\; c.t);$$

junc can be defined by means of lifting of an "if-then-else" function. Junc is especially useful within a feedback loop. For instance, the following is a circuit that counts the number of consecutive "*true*" values it inputs:

$$(K_0{\triangledown}(\dot{+}1) \circ \iota{\times}{\lhd})^\sigma.$$

When the input is *true*, the output is one plus the previous output. When the input is *false*, the output is zero.

3.3 Summary of circuits

We may now define what a circuit is:

Definition 3.9 *circuit*

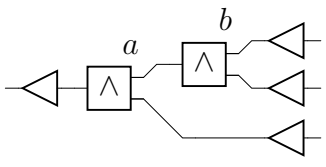
0. If R is a relation then $\dot{R}$ is a circuit.
1. The identity ι, the projections $\ll$ and $\gg$ and *term* are circuits.
2. If R,S are circuits, then $R \circ S$, $R \times S$, $R \vartriangle S$, R^{σ} and $R^{\cup}$ are circuits.
3. Delays and antidelays are circuits.

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 and antidelay 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 3.0 shows the correspondence between pictures and circuit terms. Note that the convention for drawing e.g., $R \times S$, is that R is drawn *below* S; the same convention holds for $R \vartriangle S$.

3.3.0 On combinational paths and systolic circuits

The picture interpretation of a circuit suggests that we may interpret a circuit as a graph, where delays and lifted relations are vertices, and everything else (the wires) is the edges. A *combinational path* is a path in the graph that does not contain delays. For instance, in the circuit in the figure below,



there is a combinational path from node a to node b. The presence of long combinational paths is usually bad from the point of view of the performance of a circuit. The longer a combinational path is, the longer it takes for the circuit to "settle down" after a change of state of the inputs [28].

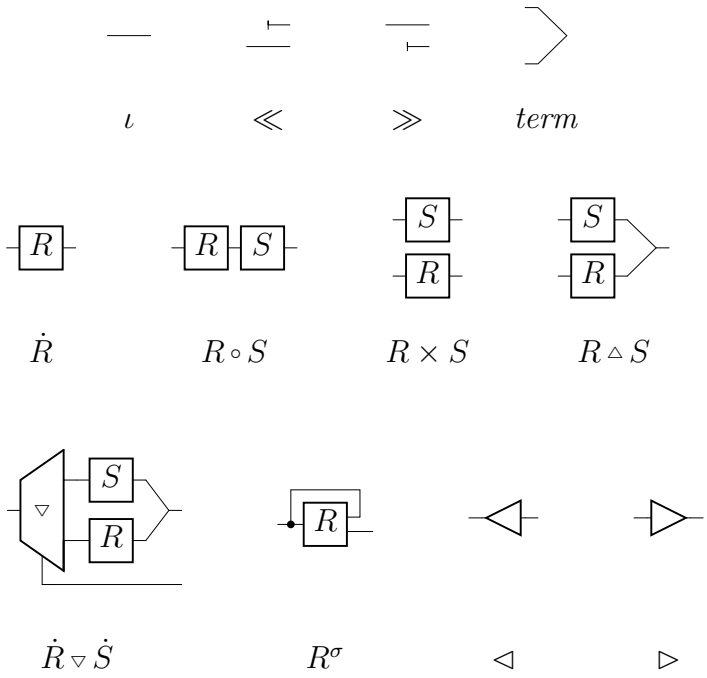


Figure 3.0: Circuits and their pictures

The operational interpretation that we give to our circuits is that of synchronous circuits. Lifted relations correspond to combinational circuits, and delays to memory elements. We suppose the existence of a global clock. At every tick of the clock, all delays change state, by reading their input and accepting it as their new state.

A combinational circuit, e.g. $\dot{\wedge}$, will not change its output immediately after a change in its input. A certain time is needed before the output becomes stable and correct; therefore after each change of state of the delays, a period of time ensues where the combinational elements compute their new output values. Only after all of the combinational elements have settled down on a value it is safe to execute the next state transition. Since all delays change state simultaneously, the clock period must be long enough to allow for the longest of all combinational paths to settle down. If one path is considerably longer than the others, this means that some parts of the system will settle down before others, and they perform no useful computation while they wait for the other parts to settle down.

For these reasons, it is preferable to keep all combinational paths as small as possible, so that no part of the circuit wastes time waiting for other parts to settle down.

A circuit is said to be *systolic* when there are no combinational paths; in other words, when all paths between combinational elements are interrupted by at least a delay [31].

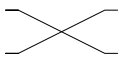
The operational interpretation of circuits is formalized in chapter 9, where the interpretation of a circuit as a graph is used, to translate a circuit into an imperative program.

3.4 Wiring relations

There is a class of relations that, rather than "performing an operation" on values, simply rearranges a structure of values into another structure. An example is the *swap* operation, that is defined as the smallest relation such that

$$(a, b)\langle swap\rangle(b, a)$$

Relations such as *swap* are usually pictured as a rearrangement of wires:

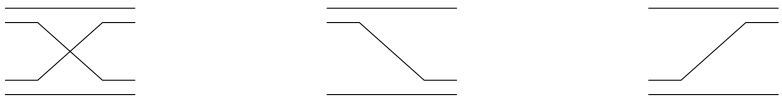


We call such relations "wiring" relations. Other examples of wiring relations are I, $\ll$ and $\gg$, and all relations built from wiring relations by composition, split and product.

3.4.0 Zips

It is common in functional programming to use a function called $\widetilde{zip}$, which is usually defined as

$$\begin{array}{lcl} \widetilde{zip}.([], []) & = & [] \\ \widetilde{zip}.(a : as, b : bs) & = & (a, b) : \widetilde{zip}.(as, bs) \end{array}$$



zip *rsh* *lsh*

Figure 3.1: Wiring relations

(here we write $a : as$ for the list that has head a and tail as, and [] for the empty list; more on this functional programming notation in chapter 9). Function $\widetilde{zip}$ transforms a pair of lists into a list of pairs:

$$\widetilde{zip}.([a,b,c],[d,e,f]) \;=\; [(a,d),(b,e),(c,f)]. \tag{3.10}$$

(Note that the definition of *zip* above holds for infinite as well as finite lists.)

We define a wiring relation that is the same as $\widetilde{zip}$ above, that relates a pair of pairs to a pair of pairs:

$$((a,b),(c,d))\langle zip\rangle((a,c),(b,d)). \tag{3.11}$$

Figure 3.1 shows a picture interpretation of *zip*. To relate *zip* with $\widetilde{zip}$, note that when applied to a pair of two-elements list $\widetilde{zip}$ satisfies:

$$\widetilde{zip}.([a,b],[c,d]) \;=\; [(a,c),(b,d)].$$

From the definition, a number of properties follow immediately:

$$\begin{array}{lcl} zip & = & zip^{\cup} \\ zip \circ zip & = & (I\times I)\times(I\times I) \\ zip \circ (R\times S)\times(T\times U) & = & (R\times T)\times(S\times U) \circ zip \\ zip \circ (R\times S) \vartriangle (T\times U) & = & (R \vartriangle T)\times(S \vartriangle U) \\ zip \circ (R \vartriangle S)\times(T \vartriangle U) & = & (R\times T) \vartriangle (S\times U). \end{array} \tag{3.12}$$

All of these equations are easily proved by pointwise reasoning. The proof of the third equation is:

$$
\begin{array}{ll}
 & ((a,b),(c,d))\langle zip \circ (R{\times}S){\times}(T{\times}U)\rangle((x,y),(z,w)) \\
\equiv & \quad \{ \quad \text{definition (3.11)} \quad \} \\
 & ((a,c),(b,d))\langle (R{\times}S){\times}(T{\times}U)\rangle((x,y),(z,w)) \\
\equiv & \quad \{ \quad \text{definition of product} \quad \} \\
 & a\langle R\rangle x \ \wedge \ c\langle S\rangle y \ \wedge \ b\langle T\rangle z \ \wedge \ d\langle U\rangle w \\
\equiv & \quad \{ \quad \text{definition of product} \quad \} \\
 & ((a,b),(c,d))\langle (R{\times}T){\times}(S{\times}U)\rangle((x,z),(y,w)) \\
\equiv & \quad \{ \quad \text{definition (3.11)} \quad \} \\
 & ((a,b),(c,d))\langle (R{\times}T){\times}(S{\times}U) \circ zip\rangle((x,y),(z,w)).
\end{array}
$$

The proof of the fourth equation is

$$
\begin{array}{ll}
 & ((a,b),(c,d))\langle zip \circ (R{\times}S) \vartriangle (T{\times}U)\rangle((x,y),(z,w)) \\
\equiv & \quad \{ \quad \text{definition (3.11)} \quad \} \\
 & ((a,c),(b,d))\langle (R{\times}S) \vartriangle (T{\times}U)\rangle((x,y),(z,w)) \\
\equiv & \quad \{ \quad \text{definition of split and product} \quad \} \\
 & a\langle R\rangle(x,y) \ \wedge \ c\langle S\rangle(z,w) \ \wedge \ b\langle T\rangle(x,y) \ \wedge \ d\langle U\rangle(z,w) \\
\equiv & \quad \{ \quad \text{definition of split and product} \quad \} \\
 & ((a,b),(c,d))\langle (R \vartriangle T){\times}(S \vartriangle U)\rangle((x,y),(z,w)),
\end{array}
$$

and the proof of the fifth equation is entirely similar to the last one:

$$
\begin{array}{ll}
 & ((a,b),(c,d))\langle zip \circ (R \vartriangle S){\times}(T \vartriangle U)\rangle((x,y),(z,w)) \\
\equiv & \quad \{ \quad \text{definition (3.11)} \quad \} \\
 & ((a,c),(b,d))\langle (R \vartriangle S){\times}(T \vartriangle U)\rangle((x,y),(z,w)) \\
\equiv & \quad \{ \quad \text{definition of split and product} \quad \} \\
 & a\langle R\rangle(x,y) \ \wedge \ c\langle S\rangle(x,y) \ \wedge \ b\langle T\rangle(z,w) \ \wedge \ d\langle U\rangle(z,w) \\
\equiv & \quad \{ \quad \text{definition of split and product} \quad \} \\
 & ((a,b),(c,d))\langle (R{\times}T) \vartriangle (S{\times}U)\rangle((x,y),(z,w)).
\end{array}
$$

3.4.1 Left and right shifts

Other useful wiring relations are *lsh* and *rsh*, defined by

$$((a,b),c)\langle lsh\rangle(a,(b,c)) \qquad \text{and} \qquad (a,(b,c))\langle rsh\rangle((a,b),c).$$

The names stand for "left shift" and "right shift". Clearly it holds

$$lsh \circ rsh = (\iota\times\iota)\times\iota \qquad \text{and} \qquad rsh \circ lsh = \iota\times(\iota\times\iota).$$

Figure 3.1 on page 37 shows picture interpretations of *lsh* and *rsh*.

3.5 Circuit transformations

In this section, we illustrate some techniques that can be used to improve the performance of circuits. The two most important techniques that are used to transform circuits are slowdown and retiming. These techniques were introduced in two papers by Leiserson and Saxe [32, 33]. Additional bibliography can be found in F.T. Leighton's book [30]. The presentation here is based on Jones [24].

3.5.0 Retiming

Retiming [33] is a transformation that is essentially based on the following laws: given that R is a circuit as defined above,

$$(3.13) \quad \triangleleft \circ R = R \circ \triangleleft \qquad \text{and} \qquad \triangleright \circ R = R \circ \triangleright.$$

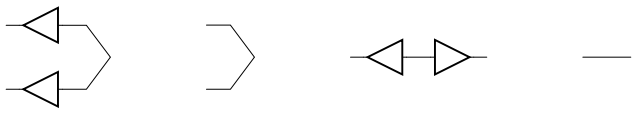
These laws can be proved by structural induction on definition 3.9; see section A. Combining (3.13) with the property that

$$(3.14) \quad \triangleright \circ \triangleleft = \iota = \triangleleft \circ \triangleright$$

we obtain the properties

$$(3.15) \quad R = \triangleright \circ R \circ \triangleleft \qquad \text{and} \qquad R = \triangleleft \circ R \circ \triangleright$$

for all circuits R. The use of the retiming law in circuit design is illustrated in example 3.19 on page 42.



$\lhd \circ term$ $\quad$ $term$ $\quad$ $\lhd \circ \rhd$ $\quad$ ι

Figure 3.2: Comparing instances of (3.16) and (3.15)

Note that the retiming law (3.15) 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 $\diamond \in \{\lhd, \rhd\}$,

(3.16) $\quad \diamond \circ term = term.$

In fact:

$$
\begin{array}{ll}
 & \diamond \circ term \\
= & \quad \{ \quad (3.2) \quad \} \\
 & \diamond \vartriangle \diamond \circ \top\!\!\top \\
= & \quad \{ \quad (3.5) \quad \} \\
 & \iota \vartriangle \iota \circ \diamond \circ \top\!\!\top \\
= & \quad \{ \quad \text{for all } R, \text{ it holds } R \circ \top\!\!\top = R^{<} \circ \top\!\!\top \quad \} \\
 & \iota \vartriangle \iota \circ \diamond^{<} \circ \top\!\!\top \\
= & \quad \{ \quad (3.4) \quad \} \\
 & term.
\end{array}
$$

An intuitive understanding of why this law is true can be gained by comparing the picture interpretation of (3.16), with the picture interpretation (3.15), for $R = \iota$ (see figure 3.2): the pictures are the same, modulo the orientation of the wires.

3.5.1 Slowdown

Another optimisation technique is slowdown [26]. 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

$$\mathcal{B} \;=\; \mu(X \;\mapsto\; \overline{\mathcal{B}} \;\cup\; X{\times}X \circ zip) \tag{3.17}$$

and $\overline{\mathcal{B}}$ is defined by

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

A bundle, when seen as a circuit with input on the right, is a device that repeatedly inputs a pair on the right, and outputs the elements of the pair one at a time on the left.

An immediate consequence of (3.17) is

$$\iota{\times}\iota \circ \mathcal{B} \;=\; \mathcal{B}{\times}\mathcal{B} \circ zip \;=\; \mathcal{B} \circ (\iota{\times}\iota){\times}(\iota{\times}\iota). \tag{3.18}$$

It can be shown by structural induction (see section B) that, for any circuit R, the circuit $slow.R$ is equal to the one 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. A slowed circuit can be seen as a circuit that performs two independent interleaved calculations, one on the odd clock ticks, the other on the even ticks.

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 [32].

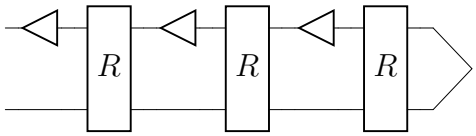
A slowed circuit generally takes more area than its non-slowed counterpart; this is the consequence of having twice as many delays. On the other hand, the slowed circuit can often be retimed in a way to make it faster. Furthermore, the slowed circuit performs an extra interleaved computation that can be exploited.

Example 3.19 *Retiming and Slowdown*

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

$$(3.20) \quad (\iota \times \triangleleft \circ R)^n \circ term$$

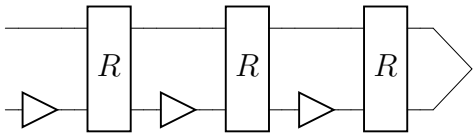
for some $n > 0$. The picture interpretation (for $n = 3$) shows a long combinational path:



By retiming, one obtains:

$$\begin{array}{ll} & (\iota \times \triangleleft \circ R)^n \circ term \\ = & \quad \{ \quad \text{retiming (3.15)} \quad \} \\ & (\triangleright \circ \iota \times \triangleleft \circ R \circ \triangleleft)^n \circ term \\ = & \quad \{ \quad \text{delays, (3.3) and (3.14)} \quad \} \\ & (\triangleright \times \iota \circ R \circ \triangleleft)^n \circ term \\ = & \quad \{ \quad \text{retiming (3.13)} \quad \} \\ & (\triangleright \times \iota \circ R)^n \circ \triangleleft^n \circ term \\ = & \quad \{ \quad \text{equation (3.16)} \quad \} \\ & (\triangleright \times \iota \circ R)^n \circ term. \end{array}$$

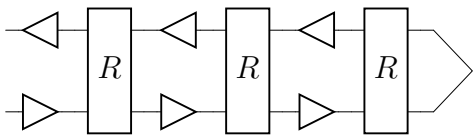
This transformation does not buy us anything, since the resulting circuit still has a long combinational delay:



But, if we choose to implement a slowed version of (3.20) instead, we have:

$$
\begin{array}{ll}
 & \mathit{slow}.((\iota\times\triangleleft \circ R)^n \circ \mathit{term}) \\
= & \quad \{ \quad \text{slowing is the same as doubling the delays;} \\
 & \qquad \text{assume } R \text{ is combinational} \quad \} \\
 & (\iota\times(\triangleleft \circ \triangleleft) \circ R)^n \circ \mathit{term} \\
= & \quad \{ \quad \text{fusion (2.1)} \quad \} \\
 & (\iota\times\triangleleft \circ \iota\times\triangleleft \circ R)^n \circ \mathit{term} \\
= & \quad \{ \quad \text{the above derivation,} \\
 & \qquad \text{taking } R \;:=\; \iota\times\triangleleft \circ R \quad \} \\
 & (\triangleright\times\iota \circ \iota\times\triangleleft \circ R)^n \circ \mathit{term} \\
= & \quad \{ \quad \text{fusion (2.1)} \quad \} \\
 & (\triangleright\times\triangleleft \circ R)^n \circ \mathit{term}.
\end{array}
$$

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".
□

Example 3.21 *Tangram and shift registers*

The purpose of this example is to show how the slowdown theorem allows one to reason about different implementations of buffers, and to introduce the Tangram language, which is described in more detail in chapter 9.

An n-places shift register is a device with an input stream a and an output stream b, such that for all t,

(3.22) $\quad b.t \;=\; a.(t-n).$

It is immediately obvious that $\triangleleft^n$ is a circuit that satisfies this specification.

In his Ph.D. thesis [51], Van Berkel discusses the pros and cons of two different designs for shift registers. Below is the Tangram code from a technical report by Van Berkel and Rem [52], for what they call a "ripple register", with $n = 2$.

$$
\begin{array}{ll}
 & (a?T \;\&\; b!T)\bullet \\
 & \textbf{begin} \\
(3.23) & \quad x0, x1 : \textbf{var}\ T \\
 & \mid\ \textbf{forever do}\ b!x1;\ x1 := x0;\ a?x0\ \textbf{do} \\
 & \textbf{end}
\end{array}
$$

The circuit described by this Tangram fragment is very similar to

$$\lhd^2.$$

It satisfies the same specification about the input-output histories of its ports a and b; and one could identify the two memory elements named $x0$ and $x1$ in the above program with the two unnamed delays in $\lhd^2$, and verify that the sequences of values held in $x0$ and $x1$ are equal to the sequences of values "held", respectively, in the rightmost and leftmost delay. In [52] it is remarked that in ripple registers values travel through all of the memory elements before being output, thus causing a relatively large energy dissipation. The cause of the problem is the command $x1 := x0$. An alternative design is then proposed, called a "wagging register". A two-places wagging register is described by the following Tangram fragment:

$$
\begin{array}{ll}
 & (a?T \;\&\; b!T)\bullet \\
 & \textbf{begin} \\
(3.24) & \quad x0, x1 : \textbf{var}\ T \\
 & \mid\ \textbf{forever do}\ b!x0;\ a?x0;\ b!x1;\ a?x1\ \textbf{do} \\
 & \textbf{end}
\end{array}
$$

The above program can be seen to satisfy (3.22); yet there is no internal copying of values, like in (3.23). The input values are written first in $x0$ and then in $x1$, in alternating fashion. The result is a circuit that dissipates less energy. The alternating behaviour of (3.24) suggests that we may obtain a similar design by means of bundles. Indeed, by the *slow* theorem, we know that $\lhd^2 \;=\; \mathcal{B} \circ \lhd\times\lhd \circ \mathcal{B}^{\cup}$. The right hand side of the last formula closely corresponds closely to (3.24).

This example shows how much more concise the relational notation is compared to CSP-like notations. It also shows that this fragment of "register theory" is a special case of the more general *slow* theorem.

□

3.5.2 Pipelining

A standard technique that can be used to make a circuit systolic is *pipelining*, which consists in breaking a computation in more stages, by placing delays, in order to shorten propagation paths. This technique is illustrated in the following example. Suppose we are to implement circuit

(3.25) R^n.

By the retiming law (3.15), we have that (3.25) is equal to

(3.26) $\triangleright^n \circ (\triangleleft \circ R)^n$.

If in (3.25) all information flows from right to left, then the term

(3.27) $(\triangleleft \circ R)^n$.

is implementable, but (3.26) is not, because of the $\triangleright^n$ term. The solution is to implement (3.27) only, and consider $\triangleright^n$ as an interface that documents the difference between (3.25) and (3.27). This interface can be interpreted to mean that the results will appear on the left side of circuit (3.27) exactly n clock ticks later than they do in the original circuit. The delay in the arrival of results is called the *latency time* of a pipelined circuit.

In summary, when we pipeline a circuit we trade clock period with latency time.

3.6 Realizability

We conclude this chapter 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 of directions 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. For instance, the following program,

$$\triangleright \circ add,$$

when assigned input on the right hand side and output on the left hand side, would produce a stream of numbers such that each one is the sum of the two numbers *that will be input one clock tick in the future*! Since digital technology is not (yet) able to predict the future, it is reasonable to rule out circuits such as this as non-implementable.

A more formal and thorough discussion of implementability is given in Hutton's thesis [23]. Notes on the implementability of bundle are in Sandum's report [47].

Chapter 4

Tuples and generalised products

This chapter contains introductory material that is only needed for chapters 5, 6 and 10.

In the previous chapter a calculus of circuits is introduced. The two main combining forms are composition and product. In this chapter we generalise product to n-wide products; this will allow us to describe circuits whose size and structure depends on integer parameters. Most of the space is dedicated to combining forms, that are n-wide generalisations of operators described in the previous chapter.

We generalise the relational product to n-wide products. A product of a single relation is the relation itself. A product of two relations is the usual relational product. The product of n relations R_0 through R_{n-1} is $R_0 \times (R_1 \times (\ldots \times R_{n-1}))$. By adopting the convention that $\times$ associates to the right, we can write the above product as simply $R_0 \times R_1 \times \ldots \times R_{n-1}$. Corresponding to n-wide products we have n-tuples. For instance, a 3-tuple has the shape $(a, (b, c))$ for some a, b, c. By adopting the convention that pairing associates to the right, we write the above tuple as simply (a, b, c).

Our convention for representing tuples is ambiguous: if we substitute $c := (d, e)$ in (a, b, c) it turns out that the "3-tuple" is also a 4-tuple. This ambiguity is not a problem for our purposes. When we write (a, b, c), all we know is that it represents a tuple of *at least* three elements. The same ambiguity happens with n-wide products: the product $R \times S \times T$ is 3-wide, but if we substitute $T := U \times V$ in it we get a 4-wide product.

We now define a notion of left and right *arity* for circuits. First we introduce

the (n) family of monotypes:

$$(4.0)\qquad \begin{array}{lcl} (1) & = & \iota \\ (n+1) & = & \iota\times(n) \quad \text{for } n \geq 1. \end{array}$$

For any expression E in the positive integers, we can assign a corresponding monotype (E) by means of definition (4.0). When the expression is just a numeral or a single letter, we can drop parentheses, and write n in place of (n).

We say that R has *right arity* n iff

$$R \;=\; R \circ n.$$

similarly, we say that R has *left arity* n whenever

$$R \;=\; n \circ R.$$

We take the expression $R \in n \sim m$ to mean that R has left arity n and right arity m:

$$R \in n \sim m \;\equiv\; R = n \circ R \;\wedge\; R = R \circ m.$$

So for instance we can say that pointwise disjunction, $\dot\vee$, has arity $1 \sim 2$.

The following is a simple consequence of (4.0) and (3.1):

$$(n) \subseteq (m) \Leftarrow n > m.$$

4.0 Maps

We now define new operations. The well-known *map* operation generates the product of n copies of a circuit:

$$\begin{array}{lcl} map_1.R & = & R \\ map_{n+1}.R & = & R \times map_n.R \quad \text{for } n \geq 1 \end{array}$$

For every R, we have $map_n.R \in n \sim n$. Corresponding to the fusion law (2.1) we have the *map fusion* law:

$$(4.1)\qquad map_n.(R \circ S) \;=\; map_n.R \circ map_n.S.$$

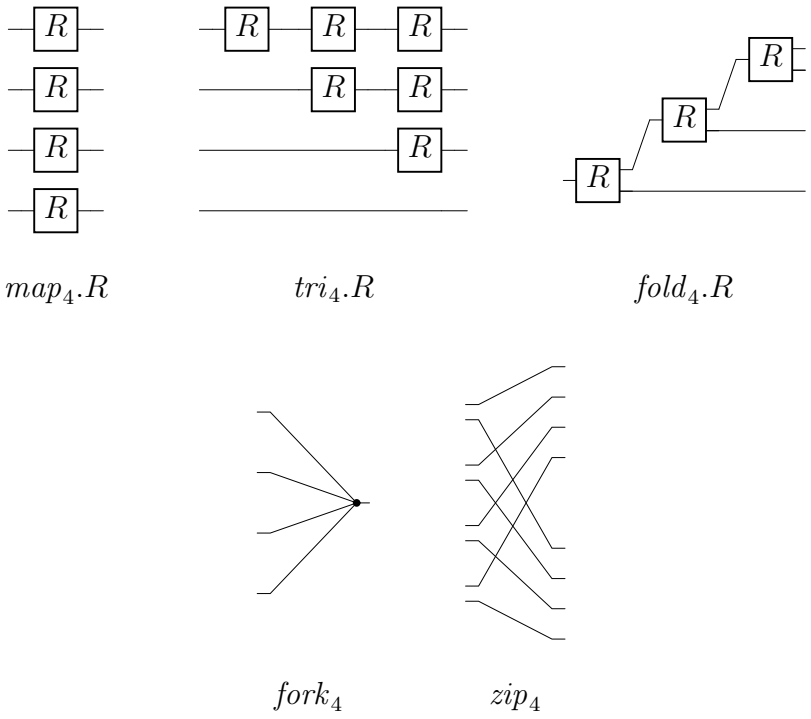


Figure 4.0: Instances of tuple operations

Suppose F is a function from integers to circuits. The parallel composition

$$F.0 \times F.1 \times \ldots \times F.(n-1)$$

can be represented by extending the *map* notation in the following way:[0]

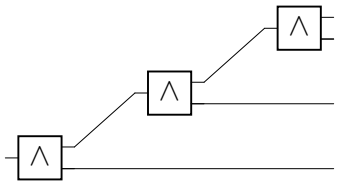
$$map.(i:\ 0 \le i < n:\ F.i).$$

In the remainder of the thesis, the subscripts will be dropped from *map* and similar operations where they are easily inferred from the context.

[0]This extension of the *map* operator is the only example of *heterogeneous combinator* that we need in this thesis. Such combinators were introduced by Wayne Luk in [34]. Luk's notation for $map.(i:\ 0 \le i < n:\ F.i)$ is $\|_{0 \le i < n} F_i$.

4.1 Folds

Suppose that a circuit R has arity $1 \sim 2$. Typical examples of this kind of circuit are lifted binary operations like $\dot{\wedge}$. To "apply" $\dot{\wedge}$ to a collection of wires, one would use a circuit of the shape



The *fold* operation generalises the above example[1]. We define *fold* by

$$\begin{array}{lcl} \mathit{fold}_1.R & = & \iota \\ \mathit{fold}_{n+1}.R & = & R \circ \iota \times \mathit{fold}_n.R \quad \text{for } n \geq 1 \end{array}$$

From the definition, we have

$$\mathit{fold}_n.R \in 1 \sim n.$$

A law about *map* and *fold* is the following: given R and S such that

$$R \circ S{\times}S \;=\; S \circ R$$

then

$$\mathit{fold}_n.R \circ \mathit{map}_n.S \;=\; S \circ \mathit{fold}_n.R. \tag{4.2}$$

The proof is by induction on n; for $n = 1$ it is trivially true. For $n + 1$ we have

$$\begin{array}{ll} & \mathit{fold}_{n+1}.R \circ \mathit{map}_{n+1}.S \\ = & \quad \{ \quad \text{definitions} \quad \} \\ & R \circ \iota \times \mathit{fold}_n.R \circ S \times \mathit{map}_n.S \\ = & \quad \{ \quad \text{fusion, induction hypothesis} \quad \} \\ & R \circ S{\times}(S \circ \mathit{fold}_n.R) \\ = & \quad \{ \quad \text{proviso: } R \circ S{\times}S \;=\; S \circ R\text{; fusion} \quad \} \\ & S \circ R \circ \iota \times \mathit{fold}_n.R \\ = & \quad \{ \quad \text{definition} \quad \} \\ & S \circ \mathit{fold}_{n+1}.R. \end{array}$$

[1]The Ruby literature defines a similar construction called *rdr*.

The proviso $R \circ S \times S \;=\; S \circ R$ is a distributivity requirement. For instance, if we take ordinary addition for R, and we take $(*x)$ for S, i.e. S is the relation that multiplies the input by x, the proviso equivales the distributivity law for numbers, $\forall(y, z ::\; x*y + x*z \;=\; x*(y+z))$. The proviso is also satisfied by choosing $S \;=\; \lhd$, for any circuit R.

4.2 Forks

A generalisation of $\iota \vartriangle \iota$ is *fork*, defined by

$$\begin{array}{lcl} fork_1 & = & \iota \\ fork_{n+1} & = & \iota \vartriangle fork_n \quad \text{for } n \geq 1 \end{array}$$

The arity of *fork* is

$$fork_n \;\in\; n \;\sim\; 1.$$

Clearly it holds that

$$(a_1, a_2, \ldots, a_n)\langle fork_n \rangle b \;\equiv\; \forall(i :\; 1 \leq i \leq n :\; a_i \;=\; b).$$

The property analogous of (2.0) is, for f a deterministic relation,

$$fork_n \circ f \;\equiv\; map_n.f \circ fork_n. \tag{4.3}$$

4.3 Triangles

Another useful construct is *tri* (triangle), defined by

$$\begin{array}{lcl} tri_1.R & = & \iota \\ tri_{n+1}.R & = & \iota \times (tri_n.R \circ map_n.R) \quad \text{for } n \geq 1 \end{array}$$

The arity of *tri* is:

$$tri_n.R \;\in\; n \;\sim\; n.$$

An alternative definition of *tri* is:

$$tri_n.R \;=\; map.(i :\; 0 \leq i < n :\; R^i). \tag{4.4}$$

There is a so-called *Horner's rule* [25] that is useful in reasoning with *tri* and *fold*: if it is the case that $R \circ S{\times}S \;=\; S \circ R$, then, for all $n \geq 2$

$$(4.5) \qquad \mathit{fold}_n.R \circ \mathit{tri}_n.S \;=\; \mathit{fold}_n.(R \circ \iota{\times}S).$$

This rule is useful because the right hand side suggests an implementation that requires $O(n)$ copies of S, whereas the left hand side requires $O(n^2)$. If we take ordinary addition for R, and we take $(*x)$ for S, the rule corresponds to the well-known Horner's rule for evaluating polynomials:

$$a_0 \;+\; a_1{*}x \;+\; a_2{*}x^2 \;+\; \cdots \;+\; a_n{*}x^n \;=\; a_0 \;+\; x{*}(a_1 \;+\; x{*}(a_2 \;+\; \cdots \;+\; x{*}a_n)).$$

The proof of Horner's rule is by induction on n. For $n \;=\; 2$ both sides of (4.5) reduce to $R \circ \iota{\times}S$. For $n \geq 2$,

$$\begin{array}{ll}
 & \mathit{fold}_{n+1}.R \circ \mathit{tri}_{n+1}.S \\
= & \qquad \{ \quad \text{definitions} \quad \} \\
 & R \;\circ\; \iota \times \mathit{fold}_n.R \;\circ\; \iota{\times}(\mathit{tri}_n.S \circ \mathit{map}_n.S) \\
= & \qquad \{ \quad \text{fusion} \quad \} \\
 & R \circ \iota{\times}(\mathit{fold}_n.R \circ \mathit{tri}_n.S \circ \mathit{map}_n.S) \\
= & \qquad \{ \quad \mathit{tri}.X \text{ and } \mathit{map}.X \text{ commute; the proof is simple} \quad \} \\
 & R \circ \iota{\times}(\mathit{fold}_n.R \circ \mathit{map}_n.S \circ \mathit{tri}_n.S) \\
= & \qquad \{ \quad \text{by the proviso, we can use } (4.2) \quad \} \\
 & R \circ \iota{\times}(S \circ \mathit{fold}_n.R \circ \mathit{tri}_n.S) \\
= & \qquad \{ \quad \text{induction hypothesis} \quad \} \\
 & R \circ \iota{\times}(S \circ \mathit{fold}_n.(R \circ \iota{\times}S)) \\
= & \qquad \{ \quad \text{fusion} \quad \} \\
 & R \;\circ\; \iota{\times}S \;\circ\; \iota \times \mathit{fold}_n.(R \circ \iota{\times}S) \\
= & \qquad \{ \quad \text{definition} \quad \} \\
 & \mathit{fold}_{n+1}.(R \circ \iota{\times}S).
\end{array}$$

This concludes the proof of Horner's rule.

4.4 Zips

The generalised version of *zip* is as follows:

$$\begin{array}{lcl}
\mathit{zip}_1.(a,b) & = & (a,b) \\
\mathit{zip}_{n+1}.((a,b),(c,d)) & = & ((a,c),\mathit{zip}_n.(b,d)) \quad .
\end{array}$$

The above is an acceptable relational definition, since a function is also a relation. Informally, zip_n transforms a pair of n-tuples into an n-tuple of pairs. The arity of zip_n is thus

$$zip_n \ \in \ n \sim n{\times}n.$$

For $n = 2$ the above definition simplifies to definition (3.11). A property of *zip*, corresponding to one of (3.12), is

$$zip_n \circ map_n.R \times map_n.S \ = \ map_n.(R{\times}S) \circ zip_n, \tag{4.6}$$

and a similar law holds about *zip* and *tri*:

$$zip_n \circ tri_n.R \times tri_n.S \ = \ tri_n.(R{\times}S) \circ zip_n. \tag{4.7}$$

Two other laws about zip_n are

$$zip_n \circ map_n.R \vartriangle map_n.S \ = \ map_n.(R \vartriangle S) \tag{4.8}$$

and

$$zip_n \circ fork_n \vartriangle fork_n \ = \ map_n.(\iota \vartriangle \iota) \circ fork_n. \tag{4.9}$$

There is a law that links zip_n and zip_2: for $n \geq 2$,

$$zip_n \ = \ (fold_n.zip_2)^{\cup}; \tag{4.10}$$

the proof is by induction:

$$\begin{array}{ll} & zip_2 \ = \ (fold_2.zip_2)^{\cup} \\ \equiv & \quad \{ \quad fold_2 \ = \ \iota{\times}\iota \quad \} \\ & zip_2 \ = \ {zip_2}^{\cup} \\ \equiv & \quad \{ \quad (3.12) \quad \} \\ & true \end{array}$$

and

$$
\begin{array}{ll}
 & zip_{n+1} \;=\; (fold_{n+1}.zip_2)^{\cup} \\
\equiv & \quad \{ \quad \text{definition of } fold \quad \} \\
 & zip_{n+1} \;=\; (zip_2 \;\circ\; \iota \times fold_n.zip_2)^{\cup} \\
\equiv & \quad \{ \quad \text{inverse over composition} \quad \} \\
 & zip_{n+1} \;=\; \iota \times fold_n.zip_2 \;\circ\; zip_2 \\
\equiv & \quad \{ \quad \text{by induction} \quad \} \\
 & zip_{n+1} \;=\; \iota \times zip_n \;\circ\; zip_2.
\end{array}
$$

We continue pointwise. We can use function application notation instead of $\langle - \rangle$ notation because *zip* is deterministic; and since *zip* is only defined on pairs of pairs, we can write:

$$
\begin{array}{ll}
 & (\iota \times zip_n \;\circ\; zip_2).((a, as), (b, bs)) \\
= & \quad \{ \quad \text{application of } zip_2 \quad \} \\
 & (\iota \times zip_n).((a, b), (as, bs)) \\
= & \quad \{ \quad \text{definition of product} \quad \} \\
 & ((a, b), zip_n.(as, bs)) \\
= & \quad \{ \quad \text{definition of } zip \quad \} \\
 & zip_{n+1}.((a, as), (b, bs)).
\end{array}
$$

This concludes the proof of (4.10).

4.5 Bundles

Bundle can also be generalised. Define, for $n \geq 1$,

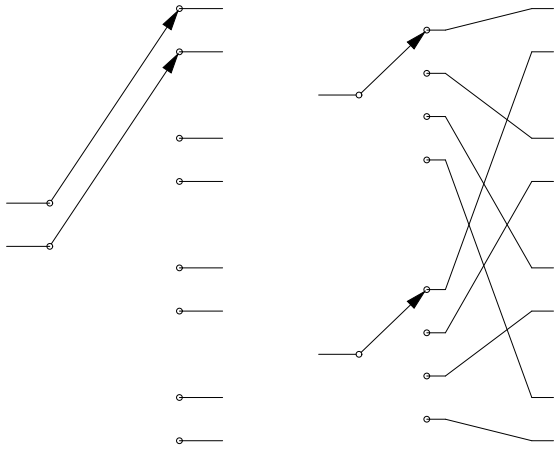
$$a\langle \overline{\mathcal{B}}_n \rangle (b_0, b_1, \ldots, b_{n-1}) \;\equiv\; \forall (t, k :\; 0 \leq k < n :\; a.(t{*}n \;+\; k) \;=\; b_k.t)$$

and

$$(4.11) \quad \mathcal{B}_n \;=\; \mu(X \;\mapsto\; \overline{\mathcal{B}}_n \cup X \times X \;\circ\; zip_n{}^{\cup}).$$

The last equation is a generalisation of (3.17), because $\overline{\mathcal{B}}_2 = \overline{\mathcal{B}}$ and $zip_2 = zip = zip^{\cup}$. The arity of bundle is then:

$$\mathcal{B}_n \;\in\; 1 \;\sim\; n.$$



$\iota \times \iota \circ \mathcal{B}_4$ $\mathcal{B}_4 \times \mathcal{B}_4 \circ zip_4^{\cup}$

Figure 4.1: An instance of (4.12)

From (4.11) it is immediate to obtain

(4.12) $\iota \times \iota \circ \mathcal{B}_n \;=\; \mathcal{B}_n \times \mathcal{B}_n \circ zip_n{}^{\cup} \;=\; \mathcal{B}_n \circ map_n.(\iota \times \iota).$

that is a generalisation of (3.18). A more general result is

(4.13) $m \circ \mathcal{B}_n \;=\; \mathcal{B}_n \circ map_m.n \;=\; map_m.\mathcal{B}_n \circ trn_{n,m},$

where $trn_{n,m}$ is the component that transposes n m-tuples into m n-tuples:

$$((a_{1,1},\ldots,a_{1,m}),\ldots,(a_{n,1},\ldots,a_{n,m}))\ \langle trn_{n,m}\rangle\ ((a_{1,1},\ldots,a_{n,1}),\ldots,(a_{m,1},\ldots,a_{m,n}))\ .$$

In other words, *trn* is the transpose operation that takes a n-tuple of m-tuples, interpreted as a $n \times m$ matrix, into the transposed $m \times n$ matrix. This component may be defined by

$$trn_{n,m} \;=\; fold_n.zip_m.$$

Some properties of *trn* are

$$trn_{2,m} \;=\; zip_m$$

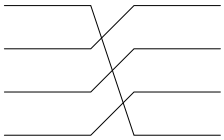


Figure 4.2: rot_4

and

$$trn_{n,2} \;=\; zip_n{}^{\cup}.$$

Another property of bundle is

$$\dot{R} \circ \mathcal{B}_n \;=\; \mathcal{B}_n \circ map_n.\dot{R}. \tag{4.14}$$

4.6 Rotations

A useful wiring relation is *rot*, defined by

$$rot_n \;=\; append_n \circ \gg^{\triangle} \ll,$$

where $append_n$ transforms a pair of n-length tuples in a single $2n$-length tuple:

$$\begin{array}{rcl} (a,b)\langle append_1\rangle(a,b) & & \\ (a,d)\langle append_{n+1}\rangle((a,b),c) & \equiv & d\langle append_n\rangle(b,c) \quad \text{for } n \geq 1 \end{array}$$

Informally, *rot* "rotates" the wires one position to the left (you are supposed to think of input being on the right side):

$$(a_2, a_3, \ldots, a_n, a_1)\langle rot\rangle(a_1, a_2, a_3, \ldots, a_n).$$

The *append* in the definition of *rot* is needed to restore the proper structure of the wires; for instance,

$$(\gg^{\triangle} \ll).(a,b,c,d) \;=\; (\gg^{\triangle} \ll).(a,(b,c,d)) \;=\; ((b,c,d),a)$$

and

$$append.((b,c,d),a) \;=\; (b,(c,d,a)) \;=\; (b,c,d,a).$$

Clearly, *rot* must satisfy

$$rot_n^{\,n} \;=\; n,$$

and

$$rot_n{}^{\cup} \circ rot_n \;=\; n,$$

and

$$rot_n{}^{\cup} \;=\; rot_n^{\,n-1}.$$

4.7 Cyclic multiplexers

Wayne Luk introduced in [35] the $cmx_{i,N}$ component (from "cyclic multiplexer"), defined by

$$cmx_{i,n} \;=\; \mathcal{B}_n \circ rot^{\cup i} \circ \ll \times map_{n-1}.\gg \circ rot^i \circ \mathcal{B}_n{}^{\cup}$$

(the definition presented here is more general; Luk's definition only considers the case $i = 0$). The cyclic multiplexer can be interpreted as a device that has two inputs (on the right) and one output (on the left). At each time t, if $t \bmod n = i$ then $cmx_{i,n}$ behaves like $\gg$; otherwise, it behaves like $\ll$. In other words, the left input "passes through" once every n clock ticks. Formally, $cmx_{i,n}$ satisfies

$$a\langle cmx_{i,n}\rangle(b,c) \;\equiv\; \forall(t:\; t \bmod n = i:\; a.t = b.t) \;\wedge\; \forall(t:\; t \bmod n \neq i:\; a.t = c.t). \tag{4.15}$$

A useful property of $cmx_{i,n}$ is

$$cmx_{i,n} \circ \dot{R} \vartriangle \dot{S} \;=\; \mathcal{B}_n \circ rot^{\cup i} \circ \dot{R} \times map_{n-1}.\dot{S} \circ rot^i \circ \mathcal{B}_n{}^{\cup}. \tag{4.16}$$

The proof is easily obtained by pointwise reasoning.

A cyclic multiplexer is easily implemented in practice using a counter and a multiplexer; see for instance Katz [28].

Chapter 5

The carré problem

In this chapter we show the derivation of an efficient circuit, from a specification which is implementable, but inefficient in terms of area.

The carré problem is described, and solved, in a short note by Rem [45]. A carré is a sequence such that the first half is equal to the second half. We want to derive a circuit that, given an input stream, flags whether the last $2N$ symbols read were a carré, for fixed N.

Rem solves the problem by deriving an imperative program, that can be interpreted as a circuit, by means of pointwise predicate calculus. Here we solve the problem in a different way: we will obtain a relation algebra term by mostly point-free relation algebra calculations. This term can be interpreted as a circuit as well. The circuit interpretation of our result is similar to the circuit presented by Rem, and is just as efficient in terms of area. However, our calculations are

We want to construct a circuit C that, given an input sequence, outputs *true* whenever the last $2N$ symbols read were a carré, and outputs *false* otherwise. Formally,

$$a\langle C\rangle b \;\equiv\; \forall(n ::\; a.n \;\equiv\; b(n-2N, n-N] \;=\; b(n-N, n]),$$

where $c(l,k]$ denotes the sequence $[c.(l+1), c.(l+2), \ldots, c.k]$. A straightforward way to translate the above pointwise specification into a point-free one is as follows (see picture 5.0):

$$(5.0) \qquad \mathit{fold}.\dot{\wedge} \circ \mathit{map}.\doteq \circ \mathit{zip} \circ (\mathit{tri}.\lhd \circ \mathit{fork}_N)\times(\mathit{tri}.\lhd \circ \mathit{fork}_N) \circ \iota \vartriangle \lhd^N$$

An informal operational interpretation of (5.0) is: reading from right to left, $\iota \vartriangle \lhd^N$ creates two copies of the input stream, and delays one of them N

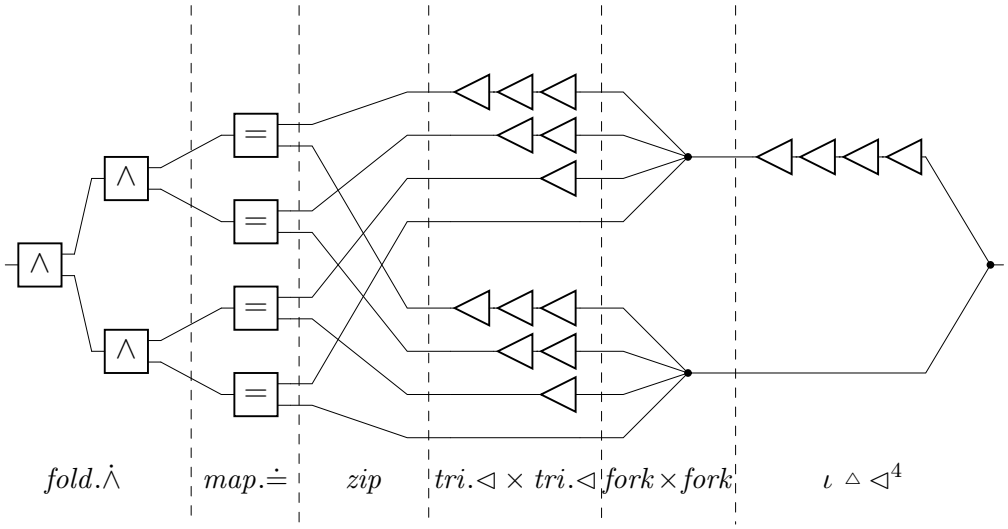


Figure 5.0: An instance of (5.0)

times. By applying $(tri.\triangleleft \circ fork_N)$ to a stream c one obtains the N-tuple $(c, \triangleleft.c, \triangleleft^2.c, \ldots, \triangleleft^{N-1}.c)$; hence the left domain of $(tri.\triangleleft \circ fork_N)\times(tri.\triangleleft \circ fork_N)$ "contains" the two sequences to be compared. Then composition with $map.\doteq \circ zip$ tests for equality corresponding pairs of the two sequences. Finally, $fold.\dot{\wedge}$ tests whether all the equality tests were successful.

Term (5.0) is implementable; however, it is wasteful in terms of area. Each $tri.\triangleleft$ component contains $O(N^2)$ delays, and these delays are of the same width as the width of the input stream. While the retiming laws tells us that $\triangleleft \circ \doteq$ and $\doteq \circ \triangleleft$ represent the same relation, the circuit interpretations of the two terms are different: in the first circuit we have a boolean delay, while in the second we have a delay over a pair of symbols, and the representation of each symbol presumably requires more than one bit. Hence the the informal goal in the following derivation is to transform the symbol-wide delays into boolean delays. In the course of the calculation it will become clear that we can also transform $map.\doteq$ into a single $\doteq$.

$$
\begin{array}{ll}
 & map.\doteq \circ\ zip\ \circ\ tri.\triangleleft \times tri.\triangleleft\ \circ\ fork \vartriangle (fork \circ \triangleleft^N) \\
= & \qquad \{ \quad zip \ (4.7) \text{ and delay } (3.2) \quad \} \\
 & map.\doteq \circ\ tri.\triangleleft \circ zip \circ fork \vartriangle (fork \circ \triangleleft^N) \\
= & \qquad \{ \quad \text{retiming } (3.13) \quad \}
\end{array}
$$

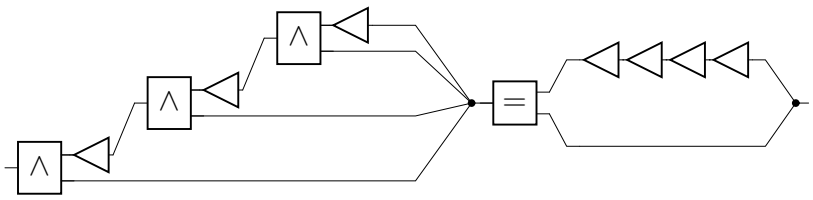


Figure 5.1: An instance of the rhs of (5.1)

$$
\begin{array}{ll}
 & \mathit{tri}.\triangleleft \circ \mathit{map}.\doteq \circ \mathit{zip} \circ \mathit{fork} \vartriangle (\mathit{fork} \circ \triangleleft^N) \\
= & \quad \{ \quad \text{delay is deterministic, (4.3)} \quad \} \\
 & \mathit{tri}.\triangleleft \circ \mathit{map}.\doteq \circ \mathit{zip} \circ \mathit{fork} \vartriangle (\triangleleft^N \circ \mathit{fork}) \\
= & \quad \{ \quad \text{fusion (2.1); } \mathit{zip} \text{ (4.8)} \quad \} \\
 & \mathit{tri}.\triangleleft \circ \mathit{map}.\doteq \circ \mathit{map}.(\iota \times \triangleleft^N) \circ \mathit{zip} \circ \mathit{fork} \vartriangle \mathit{fork} \\
= & \quad \{ \quad \mathit{zip} \text{ (4.9)} \quad \} \\
 & \mathit{tri}.\triangleleft \circ \mathit{map}.\doteq \circ \mathit{map}.(\iota \times \triangleleft^N) \circ \mathit{map}.(\iota \vartriangle \iota) \circ \mathit{fork} \\
= & \quad \{ \quad \text{map fusion (4.1); fusion (2.1)} \quad \} \\
 & \mathit{tri}.\triangleleft \circ \mathit{map}.\doteq \circ \mathit{map}.(\iota \vartriangle \triangleleft^N) \circ \mathit{fork} \\
= & \quad \{ \quad \iota, \triangleleft^N \text{ and } \doteq \text{ are deterministic, (4.3)} \quad \} \\
 & \mathit{tri}.\triangleleft \circ \mathit{fork} \circ \doteq \circ \iota \vartriangle \triangleleft^N
\end{array}
$$

A law that permits the simplification of a fold composed with a triangle is Horner's rule (4.5); the proviso

$$\dot{\wedge} \circ \triangleleft \times \triangleleft \;=\; \triangleleft \circ \dot{\wedge}$$

holds by the properties of delay (3.13) and (3.2), hence Horner's rule is applicable:

$$\mathit{fold}.\dot{\wedge} \circ \mathit{tri}.\triangleleft \;=\; \mathit{fold}.(\dot{\wedge} \circ \iota \times \triangleleft).$$

Summing up, we have proved

$$(5.1) \qquad (5.0) \;=\; \mathit{fold}.(\dot{\wedge} \circ \iota \times \triangleleft) \circ \mathit{fork} \circ \doteq \circ \iota \vartriangle \triangleleft^N,$$

and this brings the number of required memory elements from $O(N^2)$ to $O(N)$. Figure 5.0 shows a picture interpretation of (5.1).

Now, consider

(5.2) $\quad a\langle \mathit{fold}.\dot{\wedge} \circ \mathit{tri}.\lhd \circ \mathit{fork}_N\rangle b.$

The above holds if and only if, for all t,

$$a.t \;\equiv\; b.t \wedge b.(t-1) \wedge \ldots \wedge b.(t-N+1);$$

in words, $a.t$ is true whenever the last consecutive N inputs were true. All that matters then is the *number* of consecutive true inputs. This suggests that we may implement the above machinery with a counter. It is advantageous to do so, because a counter can be realized using $O(\log N)$ memory elements, a result that is better than what we obtained with Horner's rule.

The left domain of $\mathit{tri}.\lhd \circ \mathit{fork}_N$ is a tuple of boolean stream, and what we are interested in is the number of consecutive true values from the left at any time t. Let's define then cnt_n for all n-tuples of booleans as

$$\mathit{cnt}_n.(x_0,\ldots,x_{n-1}) \;=\; \uparrow\big(k :: \forall(i :\; 0 \le i < k :\; x_i \;\equiv\; \mathit{true})\big),$$

where $l \uparrow m$ is the maximum of l and m. Function cnt_n counts the number of consecutive true values from the left in an n-tuple. Clearly $\mathit{fold}_N.\dot{\wedge} \circ \dot{\mathit{cnt}}_N{}^{\cup} \circ \dot{\mathit{cnt}}_N \;=\; \mathit{fold}_N.\dot{\wedge}$, so we can rewrite the circuit in (5.2) to

(5.3) $\quad \mathit{fold}.\dot{\wedge} \circ \dot{\mathit{cnt}}_N{}^{\cup} \circ \dot{\mathit{cnt}}_N \circ \mathit{tri}.\lhd \circ \mathit{fork}_N.$

Now, $\mathit{fold}.\dot{\wedge} \circ \dot{\mathit{cnt}}_N{}^{\cup}$ is just $(\dot{=}N)$. For $\dot{\mathit{cnt}}_N \circ \mathit{tri}.\lhd \circ \mathit{fork}_N$ we reason:

$$\begin{array}{ll}
 & b\langle \dot{\mathit{cnt}}_N \circ \mathit{tri}.\lhd \circ \mathit{fork}_N\rangle a \\
\equiv & \quad \{ \quad \text{definitions, lifting} \quad \} \\
 & \forall\big(t :: \; b.t \;=\; \uparrow\big(k :: \; \forall(i :\; 0 \le i < k :\; (\lhd^i.a).t \;\equiv\; \mathit{true})\big)\big) \\
\equiv & \quad \{ \quad \text{calculus; } l \downarrow m \text{ is the minimum of } l \text{ and } m \quad \} \\
 & \quad \forall\big(t :\; a.t \;\equiv\; \mathit{false} :\; b.t \;=\; 0\big) \\
 & \wedge\, \forall\big(t :\; a.t \;\equiv\; \mathit{true} :\; b.t \;=\; (b.(t-1) \;+\; 1)\downarrow N\big) \\
\equiv & \quad \{ \quad \text{defining } l +_N m = (l+m)\downarrow N \quad \} \\
 & b\langle (K_0{}^{\triangledown}((+_N\dot{1}) \circ \lhd))^{\sigma}\rangle a
\end{array}$$

The components in the last circuit can be implemented in $O(\log N)$ area. The reduction in area is due to the change in representation defined by $\dot{\mathit{cnt}}_N$: while an N-tuple requires N wires, a number between 0 and N can be represented with just $\lceil \log N \rceil$ binary digits.

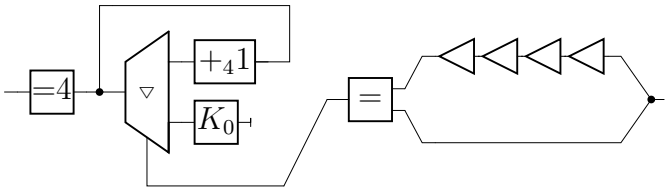


Figure 5.2: An instance of the rhs of (5.4)

Summarizing, we have proved that

$$(5.4) \qquad (5.0) \;=\; (=\dot{N}) \circ (K_0{\triangledown}((+\dot{N}1) \circ \lhd))^{\sigma} \circ \doteq \circ\, \iota \vartriangle \lhd^N.$$

Figure 5.2 shows an instance of the final design.

5.0 Conclusions

We have shown how to use Ruby to solve a simple circuit derivation problem. It is interesting to compare our derivation with the one by Rem [45]. While they are rather different, the results when interpreted as circuits are similar. Our calculations are slightly longer, but straightforward. The main invention is the introduction of $\dot{cnt}_N$, which is suggested by an analysis of the operational meaning of the circuit. The rest of the derivation is guided by the shape of the formulae.

The introduction of $\dot{cnt}_N$ allows us to break the derivation in two parts; this is an advantage of the algebraic style of reasoning.

One other advantage of our derivation is that the circuit terms we obtain are much smaller than Rem's corresponding CSP-like programs. It appears that Ruby is able to represent some circuits very compactly.

The carré derivation does not need the full generality of relations; all relations we use are deterministic, with the exception of $\dot{cnt}_N{}^{\cup}$, since $\dot{cnt}_N$ is deterministic, but not injective. Should we want to recast the whole calculation in functional terms, we'd only have to treat the proof of $fold_N.\dot{\wedge} \circ \dot{cnt}_N{}^{\cup} \circ \dot{cnt}_N \;=\; fold_N.\dot{\wedge}$ as "special". Since all the information flow in the circuit is from right to left, it is straightforward to interpret the circuit we obtain as a functional program. But restricting our calculations to functions would lead to a dead end, since in most circuits information flows in two directions,

so they are *not* functional programs; this is demonstrated by the appearance of contra-flow in the regular language recognizers of chapter 7.

Chapter 6

A round-robin scheduler

6.0 Introduction

Kropf [29] has collected together a number of problems that can be used as benchmarks for hardware *verification*, one of which is a bus arbiter. The function of the bus arbiter is to assign the use of a shared resource on each clock cycle to one out of N subsystems that may want to use it, and in such a way that no subsystem is denied access forever.

In this chapter we consider the problem of *constructing*, rather than verifying, a component of this arbiter.

6.1 The specification

A bus arbiter is a device that should assign the use of a resource at each clock cycle to at most one out of N subsystems that may want to use it, and it should do it so that no subsystem is denied access forever. More specifically, a bus arbiter is a circuit that maps a stream of N tuples of booleans, representing requests to use the resource, to a stream of N tuples of booleans, representing acknowledgements that the resource may be used.

Note that we are not interested in dealing with metastability problems. Metastability is an effect that may occur whenever a signal is sampled asynchronously; what may happen is that the signal floats in a state that cannot be interpreted as a logical "true" or as a logical "false". It should be clear that this effect cannot be modelled within the framework of this thesis. What we call "wire" here is a mathematical abstraction of physical wires.

Let us call the input stream *req* and the output stream *ack*. It is easier to think of $\mathbb{B}^N$ as a subset of $\{0,\ldots,N-1\}$, so we write $n \in req.t$ to mean that the n-th component of *req* is high at time t. The specification of the arbiter, as given by Kropf [29], is then:

0. No two output wires are asserted simultaneously: for each instant t,

$$\mid ack.t \mid \leq 1 \quad .$$

1. Acknowledge is not asserted without request:

$$ack.t \subseteq req.t \quad .$$

2. Every persistent request is eventually acknowledged: there is no pair (n,t) such that

$$\forall(t' :\ t \leq t' :\ n \in req.t' \ \wedge \ n \notin ack.t') \quad .$$

Kropf himself suggests an implementation. The arbiter should normally grant acknowledge to the request that is lowest in index, unless there is some other wire that has been asserting its request for more than a set amount of time, in which case the latter wire is granted instead. This is accomplished as follows: at any given moment there is a privileged wire. For simplicity, we may take t mod N to be the privileged wire at any time t. If wire n is privileged, and is asserting a request, and it was asserting its request the previous time it was privileged, then it is acknowledged. This way any wire will be acknowledged in less than $2N$ clock cycles. In the limit case where all requests are asserted at all times, they will be granted in round-robin fashion, i.e., each wire is granted once every N clock cycles.

One way to implement this arbiter is to construct two modules, called for instance LT and RR, the first one granting request to the lowest index asserted in its input, and the second one implementing the round-robin algorithm. The first module is combinational, while the second one has state. Module LT simply returns the lowest numbered signal that is asserted.

In the rest of this chapter the focus is on the development of the round-robin scheduler, RR. Suppose $\lhd^N$ is a function that delays an input stream by N clock cycles. That is,

$$(\lhd^N.b).t \ = \ b.(t-N) \quad .$$

Then, viewing RR as a binary relation between output ack and input req, its specification is

$$ack\langle RR\rangle req \;\equiv\; \forall(t::\; ack.t \;=\; \{t \bmod N\} \cap req.t \cap (\lhd^N.req).t) \;\;.$$

The above can be interpreted as follows: at each instant t, the set of acknowledged wires $ack.t$ is the intersection of the set of requests at time t, that is $req.t$, of the set of requests at time $t-N$, that is $(\lhd^N.req).t$, and of the singleton set $\{t \bmod N\}$, which is the singleton set of the privileged wire number at time t. The task is to construct a circuit that implements RR as specified above.

6.2 Design Steps

Because the development of the round robin scheduler is quite long we begin first by giving an overview. Some of the terms used in this overview may not be completely clear at this stage. They will however be explained in full detail later.

The steps are as follows:

0. Low level specification.

   In the first step we reformulate the given specification of RR using tuples of booleans to represent sets. The new specification takes the form

   $$(6.0) \qquad RR \;=\; \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N \;\;.$$

   In this specification the definition of RR has been split into three components. Reading from right to left, the first component $\iota \vartriangle \lhd^N$ makes two copies of the input stream, one of which is delayed N time units with respect to the other. The second component *intersect* takes two streams b and c, each of which represents a stream of sets and computes the representation of $b \dot{\cap} c$ (the stream whose tth element is $b.t \cap c.t$). In combination with the first component, this component maps input stream req to $req \dot{\cap} \lhd^N.req$. Finally, the third component *filt* implements the function $(\{t \bmod N\}\cap)$. In this way the output value at time t is

   $$ack.t \;=\; \{t \bmod N\} \;\cap\; (req.t \cap (\lhd^N.req).t) \;\;.$$

where the bracketing shows the order in which the individual terms are computed.

1. Analysis of implementability.

   There are two advantages of a modular specification. One is ease of understanding, which is of considerable help to ensuring that the informal requirements are correctly recorded in the formal specification. The other is that it is much easier to identify potential inefficiencies in the implementation. In the second step we analyse the three components with respect to implementability.

   The *filt* component can be implemented in $O(n)$ area using a cyclic multiplexer.

   The problem with the implementability of RR as specified by (6.0) is the component $\iota \vartriangle \lhd^N$. The area required for its implementation is $O(N^2)$ since it consists of N delays each with arity N. The conclusion of this phase is thus that it is this component on which we should focus our attention.

2. Goal.

   Having analysed the source of inefficiency in (6.0) we can proceed to formulating the goal. Specifically, we wish to construct memory devices $\mathit{ff}_{k,N}$ such that

   $$RR \;=\; \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \mathit{map}.(k :\; 0 \leq k < N :\; \mathit{ff}_{k,N}) \tag{6.1}$$

   and such that each such component has at most one delay element. In this way the $O(N^2)$ area required by the component

   $$\lhd^N$$

   is replaced by the $O(N)$ area required by the component

   $$\mathit{map}.(k :\; 0 \leq k < N :\; \mathit{ff}_{k,N}) \quad .$$

   (The components are called "flip-flops" because this is a name that is commonly given to memory elements.)

3. Simplification of the goal.

The first step in the achievement of the goal is to simplify it so that it becomes more manageable. The requirement on $ff_{k,N}$ is that

$$\begin{array}{ll} & \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N \\ = & \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \mathit{map}.(k:\ 0 \leq k < N:\ ff_{k,N}) \end{array}$$

However, we show, in a series of steps, how to reduce it to

$$(6.2) \qquad D_{k,N} \circ \lhd^N \ = \ D_{k,N} \circ ff_{k,N}$$

(for each k) where

$$a\langle D_{k,N}\rangle b \ \equiv \ \forall(t:\ t \bmod N \ = \ k:\ a.t \ = \ b.t) \ .$$

Note that (6.2) specifies the behaviour of the flip-flop $ff_{k,N}$ only at times t such that $t \bmod N \ = \ k$. At other times its behaviour is unspecified. This increased latitude (compared to the definition of $\lhd^N$ whose behaviour is specified at all instants) is what is needed to construct an efficient implementation of the circuit.

4. Construction of the flip-flops.

   The final step is the construction of the components $ff_{k,N}$. Again in a series of steps, we calculate that

   $$(6.3) \qquad ff_{k,N} \ = \ (\lhd \circ \mathit{cmx}_{k,N})^\sigma \ ,$$

   where $\mathit{cmx}_{k,N}$ is a cyclic multiplexer. Thus the implementation of the flip-flop does indeed require only one delay element. The combination of (6.1) and (6.3) is then the desired implementation of the round robin scheduler.

This then is the overview. Let us now present the full details.

6.3 Low Level Specification

6.3.0 Bit Representation

Let us recall the original specification of the round robin scheduler. For input stream b and output stream a,

$$a\langle RR\rangle b \ \equiv \ \forall(t::\ a.t \ = \ \{t \bmod N\} \cap b.t \cap (\lhd^N.b).t) \ .$$

As is usual we choose to represent a subset of a set of N elements by a sequence of N bits. A stream of subsets of $\{0, \ldots, N-1\}$ is represented by a N-tuple of boolean streams. Let a be such a N-tuple. We denote the k-th stream in a by a_k; the set that a represents at time t is $\{k \mid 0 \leq k < N \ \wedge \ a_k.t = \mathit{true}\}$.

The intersection operator on sets is then translated into the conjunction operator mapped over the N input wires. The implementation of the relation R where

$$a\langle R\rangle b \ \equiv \ \forall(t :: \ a.t \ = \ b.t \cap (\lhd^N.b).t)$$

is easily derived. Specifically, we have:

$$\begin{array}{ll} & \forall(t :: \ a.t \ = \ b.t \cap (\lhd^N.b).t) \\ \equiv & \quad \{ \quad \text{representation of sets as sequence of } N \text{ bits} \quad \} \\ & \forall(t, k : \ 0 \leq k < N : \ a_k.t \ \equiv \ b_k.t \ \wedge \ (\lhd^N.b_k).t) \\ \equiv & \quad \{ \quad \text{definition of } \mathit{map} \quad \} \\ & \forall(t :: \ a.t \ = \ (\mathit{map}_N.\wedge).(b.t \, , (\lhd^N.b).t)) \\ \equiv & \quad \{ \quad \text{definition of } \mathit{zip} \quad \} \\ & \forall(t :: \ a.t \ = \ (\mathit{map}_N.\wedge).((\mathit{zip}_N.(b \, , \lhd^N.b)).t)) \\ \equiv & \quad \{ \quad \text{lifting, definition of split and composition} \quad \} \\ & a \ = \ (\mathit{map}_N.\dot{\wedge} \circ \mathit{zip}_N \circ \iota \vartriangle \lhd^N).b \ . \end{array}$$

Whence

$$R \ = \ \mathit{intersect} \circ \iota \vartriangle \lhd^N$$

where

$$\mathit{intersect} \ = \ \mathit{map}_N.\dot{\wedge} \circ \mathit{zip}_N \ .$$

See figure 6.o on the next page for a picture interpretation of R. It follows that

$$RR \ = \ \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N \tag{6.4}$$

where the component *filt* must satisfy the requirement:

$$a\langle \mathit{filt}\rangle b \ \equiv \ \forall(t :: \ a.t \ = \ \{t \bmod N\} \cap b.t) \ .$$

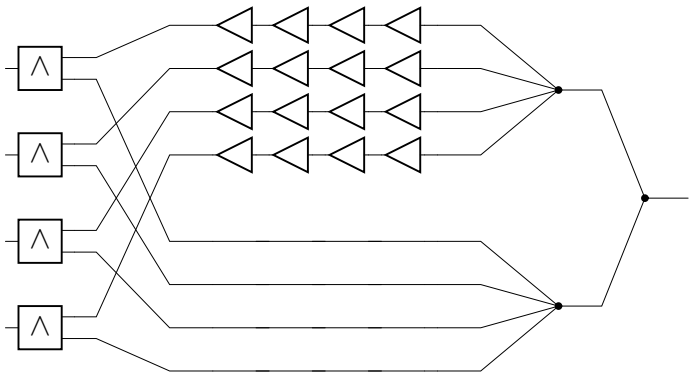


Figure 6.o: An instance of R

6.3.1 Implementing The Filter Component

With the exception of *filt*, it is clear that all components in (6.4) can be implemented directly. In this section we consider how *filt* is implemented.

In terms of the bit representation *filt* must satisfy:

$$(6.5) \qquad \begin{array}{rcl} a\langle \mathit{filt} \rangle b & \equiv & \forall(t,k:\ t \bmod N\ =\ k:\ a_k.t\ =\ b_k.t) \\ & \wedge & \forall(t,k:\ t \bmod N\ \neq\ k:\ a_k.t\ =\ \mathit{false}) \end{array}$$

In software, *filt* would be implemented with a straightforward if-then-else statement. In Ruby the full generality of an if-then-else statement is typically shunned, since it enjoys few useful algebraic properties (see Jones [24]). In this case, however, the full generality is not needed and the component can be implemented using a cyclic multiplexer $cmx_{k,N}$.

Comparing the definition of *filt* with that of $cmx_{k,N}$ it is clear that *filt* can be implemented by pairing each of the N bits of the input stream with a stream of *false* bits and passing each pair of bits to the corresponding cyclic multiplexer. That is,

$$(6.6) \qquad \mathit{filt}\ =\ \mathit{map}.(k:\ 0 \le k < N:\ cmx_{k,N} \circ \iota \vartriangle K_F)\ .$$

where K_F is a circuit that ignores its input and constantly outputs the value F (false).

This concludes this section. The combination of (6.4) and (6.6) is a correct implementation of the round-robin scheduler:

$$RR \quad = \quad map.(k:\ 0 \le k < N:\ cmx_{k,N} \circ \iota \vartriangle K_F) \circ map_N.\dot{\wedge} \circ zip_N \circ \iota \vartriangle \lhd^N \quad .$$

6.4 Efficiency Analysis and the Goal

One advantage of a modular specification like (6.4) is that it simplifies the task of identifying potential inefficiencies. We need only examine each component in turn.

Assuming that a cyclic multiplexer has an efficient implementation, it is clear that the two components *filt* and *intersect* have efficient implementations. The bottleneck in the implementation is in fact the component $\lhd^N$. Note that the input arity of this component is N since the input stream is in fact a stream of N bits. The total area required for its implementation is thus $O(N^2)$. On the other hand, it seems plausible that an $O(N)$ implementation can be found for RR (although not the component $\lhd^N$) since at any stage only N bits need to be recorded. Specifically, at any time t it suffices to record the value of the kth input bit only at the last time that it was privileged.

We can express our intuition about the memory component that is required as follows. For each input bit k we replace the $\lhd^N$ component by a memory element $ff_{k,N}$, "ff" standing for "flip-flop" (this being the name often given by circuit designers to memory elements). That is, we wish to design $ff_{k,N}$ such that

$$\begin{array}{ll} & filt \circ intersect \circ \iota \vartriangle \lhd^N \\ = & filt \circ intersect \circ \iota \vartriangle map.(k:\ 0 \le k < N:\ ff_{k,N}) \quad . \end{array}$$

Moreover, the implementation of the flip-flops should involve at most one delay element.

6.5 Simplifying the Goal

In the following discussion it will be useful to introduce the name *specRR* for the term

$$filt \circ intersect \circ \iota \vartriangle \lhd^N$$

and *impRR* for the term

$$\mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \mathit{map}.(k :: 0 \leq k < N :: \mathit{ff}_{k,N}) \quad .$$

The goal is to derive $\mathit{ff}_{k,N}$ such that $\mathit{specRR} = \mathit{impRR}$. In this section we simplify the goal by splitting it up into separate requirements for the individual flip-flops and by eliminating the "$\mathit{map}.\dot{\wedge}$" term in the definition of *intersect*. We begin by splitting the goal up.

Observe first that $\mathit{zip} \circ \iota \vartriangle \lhd^N$ and $\mathit{zip} \circ \iota \vartriangle \mathit{map}.(k :: 0 \leq k < N :: \mathit{ff}_{k,N})$ can be written as maps. Specifically,

$$\begin{array}{ll} & \mathit{zip}_N \circ \iota \vartriangle \lhd^N \\ = & \quad \{ \quad \text{arity of } \mathit{zip} \quad \} \\ & \mathit{zip}_N \circ N{\times}N \circ \iota \vartriangle \lhd^N \\ = & \quad \{ \quad \text{fusion, polymorphism of } \lhd\text{: (3.2)} \quad \} \\ & \mathit{zip}_N \circ \mathit{map}_N.\iota \vartriangle \mathit{map}_N.\lhd^N \\ = & \quad \{ \quad (4.8) \quad \} \\ & \mathit{map}_N.(\iota \vartriangle \lhd^N) \quad . \end{array}$$

Similarly,

$$\begin{array}{ll} & \mathit{zip} \circ \iota \vartriangle \mathit{map}.(k :: 0 \leq k < N :: \mathit{ff}_{k,N}) \\ = & \mathit{map}.(k :: 0 \leq i < N :: \iota \vartriangle \mathit{ff}_{k,N}) \quad . \end{array}$$

Substituting these terms back into the definitions of *specRR* and *impRR*, we conclude that $\mathit{ff}_{k,N}$ must satisfy

$$\begin{array}{ll} & \mathit{map}.(k :: \mathit{cmx}_{k,N} \circ \iota \vartriangle K_F) \circ \mathit{map}.\dot{\wedge} \circ \mathit{map}.(\iota \vartriangle \lhd^N) \\ = & \mathit{map}.(k :: \mathit{cmx}_{k,N} \circ \iota \vartriangle K_F) \circ \mathit{map}.\dot{\wedge} \circ \mathit{map}.(k :: \iota \vartriangle \mathit{ff}_{k,N}) \quad . \end{array} \tag{6.7}$$

Thus, by map fusion and introducing the abbreviation $\varphi_{k,N}$ where, by definition,

$$\varphi_{k,N} \;=\; \mathit{cmx}_{k,N} \circ \iota \vartriangle K_F \quad , \tag{6.8}$$

we have obtained individual requirements on each flip-flop. Specifically, we require that

$$\forall(k :: 0 \leq k < N :: \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \lhd^N \;=\; \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \mathit{ff}_{k,N}) \quad . \tag{6.9}$$

In order to remove the spurious conjunction we observe that $\varphi_{k,N}$ satisfies

$$a\langle\varphi_{k,N}\rangle b \equiv \forall(t:\ t \bmod N = k:\ a.t = b.t) \\ \wedge\ \forall(t:\ t \bmod N \neq k:\ a.t = F)\ .$$

From this definition, using the fact that $F \wedge F = F$, it is easy to see that

$$\varphi_{k,N} \circ \dot{\wedge} = \dot{\wedge} \circ \varphi_{k,N} \times \varphi_{k,N}\ . \tag{6.10}$$

(We leave the proof to the reader.) Hence we may rewrite (6.9): for all k, $0 \leq k < N$:

$$\begin{array}{ll} & \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \lhd^N = \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \mathit{ff}_{k,N} \\ \equiv & \quad \{ \text{ above property of } \varphi_{k,N} \ \} \\ & \dot{\wedge} \circ \varphi_{k,N} \times \varphi_{k,N} \circ \iota \vartriangle \lhd^N = \dot{\wedge} \circ \varphi_{k,N} \times \varphi_{k,N} \circ \iota \vartriangle \mathit{ff}_{k,N} \\ \equiv & \quad \{ \text{ fusion } \} \\ & \dot{\wedge} \circ \varphi_{k,N} \vartriangle (\varphi_{k,N} \circ \lhd^N) = \dot{\wedge} \circ \varphi_{k,N} \vartriangle (\varphi_{k,N} \circ \mathit{ff}_{k,N}) \\ \Leftarrow & \quad \{ \text{ Leibniz } \} \\ & \varphi_{k,N} \circ \lhd^N = \varphi_{k,N} \circ \mathit{ff}_{k,N}\ . \end{array}$$

So we have reduced the specification of $\mathit{ff}_{k,N}$ from (6.7) to

$$\varphi_{k,N} \circ \lhd^N = \varphi_{k,N} \circ \mathit{ff}_{k,N}\ . \tag{6.11}$$

One further simplification is possible. Recalling the definition of $\varphi_{k,N}$ (equation (6.8)):

$$\varphi_{k,N} = \mathit{cmx}_{k,N} \circ \iota \vartriangle K_F$$

and the definition of $\mathit{cmx}_{k,N}$

$$a\langle \mathit{cmx}_{k,N}\rangle(b,c) \equiv \forall(t:\ t \bmod N = k:\ a.t = b.t) \\ \wedge\ \forall(t:\ t \bmod N \neq k:\ a.t = c.t)$$

it is clear that $\mathit{cmx}_{k,N}$ ignores the first component of its input stream whenever $t \bmod N \neq k$. As a consequence, $\varphi_{k,N}$ ignores its input stream entirely whenever $t \bmod N \neq k$. We can express this formally by introducing the relation $D_{k,N}$ defined by

$$a\langle D_{k,N}\rangle b \equiv \forall(t:\ t \bmod N = k:\ a.t = b.t)\ .$$

Two input streams are related by $D_{k,N}$ whenever they are equal for all times t such that $t \bmod N = k$; at all other times no relation between the two streams is required. Thus the fact that $cmx_{k,N}$ ignores the first input stream whenever $t \bmod N \neq k$ is expressed by the equation:

$$cmx_{k,N} \circ D_{k,N} \times \iota \quad = \quad cmx_{k,N} \quad ,$$

and the fact that $\varphi_{k,N}$ ignores its input stream entirely whenever $t \bmod N \neq k$ is expressed by the equation:

$$\varphi_{k,N} \circ D_{k,N} \quad = \quad \varphi_{k,N} \quad .$$

The verification of the former equation follows by straightforward pointwise reasoning. The derivation of the latter equation proceeds as follows:

$$\begin{array}{ll} & \varphi_{k,N} \\ = & \quad \{ \quad \text{definition of } \varphi_{k,N} \quad \} \\ & cmx_{k,N} \circ \iota \vartriangle K_F \\ = & \quad \{ \quad \text{above} \quad \} \\ & cmx_{k,N} \circ D_{k,N} \times \iota \circ \iota \vartriangle K_F \\ = & \quad \{ \quad \text{fusion} \quad \} \\ & cmx_{k,N} \circ D_{k,N} \vartriangle K_F \\ = & \quad \{ \quad K_F = K_F \circ D_{k,N} \quad \} \\ & cmx_{k,N} \circ D_{k,N} \vartriangle (K_F \circ D_{k,N}) \\ = & \quad \{ \quad (2.0),\ K_F \text{ is a left condition} \quad \} \\ & cmx_{k,N} \circ \iota \vartriangle K_F \circ D_{k,N} \\ = & \quad \{ \quad \text{definition of } \varphi_{k,N} \quad \} \\ & \varphi_{k,N} \circ D_{k,N} \quad . \end{array}$$

Substituting this equation in (6.11) we obtain the final simplification to the requirement on the flip-flops.

$$\begin{array}{ll} & \varphi_{k,N} \circ \lhd^N \quad = \quad \varphi_{k,N} \circ \mathit{ff}_{k,N} \\ \equiv & \quad \{ \quad \text{above} \quad \} \\ & \varphi_{k,N} \circ D_{k,N} \circ \lhd^N \quad = \quad \varphi_{k,N} \circ D_{k,N} \circ \mathit{ff}_{k,N} \\ \Leftarrow & \quad \{ \quad \text{Leibniz} \quad \} \\ & D_{k,N} \circ \lhd^N \quad = \quad D_{k,N} \circ \mathit{ff}_{k,N} \quad . \end{array}$$

In summary, the requirement on $\mathit{ff}_{k,N}$ is:

$$D_{k,N} \circ \lhd^N \quad = \quad D_{k,N} \circ \mathit{ff}_{k,N} \quad . \tag{6.12}$$

6.6 Construction of the flip-flops

From the definition of $D_{k,N}$ it is clear that (6.12) specifies the behaviour of $ff_{k,N}$ only at times t such that $t \bmod N = k$; at all other times there is complete latitude in its behaviour. It is this latitude that we now exploit.

The component $\lhd^N$ can be seen as a memory element that stores N input values. This is because its implementation demands that the input value at each time t is recorded for use at time $t+N$ after which it can be discarded. From (6.12) it is clear, however, that it suffices to record the input value only at times t such that $t \bmod N = k$, that is once every N clock beats. The crucial step in the calculation of $ff_{k,N}$ below is thus to replace the function mapping t to $t-N$ by a function that is constant for N time intervals. A well known example of such a function is the function mapping t to $t \operatorname{div} N$. But this function does not suffice because of the additional requirement that the function's value should equal $t-N$ when $t \bmod N = k$. Noting that

$$(t \operatorname{div} N)*N = t - t \bmod N$$

is the clue to discovering the appropriate function.

Since it occurs twice in the following calculation it is useful to begin by observing that, in general, for arbitrary function f

$$a\langle D_{k,N} \circ f\rangle b \equiv \forall(t\colon t \bmod N = k\colon a.t = (f.b).t) \ .$$

This is because

$$\begin{array}{ll} & a\langle D_{k,N} \circ f\rangle b \\ \equiv & \quad \{ \quad \text{composition and one point rule} \quad \} \\ & a\,\langle D_{k,N}\rangle\, f.b \\ \equiv & \quad \{ \quad \text{definition of } D_{k,N} \quad \} \\ & \forall(t\colon t \bmod N = k\colon a.t = (f.b).t) \ . \end{array}$$

Now,

$$\begin{array}{ll} & a\langle D_{k,N} \circ \lhd^N\rangle b \\ \equiv & \quad \{ \quad \text{above, definition of } \lhd^N \quad \} \\ & \forall(t\colon t \bmod N = k\colon a.t = b.(t-N)) \\ \equiv & \quad \{ \quad \text{This is the crucial step discussed above.} \end{array}$$

$$
\begin{array}{ll}
 & \quad \text{We replace "}t{-}N\text{" using the property of} \\
 & \quad \text{modular arithmetic:} \\
 & \quad t \bmod N \;=\; k \;\Rightarrow\; N{-}1 \;=\; (t{-}k{-}1) \bmod N \quad \} \\
 & \forall(t{:}\; t \bmod N \;=\; k{:}\; a.t \;=\; b.(t-1-(t{-}k{-}1) \bmod N)) \\
\equiv & \quad \{ \quad \text{Define } \mathit{ff}_{k,N} \text{ by} \\
 & \quad (\mathit{ff}_{k,N}.b).t \;=\; b.(t-1-(t{-}k{-}1) \bmod N) \quad \} \\
 & \forall(t{:}\; t \bmod N \;=\; k{:}\; a.t \;=\; (\mathit{ff}_{k,N}.b).t) \\
\equiv & \quad \{ \quad \text{above} \quad \} \\
 & a\langle D_{k,N} \circ \mathit{ff}_{k,N}\rangle b \quad .
\end{array}
$$

We have thus calculated a functional specification of $\mathit{ff}_{k,N}$:

$$(\mathit{ff}_{k,N}.b).t \;=\; b.(t-1-(t{-}k{-}1) \bmod N) \quad .$$

The construction of an implementation for $\mathit{ff}_{k,N}$ amounts to verifying that the function mapping t to $t-1-(t{-}1{-}k) \bmod N$ is indeed constant over N time intervals. To be precise, we explore when $(\mathit{ff}_{k,N}.b).(t+1)$ equals $(\mathit{ff}_{k,N}.b).t$:

$$
\begin{array}{ll}
 & (\mathit{ff}_{k,N}.b).(t+1) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & b.(t \;-\; (t-k) \bmod N) \\
= & \quad \{ \quad \bullet \quad \text{Suppose } (t-k) \bmod N \;\neq\; 0 \text{ . Then} \\
 & \qquad (t-k) \bmod N \;=\; (t-k-1) \bmod N + 1 \qquad \} \\
 & b.(t \;-\; ((t-k-1) \bmod N + 1)) \\
= & \quad \{ \quad \text{arithmetic} \quad \} \\
 & b.(t-1-(t{-}k{-}1) \bmod N) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & (\mathit{ff}_{k,N}.b).t \quad .
\end{array}
$$

Thus if $(t-k) \bmod N \;\neq\; 0$, $(\mathit{ff}_{k,N}.b).(t+1) \;=\; (\mathit{ff}_{k,N}.b).t$. Also, if $(t-k) \bmod N \;=\; 0$, it is obvious that $(\mathit{ff}_{k,N}.b).(t+1) \;=\; b.t$. So $\mathit{ff}_{k,N}.b$ is defined by the following equations:

$$
\begin{array}{lcll}
(\mathit{ff}_{k,N}.b).(t+1) & = & (\mathit{ff}_{k,N}.b).t & \text{if } (t-k) \bmod N \;\neq\; 0 \\
(\mathit{ff}_{k,N}.b).(t+1) & = & b.t & \text{if } (t-k) \bmod N \;=\; 0 \quad .
\end{array}
$$

We recognize in these equations a combination of the cyclic multiplexer and a feedback. Indeed,

$$\mathit{ff}_{k,N}.b \;\equiv\; m.(b\,,\mathit{ff}_{k,N}.b)$$

where

$$\begin{array}{lcl}(m.(b,c)).(t+1) & = & c.t \quad \text{if } (t-k) \bmod N \neq 0\\ (m.(b,c)).(t+1) & = & b.t \quad \text{if } (t-k) \bmod N = 0 \quad .\end{array}$$

By (3.7) this equivales $\mathit{ff}_{k,N} = m^\sigma$ where, as is obvious from the definitions of the circuit multiplexer and delay, $m = \lhd \circ \mathit{cmx}_{k,N}$. Thus,

$$\mathit{ff}_{k,N} \;=\; (\lhd \circ \mathit{cmx}_{k,N})^\sigma \quad . \tag{6.13}$$

In summary, the implementation of RR we have come to is

$$RR \;=\; \mathit{filt} \circ \mathit{map}.\dot{\wedge} \circ \mathit{zip} \circ \iota \vartriangle \mathit{map}.(k:\; 0 \le k < N :\; \mathit{ff}_{k,N}) \quad ,$$

or equivalently, exploiting (4.8)

$$RR \;=\; \mathit{filt} \circ \mathit{map}.\dot{\wedge} \circ \mathit{map}.(k:\; 0 \le k < N :\; \iota \vartriangle \mathit{ff}_{k,N}) \quad , \tag{6.14}$$

where $\mathit{ff}_{k,N}$ is defined by (6.13) and *filt* is defined by (6.6). A picture of (6.14) is in figure 6.1 on the facing page.

6.7 Conclusions

In this chapter we have shown how to transform a specification of a circuit into an implementation that is efficient in terms of area. The problem is complicated by the presence of cyclic multiplexers. One reason why this development is interesting is that it shows how a relatively simple optimization can be derived, rather than verified.

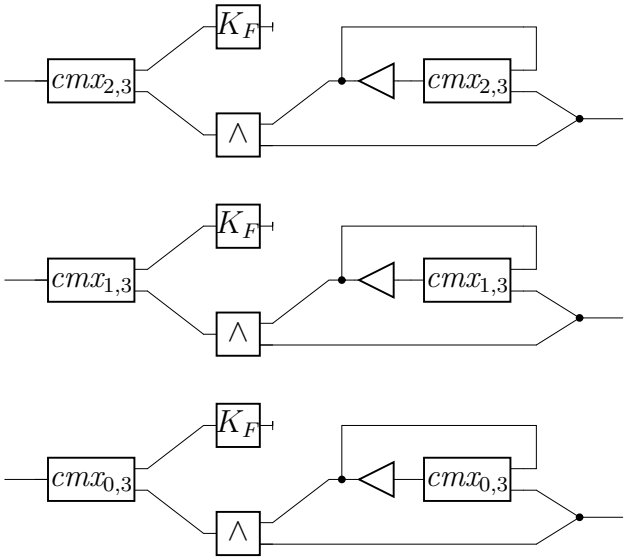


Figure 6.1: An instance of (6.14)

Chapter 7

Regular language recognizers

In 1982, Foster and Kung presented a specialised silicon compiler that constructs recognizers for regular languages [18]. By "silicon compiler" a program is meant, that produces a circuit's description from a specification of the circuit's behaviour. 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 [3] 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 [17]. Otherwise the formal calculations in Backhouse's report were disappointingly complicated and not judged by its author to be worthy of widespread publication.

Here 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, that is essentially a functional program, followed by a transformation of this design into two different systolic versions, using standard techniques of "slowdown", "retiming" and "pipelining" [25].

A formal derivation of a similar compiler has also been given by Kaldewaij and Zwaan [27], 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, we present

a class of recognizers that 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; and a second class of recognizers that can be assigned a clock period independent of the number of choice operators, although it depends on the number of sequence and stars operators.

This chapter is organized as follows: we first present the specification of the problem; then a non-systolic implementation is derived, in a style similar to functional program derivation. Then we present two different ways to transform the non-systolic design into systolic ones, each of which has different properties.

7.0 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 $\mathcal{T}$. The language associated with a regular expressions is defined as usual:

$$\begin{array}{rcll} \mathcal{L}.t & = & \{t\} & \text{for all } t \in \mathcal{T} \\ \mathcal{L}.(E+F) & = & \mathcal{L}.E \cup \mathcal{L}.F & \\ \mathcal{L}.(E\,;F) & = & \{w \cdot z \mid w \in \mathcal{L}.E, z \in \mathcal{L}.F\} & \\ \mathcal{L}.(E^*) & = & \mu(X \mapsto \varepsilon \cup \{w \cdot z \mid w \in X, z \in \mathcal{L}.E\}). & \end{array}$$

In what follows we will identify the language associated with a regular expression with the regular expression itself; i.e., we will write E in place of $\mathcal{L}.E$. The context should make clear which one is meant.

To begin with, we will define a mapping from $\mathcal{E}$ to the set of circuits. 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

(We shall use 1 and 0 as shorthands for *true* and *false*). 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 [18] 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 $e.n \;\equiv\; n>0 \;\wedge\; (n$ 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 section (7.0.0)):

$$\begin{array}{rcll} m\langle mem.(t,a)\rangle n & \equiv & m = n+1 \ \wedge\ a.n = t & \text{for all } t \in \mathcal{T} \\ mem.(E{+}F,a) & = & mem.(E,a) \cup mem.(F,a) & \\ mem.(E\,;F,a) & = & mem.(F,a) \circ mem.(E,a) & \\ mem.(E^*,a) & = & (mem.(E,a))^*. & \end{array} \tag{7.0}$$

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 are now ready to give the specification. We say that a circuit f *recognizes* regular expression E when the following holds, for all $a \in \mathit{Stream}(\mathcal{T})$ and $e \in \mathit{Stream}(\mathbb{B})$:

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

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.

7.0.0 Proof of the properties of mem

We prove the laws claimed in section 7.0 about mem, equations (7.0). 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

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

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

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

Similarly, for composition:

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

Finally, for "star" we have:

$$
\begin{array}{ll}
 & mem.(E^*,a) = mem.(E,a)^* \\
\equiv & \quad \{ \quad \text{definitions of "star" on languages and relations} \quad \} \\
 & mem.(\mu(X \mapsto \varepsilon + X\,;E),a) = \mu(X \ \mapsto \ I \cup X \circ mem.(E,a)) \\
\Leftarrow & \quad \{ \quad E{\mapsto}mem.(E,a) \text{ is universally } \cup\text{-distributive}
\end{array}
$$

$$
\begin{array}{ll}
 & \qquad\qquad \text{so we may use } \mu\text{-fusion [39]} \qquad \} \\
 & \forall(X ::\ mem.(\varepsilon + X\,;E,a) \;=\; I \cup mem.(X,a) \circ mem.(E,a)) \\
\equiv & \qquad \{ \qquad \text{above} \qquad \} \\
 & \mathit{true}
\end{array}
$$

This concludes the proof of the properties of mem.

7.1 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 start with a lemma:

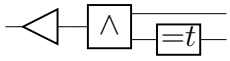
$$
\begin{array}{ll}
 & m \;\in\; mem.(t,a) \circ tt.e \\
\equiv & \qquad \{ \qquad \text{composition} \qquad \} \\
 & \exists(n ::\ m\langle mem.(t,a)\rangle n \;\wedge\; n \;\in\; tt.e) \\
\equiv & \qquad \{ \qquad mem \qquad \} \\
 & \exists(n ::\ n = m-1 \;\wedge\; a.n = t \;\wedge\; n \;\in\; tt.e)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

From this we obtain:

$$
\begin{array}{ll}
 & f \text{ recognizes letter } t \\
\equiv & \qquad \{ \qquad \text{definition} \qquad \} \\
 & \forall(a,e ::\ tt.f.(a,e) \;=\; mem.(t,a) \circ tt.e) \\
\equiv & \qquad \{ \qquad \text{sets} \qquad \}
\end{array}
$$

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

A picture of the recognizer for t is below:



The character recognizer is clearly implementable, since the direction of the delay is compatible with the direction of information through the "and" component.

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,

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

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,

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

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

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 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:

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

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

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

Now,

$$
\begin{array}{ll}
 & tt.b = mem.(E^*,a) \circ tt.e \\
\equiv & \quad \{ \quad mem\ (7.0) \quad \} \\
 & tt.b = (mem.(E,a))^* \circ tt.e \\
\equiv & \quad \{ \quad \varepsilon \notin E, \text{ so } mem.(E,a) \text{ is well-founded.} \\
 & \qquad \text{Unique extension property (2.2).} \quad \} \\
 & tt.b \ = \ tt.e \cup mem.(E,a) \circ tt.b \\
\equiv & \quad \{ \quad f \text{ recognizes } E \quad \} \\
 & tt.b = tt.e \cup tt.f.(a,b) \\
\equiv & \quad \{ \quad tt \quad \} \\
 & tt.b = tt.(e \mathbin{\dot\vee} f.(a,b)) \\
\equiv & \quad \{ \quad tt \text{ is an isomorphism} \quad \} \\
 & b = e \mathbin{\dot\vee} f.(a,b) \\
\equiv & \quad \{ \quad \text{calculus; define } reorg.((x,y),z) = (y,(x,z)) \quad \} \\
 & b = (\dot\vee \circ \iota\times f \circ reorg).((a,e),b).
\end{array}
$$

Thus

$$
g \text{ recognizes } E^*
$$

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

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:

$$
\begin{array}{rcll}
\tau.t & = & \lhd \circ \dot{\wedge} \circ (\dot{=} t)\times\iota & \text{for all } t \in \mathcal{T} \\
\tau.(E{+}F) & = & \dot{\vee} \circ \tau.E \ \triangle\ \tau.F & \\
\tau.(E\,;F) & = & \tau.F \circ \ll \triangle\ \tau.E & \\
\tau.E^{*} & = & (\dot{\vee} \circ \iota \times \tau.E \circ \mathit{reorg})^{\sigma} &
\end{array}
\tag{7.1}
$$

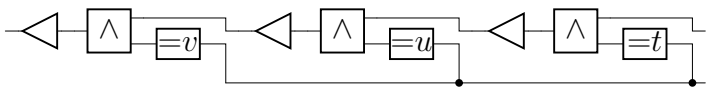
and such that, for all regular expressions E, $\tau.E$ recognizes E.

7.2 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

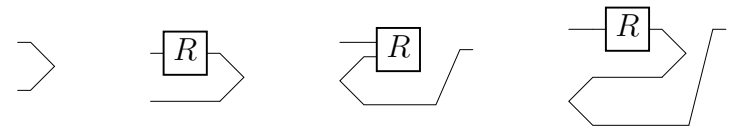
$$(\lhd \circ \dot{\wedge} \circ (\dot{=} v)\times\iota) \ \circ\ \ll\triangle\ (\lhd \circ \dot{\wedge} \circ (\dot{=} u)\times\iota) \ \circ\ \ll\triangle\ (\lhd \circ \dot{\wedge} \circ (\dot{=} t)\times\iota).$$

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 3). Then we make the design modular, by designing a separate cell for each operator (except for sequence, which we regard as the "basic" operator). Finally we apply retiming and slowdown, in order to make the design as systolic as possible.



term *bend.R* *str.R* *str.bend.R*

Figure 7.0: Picture interpretations

As we saw in the example at the end of section 3, 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

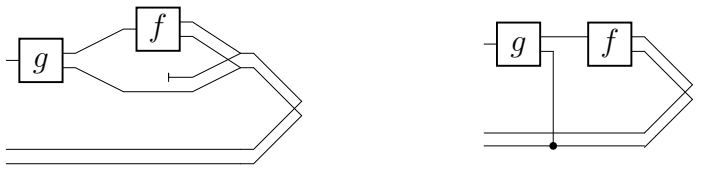
$$str.bend.R = R \tag{7.2}$$

(see figure 7.0). Since $bend$ is defined without mentioning delays, we have that for all R,

$$slow.bend.R = bend.slow.R. \tag{7.3}$$

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 7.1: 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 7.1. Let *plumb* be a wiring relation defined by

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

It holds that

$$\iota\times(g \circ \ll \triangle f) \circ term = \iota\times g \circ plumb \circ \iota\times f \circ term \tag{7.4}$$

(The proof is omitted. A simple pointwise calculation establishes the property.) By defining a new function $\upsilon.E = \iota \times slow.\tau.E$, equation (7.4) may be rewritten as

$$\rho.(E\,;F) = \upsilon.F \circ plumb \circ \rho.E. \tag{7.5}$$

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 (7.5). Every regular expression E other than a sequence will be implemented by a "cell" equal to $\upsilon.E$. For instance, the interconnection of the cells of the recognizer for the expression $t^*\ ;\ \upsilon\ ;(t+(u\,;u))$ would have the shape shown in figure 7.2.

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

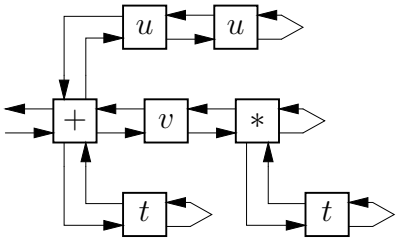


Figure 7.2: Cells layout for the recognizer of $t^* \;;\; v \;;(t+(u\,;u))$

$$
\begin{array}{ll}
 & \upsilon.(E+F) \\
= & \quad \{ \quad \text{definition of } \upsilon \quad \} \\
 & \iota \times \mathit{slow}.\tau.(E+F) \\
= & \quad \{ \quad \text{definition of } \tau \quad \} \\
 & \iota \times \mathit{slow}.(\dot{\vee} \circ \tau.E \vartriangle \tau.F) \\
= & \quad \{ \quad \text{slowing is the same as doubling delays} \quad \} \\
 & \iota \times (\dot{\vee} \circ \mathit{slow}.\tau.E \vartriangle \mathit{slow}.\tau.F) \\
= & \quad \{ \quad \text{equation (7.2)} \quad \} \\
 & \iota \times (\dot{\vee} \circ \mathit{str}.\mathit{bend}.\mathit{slow}.\tau.E \vartriangle \mathit{str}.\mathit{bend}.\mathit{slow}.\tau.F) \\
= & \quad \{ \quad \text{slow commutes with bend (7.3)} \quad \} \\
 & \iota \times (\dot{\vee} \circ \mathit{str}.\mathit{slow}.\mathit{bend}.\tau.E \vartriangle \mathit{str}.\mathit{slow}.\mathit{bend}.\tau.F) \\
= & \quad \{ \quad \text{definition of } \rho \quad \} \\
 & \iota \times (\dot{\vee} \circ \mathit{str}.\rho.E \vartriangle \mathit{str}.\rho.F)
\end{array}
$$

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. The picture interpretation for $\upsilon.(E+F)$ can be seen in figure 7.3.

We have a similar derivation for *star*:

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

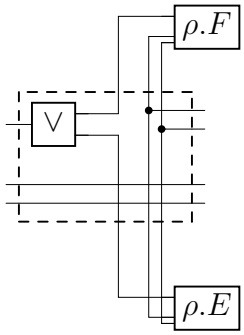


Figure 7.3: Picture interpretation of $\upsilon.(E+F)$

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

$$\begin{array}{ll} & \upsilon.t \\ = & \quad \{ \quad \text{definitions of } \rho, \tau \text{ and } \mathit{slow} \quad \} \\ & \iota\times(\lhd \circ \lhd \circ \dot{\wedge} \circ (\dot{=}t)\times\iota) \\ = & \quad \{ \quad \text{retiming (3.13) and delays (3.14), (3.3)} \quad \} \\ & \rhd\times(\lhd \circ \dot{\wedge} \circ (\dot{=}t)\times\iota) \circ \lhd. \end{array}$$

Here we'd like to get rid of the rightmost $\lhd$, but with the current definition of υ we can't. However, the circuits we generate are always of the shape $\upsilon.E \circ \ldots \circ \rho.F$, and by the definition of ρ and equation (3.16) 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 υ to

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

By this definition, the expressions for $\upsilon.(E+F)$ and $\upsilon.E^*$ that we previously derived are still valid, as well as equation (7.5); and the derivation of $\upsilon.t$ becomes:

$$\begin{array}{ll} & \upsilon.t \circ R \circ \top\!\!\top \\ = & \quad \{ \quad \text{above} \quad \} \\ & \rhd\times(\lhd \circ \dot{\wedge} \circ (\dot{=}t)\times\iota) \circ \lhd \circ R \circ \top\!\!\top \\ = & \quad \{ \quad \text{equation (3.16)} \quad \} \\ & \rhd\times(\lhd \circ \dot{\wedge} \circ (\dot{=}t)\times\iota) \circ R \circ \top\!\!\top. \end{array}$$

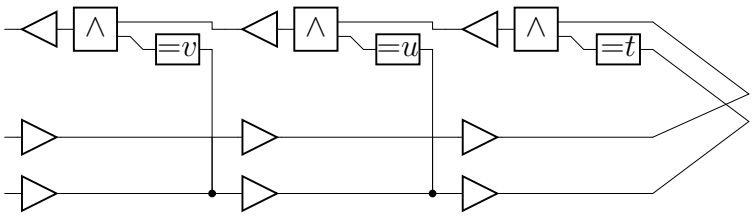


Figure 7.4: A systolic recognizer for the expression $t\,;u\,;v$

Summarizing our results, we have

$$
\begin{array}{rcll}
\rho.(E\,;F) & = & \upsilon.F \circ \mathit{plumb} \circ \rho.E & \\
\rho.E & = & \upsilon.E \circ \mathit{term} & \\
\upsilon.(E+F) & = & \iota\times(\dot{\vee} \circ \mathit{str}.\rho.E \mathbin{\triangle} \mathit{str}.\rho.F) & \\
\upsilon.E^* & = & \iota \times (\dot{\vee} \circ \iota \times \mathit{str}.\rho.E \circ \mathit{reorg})^\sigma & \\
\upsilon.t & = & \rhd\times(\lhd \circ \dot{\wedge} \circ (\dot{=}t)\times\iota) & \text{for all } t \in \mathcal{T}.
\end{array}
\tag{7.6}
$$

These equations can be interpreted as a functional program that translates regular expressions to systolic recognizers; see section 8.

7.3 A choice-privileged design

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 7.4.

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 shown in figure 7.5, and here we have combinational paths whose length depends on the number of occurrences of "+" in the expression. The design presented in section 7.2 is well-optimized for expressions that contain many more ";" operators than

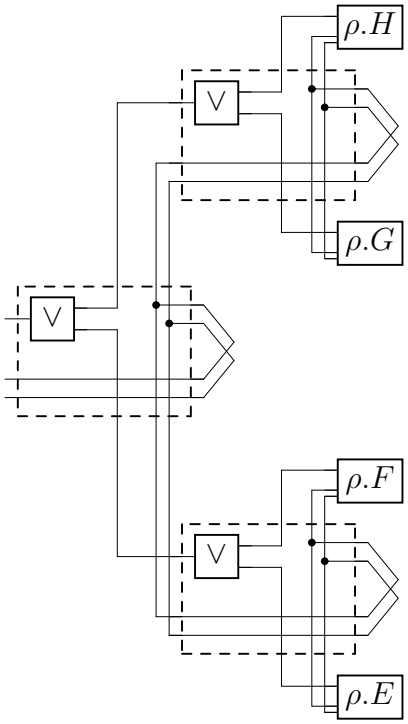


Figure 7.5: A composition of choice cells: $\rho.(E + F + G + H)$

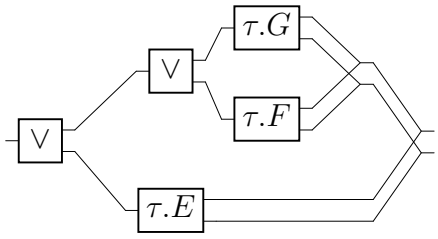
"+". This problem is also present in Foster and Kung's work; they simply fail to observe it.

In this section we start again from the τ design of section 7.2 and show that if we choose "choice" as the privileged operator, we arrive at a totally different design.

The first step is to decide on a standard "shape" for the cells. The translation for $\tau.(E + F + G)$ is

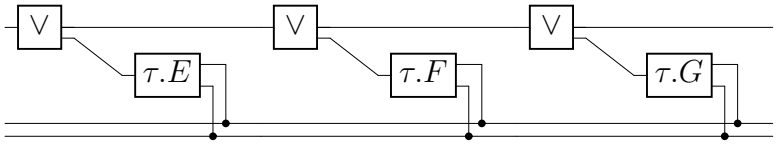
$$\dot{\vee} \circ \tau.E \triangle (\dot{\vee} \circ \tau.F \triangle \tau.G).$$

The corresponding picture is



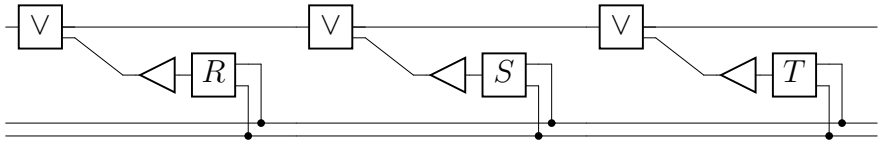
A way to make this structure modular is suggested by the following picture,

(7·7)

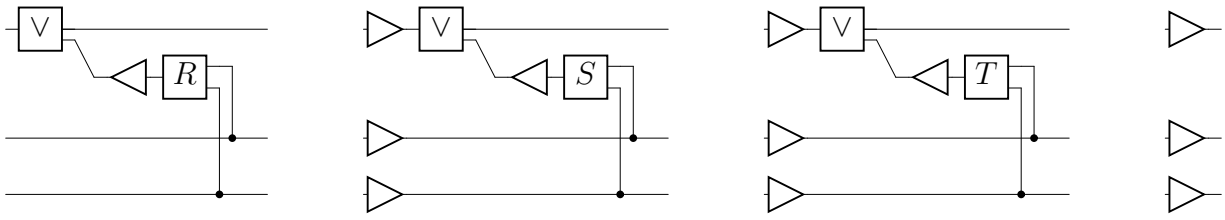


where we may see a composition of three cells of the same shape.

Note that the retiming transformation that we used in section 7.2 does not work here. In fact, even if we interpret the design (7·7) as having contraflow (the lower wire is not constrained, so we may decide that information flows from left to right on the lower wire), we are not able to design a systolic recognizer for expressions of the kind $t_0 + t_1 + \cdots + t_n$, where for all i, $t_i \in \mathcal{T}$. In fact, in that case we'd have a circuit of the following shape:



Applying slowdown and retiming, like we did in the section 7.2, would result in a design that cannot be implemented:



This circuit cannot be implemented because the antidelays in the upper wire are driven in the wrong direction.

This fact is not surprising: if we were able to design a recognizer for choice with zero latency time and a response time that does not depend on the number of choices, that would mean we are able to design an "or" gate with unbounded fan-in; and it is well known that no such component exists. For this reason we are forced to try a different design.

Formally, we define a cell for choice as

$$\kappa.E = cc.\tau.E,$$

where

$$cc.R = \iota\times\dot{\vee} \circ rsh \circ (\iota \vartriangle R)\times\iota.$$

It will be useful later to generalize the definition of cc to

$$cc_n.R = \iota\times\dot{\vee} \circ rsh \circ (\lhd^n \vartriangle R)\times\lhd^n.$$

Note that $cc.R = cc_0.R$. The relation $cc.R$ can be proved to be the least relation satisfying

$$(7.8) \qquad (a,b)\langle cc.R\rangle(a,c) \;\equiv\; \exists(d ::\; b = c\dot{\vee}d \;\wedge\; d\langle R\rangle a),$$

and from this we obtain

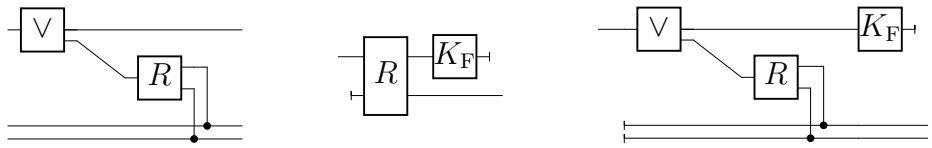
$$(7.9) \qquad (a,b)\langle \kappa.E\rangle(a,c) \;\equiv\; b = (\tau.E).a\,\dot{\vee}\,c.$$

Here wire a is actually a pair of wires (the ones we call a and e in section 7.0). It's easy to construct a left inverse of cc: we try to derive a function $ccli$ such that for all R,

$$(7.10) \qquad ccli.cc.R = R.$$

We calculate:

$$\begin{array}{ll} & a\langle ccli.cc.R\rangle b \\ = & \quad \{ \quad \bullet \quad \text{assume } ccli.X = S \circ X \circ T, \text{ for some } S \text{ and } T \quad \} \\ & a\langle S \circ cc.R \circ T\rangle b \\ = & \quad \{ \quad \text{calculus} \quad \} \\ & \exists(c,d,e ::\; a\langle S\rangle(c,d) \;\wedge\; (c,d)\langle cc.R\rangle(c,e) \;\wedge\; (c,e)\langle T\rangle b) \\ = & \quad \{ \quad (7.8) \quad \} \end{array}$$



$cc.R$ $\vartheta.R$ $\vartheta.cc.R$

Figure 7.6: cc and ϑ

$$
\begin{array}{ll}
 & \exists(c,e,f :: \ a\langle S\rangle(c, f \dot{\vee} e) \ \wedge \ f\langle R\rangle c \ \wedge \ (c,e)\langle T\rangle b) \\
= & \quad \{ \quad \bullet \quad \text{choose } S = \gg \quad \} \\
 & \exists(c,e,f :: \ a = f \dot{\vee} e \ \wedge \ f\langle R\rangle c \ \wedge \ (c,e)\langle T\rangle b) \\
= & \quad \{ \quad \bullet \quad \text{assume } (x,y)\langle T\rangle z \Rightarrow y = \mathit{false} \quad \} \\
 & \exists(c :: \ a\langle R\rangle c \ \wedge \ (c, \mathit{false})\langle T\rangle b) \\
= & \quad \{ \quad \bullet \quad \text{assume } (x,y)\langle T\rangle z \Rightarrow x = z \quad \} \\
 & a\langle R\rangle b.
\end{array}
$$

It is easily verified that the two assumptions on T can be satisfied by choosing

$$T = \iota \times K_F \circ \ll^{\cup}.$$

In summary, equation (7.10) holds by defining

$$\mathit{ccli}.R = \gg \circ \ R \circ \iota \times K_F \circ \ll^{\cup}.$$

Since we chose the cell shape κ in order to privilege choice, we expect to be able to express $\kappa.(E+F)$ in a compact way in terms of $\kappa.E$ and $\kappa.F$. In fact we have, for all a, b, c and d:

$$
\begin{array}{ll}
 & (a,b)\langle \kappa.(E+F)\rangle(a,c) \\
\equiv & \quad \{ \quad \text{interpretation of } \kappa\text{: (7.9)} \quad \} \\
 & b = (\tau.(E+F)).a \, \dot{\vee} \, c \\
\equiv & \quad \{ \quad \text{definition of } \tau \quad \} \\
 & b = (\dot{\vee} \ \circ \ \tau.E \ \vartriangle \ \tau.F).a \, \dot{\vee} \, c \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & b = (\tau.E).a \, \dot{\vee} \, (\tau.F).a \, \dot{\vee} \, c
\end{array}
$$

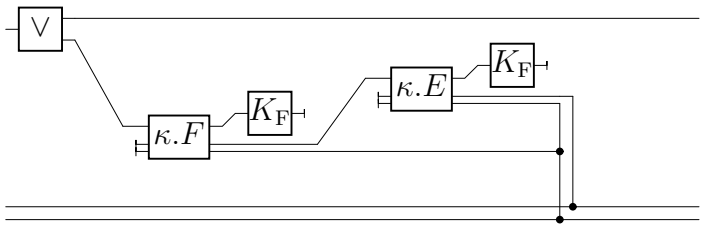


Figure 7.7: The cell for sequence

$$
\begin{array}{ll}
\equiv & \{ \quad \text{single-point rule} \quad \} \\
& \exists(d :: \ b = (\tau.E).a \mathbin{\dot\vee} d \ \ \wedge \ \ d = (\tau.F).a \mathbin{\dot\vee} c) \\
\equiv & \{ \quad (7.9) \quad \} \\
& \exists(d :: \ (a,b)\langle cc.\tau.E\rangle(a,d) \ \wedge \ (a,d)\langle cc.\tau.F\rangle(a,c)) \\
\equiv & \{ \quad \text{calculus} \quad \} \\
& (a,b)\langle \kappa.E \circ \kappa.F\rangle(a,c).
\end{array}
$$

Hence we have found, as expected, that

$$(7.11) \quad \kappa.(E+F) = \kappa.E \circ \kappa.F.$$

All we are left to do is to find a design for the cells for sequence, star and the character recognizer. We calculate:

$$
\begin{array}{ll}
& \kappa.E^* \\
\equiv & \{ \quad \text{definition} \quad \} \\
& cc.\tau.E^* \\
\equiv & \{ \quad \text{definition} \quad \} \\
& cc.(\dot\vee \ \circ \ \iota \times \tau.E \ \circ \ reorg)^\sigma \\
\equiv & \{ \quad (7.10) \quad \} \\
& cc.(\dot\vee \ \circ \ \iota \times ccli.\kappa.E \ \circ \ reorg)^\sigma.
\end{array}
$$

This calculation boils down to opening up definitions. We exploit (7.10) to make κ reappear, so that the transformation is applied recursively. In a similar way we obtain the cell for sequence:

$$(7.12) \quad \kappa.(E\,;F) = cc.(ccli.\kappa.F \ \circ \ll \triangle \ ccli.\kappa.E).$$

The character recognizer is obtained directly from the definition: for $t \in \mathcal{T}$,

$$\kappa.t = cc.\tau.t.$$

This concludes the first part of our task, that is finding a common "shape" for the implementation of all regular expression operators. The definitions so far are summarized below:

$$\begin{array}{lcll} \kappa.t & = & cc.\tau.t & \text{for all } t \in \mathcal{T} \\ \kappa.(E+F) & = & \kappa.E \circ \kappa.F & \\ \kappa.(E\,;F) & = & cc.(ccli.\kappa.F \circ \ll \triangle\ ccli.\kappa.E) & \\ \kappa.E^* & = & cc.(\dot{\vee} \circ \iota \times ccli.\kappa.E \circ reorg)^\sigma & \end{array}$$

and

$$\begin{array}{lcl} cc.R & = & \iota\times\dot{\vee} \circ rsh \circ (\iota \triangle R)\times\iota \\ cc_n.R & = & \iota\times\dot{\vee} \circ rsh \circ (\lhd^n \triangle R)\times\lhd^n \\ ccli.R & = & \gg \circ R \circ \iota\times K_F \circ \ll^\cup. \end{array}$$

* * *

We now apply pipelining to the "κ" design. As usual, we suppose we have implementations that correspond to $\kappa.E$ and $\kappa.F$, for arbitrary regular expressions E and F; we also suppose that some pipelining has been applied to them, so that they have arbitrary extra latency time. We model this by assuming our building blocks are $\lhd^n \circ \kappa.E$ and $\lhd^m \circ \kappa.F$, for some $n, m \geq 0$.

Let's begin with choice:

$$\begin{array}{ll} & \kappa.(E+F) \\ = & \quad \{ \quad (7.11) \quad \} \\ & \kappa.E \circ \kappa.F \\ = & \quad \{ \quad \text{retiming} \quad \} \\ & \rhd^{n+m+1} \circ (\lhd^n \circ \kappa.E) \circ \lhd \circ (\lhd^m \circ \kappa.F). \end{array}$$

This proves that given that R implements $\kappa.E$ with n extra latency, and S implements $\kappa.F$ with m extra latency, then $R \circ \lhd \circ S$ implements $\kappa.(E+F)$ with $n+m+1$ extra latency.

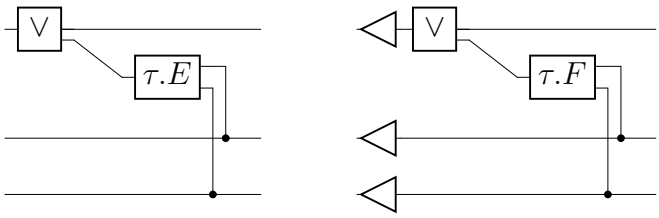


Figure 7.8: The pipelined cell for choice

We continue with sequence. Recall that, by the laws of delays,

(7.13) $\iota\times\triangleright = \triangleright \circ \triangleleft\times\iota$ and $\iota \mathbin{\triangle} \triangleright = \triangleright \circ \triangleleft \mathbin{\triangle} \iota$.

By retiming, we also obtain that for all R and $\diamond \in \{\triangleleft, \triangleright\}$,

(7.14) $\diamond \circ ccli.R = ccli.(\diamond \circ R)$.

We will need a lemma that allows us to "pull out" an antidelay from the scope of *cc*. We derive such a lemma below; the calculation is performed in very small steps to reduce the risk of errors:

$$
\begin{array}{ll}
 & cc.(\triangleright^n \circ R) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \iota\times\dot{\vee} \circ rsh \circ (\iota \mathbin{\triangle} (\triangleright^n \circ R))\times\iota \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & \iota\times\dot{\vee} \circ rsh \circ (\iota \mathbin{\triangle} \triangleright^n \circ \iota\times R)\times\iota \\
= & \quad \{ \quad (7.13) \quad \} \\
 & \iota\times\dot{\vee} \circ rsh \circ (\triangleright^n \circ \triangleleft^n \mathbin{\triangle} \iota \circ \iota\times R)\times\iota \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & \iota\times\dot{\vee} \circ rsh \circ (\triangleright^n \circ \triangleleft^n \mathbin{\triangle} R)\times\iota \\
= & \quad \{ \quad (7.13) \quad \} \\
 & \iota\times\dot{\vee} \circ rsh \circ \triangleright^n \circ (\triangleleft^n \mathbin{\triangle} R)\times\triangleleft^n \\
= & \quad \{ \quad \text{retiming} \quad \} \\
 & \triangleright^n \circ \iota\times\dot{\vee} \circ rsh \circ (\triangleleft^n \mathbin{\triangle} R)\times\triangleleft^n \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \triangleright^n \circ cc_n.R.
\end{array}
$$

Hence,

(7.15) $\quad cc.(\triangleright^n \circ R) = \triangleright^n \circ cc_n.R.$

Suppose now that R implements $\kappa.E$ with n extra latency, and S implements $\kappa.F$ with m extra latency:

$$R = \triangleleft^n \circ \kappa.E \quad \text{and} \quad S = \triangleleft^m \circ \kappa.F.$$

By the properties of delays, this is the same as

(7.16) $\quad \triangleright^n \circ R = \kappa.E \quad$ and $\quad \triangleright^m \circ S = \kappa.F.$

Given these preliminaries, we calculate for sequence:

$$\begin{array}{ll} & \kappa.(E\,;F) \\ = & \quad \{ \quad (7.12) \quad \} \\ & cc.(ccli.\kappa.F \circ \ll \vartriangle ccli.\kappa.E) \\ = & \quad \{ \quad (7.16) \text{ and } (7.14) \quad \} \\ & cc.(ccli.\kappa.F \circ \ll \vartriangle (\triangleright^n \circ ccli.R)) \\ = & \quad \{ \quad (7.13) \quad \} \\ & cc.(ccli.\kappa.F \circ \triangleright^n \circ (\triangleleft^n \circ \ll) \vartriangle ccli.R) \\ = & \quad \{ \quad (7.16), \text{ retiming} \quad \} \\ & cc.(\triangleright^{n+m} \circ ccli.S \circ (\triangleleft^n \circ \ll) \vartriangle ccli.R) \\ = & \quad \{ \quad (7.15) \quad \} \\ & \triangleright^{n+m} \circ cc_{n+m}.(ccli.S \circ (\triangleleft^n \circ \ll) \vartriangle ccli.R). \end{array}$$

We have thus shown that given that R implements $\kappa.E$ with n extra latency, and S implements $\kappa.F$ with m extra latency, then $cc_{n+m}.(ccli.S \circ (\triangleleft^n \circ \ll) \vartriangle ccli.R)$ implements $\kappa.(E\,;F)$ with $n+m$ extra latency. Unfortunately, it's not possible to apply the same reasoning to the cell for star. This is because it's not possible to take an anti-delay out of the scope of a feedback. The "closest thing" that holds is the following theorem:

$$(\triangleright \circ R)^\sigma = \triangleright \circ (R \circ \iota\times\triangleright)^\sigma.$$

The proof is:

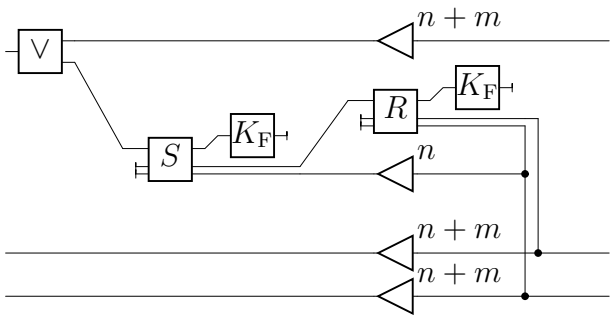


Figure 7.9: The pipelined cell for sequence

$$
\begin{array}{ll}
 & (\triangleright \circ R)^{\sigma} \\
= & \quad \{ \quad \text{loop-feedback} \quad \} \\
 & (\iota \mathbin{\triangle} \iota \circ \triangleright \circ R)^{\varpi} \\
= & \quad \{ \quad \text{delays, fusion} \quad \} \\
 & (\triangleright \times \triangleright \circ \iota \mathbin{\triangle} \iota \circ R)^{\varpi} \\
= & \quad \{ \quad \text{loop fusion} \quad \} \\
 & \triangleright \circ (\iota \times \triangleright \circ \iota \mathbin{\triangle} \iota \circ R)^{\varpi} \\
= & \quad \{ \quad \text{loop leapfrog} \quad \} \\
 & \triangleright \circ (\iota \mathbin{\triangle} \iota \circ R \circ \iota \times \triangleright)^{\varpi} \\
= & \quad \{ \quad \text{loop-feedback} \quad \} \\
 & \triangleright \circ (R \circ \iota \times \triangleright)^{\sigma}.
\end{array}
$$

We cannot isolate the antidelays out of a circuit with feedback; hence such a circuit cannot be implemented, given that we expect data to flow from the right to the left. For this reason, the pipelining transformation cannot be applied recursively to the circuit for star. Of course the circuit will still behave correctly; but we have to give up the full systolization of the recognizer for an expression that has a choice operator within the scope of a star.

Summarizing our results, we have a function that produces, for every regular expression, the extra latency of the pipelined recognizer:

$$
\begin{array}{lcll}
\ell.t & = & 0 & \text{for all } t \in \mathcal{T} \\
\ell.(E+F) & = & \ell.E \; + \; \ell.F \; + \; 1 & \\
\ell.(E\,;F) & = & \ell.E \; + \; \ell.F & \\
\ell.E^{*} & = & 0, &
\end{array}
$$

and a function that produces the pipelined recognizer itself:

$$
\begin{array}{lcl}
\eta.t & = & \kappa.t \qquad \text{for all } t \in \mathcal{T} \\
\eta.(E+F) & = & \eta.E \circ \lhd \circ \eta.F \\
\eta.(E\,;F) & = & cc_{n+m}.(ccli.\kappa.F \circ (\lhd^n \circ \ll) \vartriangle ccli.\kappa.E) \\
 & & \qquad \text{with } n,m = \ell.E, \ell.F \\
\eta.E^* & = & \kappa.E^*.
\end{array} \tag{7.17}
$$

7.4 Some considerations

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 the τ derivation is the 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.

Chapter 8

Simulation with the Ruby interpreter

To demonstrate that the designs presented in chapters 5, 6 and 7 are detailed enough to be implemented, we present here an example implementation for the Ruby interpreter written by Hutton, and documented in his thesis [23].

The implementation shown here is straightforward, and contains no new technical details; however, it bears some interest from a practical point of view.

The interpreter is written in an variant of ML (see e.g., [43]) called "Lazy ML" It can be found as of 1997 by ftp at `ftp.cs.chalmers.se`, in directory `/pub/misc/ruby`. The language Lazy ML has been superseded by Haskell; hence it is no longer widely available. Again, as of 1997 one can find it by ftp at `ftp.cs.nott.ac.uk`, directory `/haskell/chalmers/old`, or by looking for the files `lml-0.999.7.doc.tar.gz` and `lml-0.999.7.src.tar.gz` with Internet indexing services such as `ftpsearch`.

The next section gives a short introduction to the Ruby interpreter. After that, section 8.1 introduces a few basic definitions that make the interpreter's syntax more consistent with the rest of the thesis. In sections 8.2, 8.3 and 8.4, the τ, ρ and η designs from chapter 7 are implemented with the Ruby interpreter.

8.0 Introduction to the interpreter

8.0.0 The language

As is common in ML implementation of interpreters, there is no parser. The user program is not represented by text; one uses the underlying ML system to build data structures that have the correct type. In this case, the type of Ruby terms. This is particularly advantageous to us, since what we derived in the previous sections are functions from the datatype of regular expressions to the datatype of Ruby programs. These functions can be nicely implemented in ML.

There are of course a few differences between the mathematical presentation of the designs and their implementation. Mainly these differences have to do with the implementation of delays. While our calculus is essentially untyped, so that one can place a delay in front of an arbitrary wire, Hutton's interpreter distinguishes between integer-valued delays and boolean-valued delays. Moreover, delays to be placed on arbitrary collections of wires must be constructed with the product combinator.

While until now we pretended that a circuit's execution persists indefinitely in the future as well as in the past, in reality a circuit is switched on at some moment. Therefore, delay elements must either be initialized with some definite value, or left uninitialized (i.e., their value is undefined at time 0.) Hutton's interpreter chooses the former, thus forcing one to specify the initialization value of all delays.

A secondary difference is that the crisp mathematical notation must be replaced by the ascii-based syntax of ML programs. There is little we can do about this, short of writing a syntax-directed graphical editor for Ruby-generating programs. Furthermore, there are differences between our style of relation calculus and standard Ruby, mainly because in Ruby input is generally thought of as coming from the left; we embrace the opposite convention. For this reason we had to redefine the primitive components such as logic gates.

The composition operator is written "`..`"; for instance, where we usually write $R \circ S$, here we must write `R .. S`. The reason for this choice of symbol is that the symbol `.` (a single dot) is already defined in ML to stand for function composition.

The product is written "`!!`": instead of $R \times S$, we have `R !! S`. Again, the reason for this choice of symbol is that the symbol `||` has a predefined meaning in Lazy ML.

The converse of a circuit "R" is written "inv R". The identity circuit is "rid" (for "Ruby identity"). The repeated composition of a circuit R, that is R^n, is written "repeat n R". There is a special notation for defining wiring relations. For instance, the Ruby relation *swap* would be defined as

```
wiring (list [wire 1; wire2], list [wire 2; wire 1])
```

8.0.1 Using the interpreter

The rc command (Ruby Compiler) enters a circuit in the interpreter. For instance, after the command rc rid; the interpreter is ready to simulate the circuit rid (that is ι). To execute a simulation one must provide a sequence of input data, by means of the command rsim (Ruby Simulator). For instance, after entering the identity circuit with rc, the command rsim "0;2;4" produces as output

```
0 -  0 ~ 0
1 -  2 ~ 2
2 -  4 ~ 4
```

The first column contains the clock tick number; the second and third show the data that appear on the left and right sides of the circuit, respectively.

8.1 Preliminaries

The first thing to do is to load the Ruby simulator.

```
source "/home/matteo/rubysim/rubysim";
```

We continue with a number of definitions, aimed at bringing the syntax of what follows as close as possible to the mathematical definitions.

These are the projection on products, corresponding to $\ll$ and $\gg$. They are defined to be the inverse of the projection primitives offered by the interpreter. The reason is that in standard Ruby input is by convention thought of as coming from the left, while we embrace the opposite convention.

```
let outl = (inv p1);
let outr = (inv p2);
```

The split operator, $R \triangle S$:

```
let split r s = (r !! s) .. (inv fork);
```

The feedback operator, R^σ; it is built by means of

```
let rrr = wiring (list [list [wire 1; wire 2]; wire 2],
                  wire 1);
let feedback r = fork .. (r !! rid) .. rrr;
```

A few boolean gates:

```
let ANDG = (inv AND);
let ORG  = (inv OR);
```

The character comparator (the one written $(\dot{=} t)$ in the mathematical presentation)

```
let EQL t = (inv EQ) .. (first (icon t)) .. p2;
```

As discussed above, here are several delay definitions, one for each type of wire that we need: boolean, integer, and the (a, e) bus.

```
let bdelay   = (inv (bdel false));
let idelay   = (inv (idel 0));
let busdelay = idelay !! bdelay;
```

The following are the definitions of repeated delays: `bdelays n` corresponds to $\lhd^n$, on a boolean wire. The other definitions are for integer and "bus" delays.

```
let rec bdelays n   = repeat n bdelay;
let rec idelays n   = repeat n idelay;
let rec busdelays n = repeat n busdelay;
```

8.2 The τ design

The datatype of regular expressions. Since this Ruby interpreter does not support the type of "characters", we'll use integers in place of them.

```
let rec type E = char Int + choice E E + seq E E + star E;
```

From now to the end of this section, we will identify the characters t, u, v, z with the numbers 19,20,21,25 respectively (these are the ordinals of the letters t,u,v,z in the alphabet.)

```
let t = 19
and u = 20
and v = 21
and z = 25;
```

The wiring relation *reorg* is defined by

```
let reorg = wiring (list [wire 2; list [wire 1; wire 3]],
                    list [list [wire 1; wire 2]; wire 3]);
```

The definition of τ is now easily done, based on (7.1):

```
let rec tau (char t)     =    bdelay
                           .. ANDG
                           .. ((EQL t) !! rid)

||      tau (choice E F) =    ORG
                           .. (split (tau E) (tau F))

||      tau (seq E F)    =    (tau F)
                           .. (split (outl) (tau E))

||      tau (star E)     = (feedback (   ORG
                                      .. (rid !! (tau E))
                                      .. reorg  ))
;
```

We may now try a few examples. We begin with the examples in section 7.0. The regular expression is $t;t$. The following command enters $\tau.(t;t)$ in the interpreter:

```
rc (tau (seq (char t) (char t)));
```

The interpreter responds with:

```
Name      Domain             Range
---------------------------------
EQ        <0,w1>             w2
```

```
EQ        <0,w1>              w3
---------------------------------
AND       <w2,w4>             w5
AND       <w3,w6>             w7
---------------------------------
D_F       w5                  w8
D_F       w7                  w4
---------------------------------

Primitives   -  4
Delays       -  2
Longest path -  3
Parallelism  -  20%

Directions -  out ~ <in,in>

Wiring -  w8 ~ <w1,w6>

Inputs -  w1 w6
```

The output begins with a list of the "components" in the circuit, and lists their connections. Then some statistics are printed, and then —most importantly— the direction of data through the wires: the line `Directions - out ~ <in,in>` says that the circuit accepts pair of values on the right, and outputs single values on the left.

Let's submit some data: the input part of the first table in section 7.0 is submitted to the circuit with the command

```
rsim "19 F; 19 T; 19 F; 19 F; 19 T; 20 F; 19 F";
```

the number `19` here stands for character t, and `T`, `F` are the boolean *true* and *false*. The system responds with

```
 0 -  F ~ (19,F)
 1 -  F ~ (19,T)
 2 -  F ~ (19,F)
 3 -  T ~ (19,F)
 4 -  F ~ (19,T)
 5 -  F ~ (20,F)
 6 -  F ~ (19,F)
```

which is exactly the result we expected. The second table in section 7.0 is reproduced with the command

```
rsim "19 F; 19 T; 19 T; 19 F; 19 F; 19 F";
```

The output is

```
 0 -  F ~ (19,F)
 1 -  F ~ (19,T)
 2 -  F ~ (19,T)
 3 -  T ~ (19,F)
 4 -  T ~ (19,F)
 5 -  F ~ (19,F)
```

Aside. A more convenient way to feed data to the interpreter is by means of helper functions. It is not necessary to understand the definitions of `feed_data` and `feed_data1`, as long as the examples of their use are clear.

```
let feed_data [] = ""
||  feed_data (i.is) =
        show_int i @ " T"
      @ (conc (map (\x . "; " @ show_int x @ " F") is));
```

For instance, the output of

```
       feed_data [t;u;v];
```

is the string

```
       "19 T; 20 F; 21 F"
```

The variant `feed_data1` keeps the enable signal true for the first two clock ticks, instead of one.

```
let feed_data1 [] = ""
||  feed_data1 [i] = show_int i @ " T"
||  feed_data1 (i0.i1.is) =
        show_int i0 @ " T; " @ feed_data (i1.is);
```

The output of

```
feed_data1 [t;u;v];
```

is the string

```
"19 T; 20 T; 21 F"
```

End Aside.

Let's now try a more substantial example, to exercise all branches of the definition of τ. We want to recognize $(t+u)^*;(u+v)^*$:

```
rc (tau (seq (star (choice (char t) (char u)))
             (star (choice (char u) (char v)))));
```

Here is some test data:

```
rsim (feed_data [t;u;t;u;v;u;v;t;u;v]);

 0 -  T ~ (19,T)
 1 -  T ~ (20,F)
 2 -  T ~ (19,F)
 3 -  T ~ (20,F)
 4 -  T ~ (21,F)
 5 -  T ~ (20,F)
 6 -  T ~ (21,F)
 7 -  T ~ (19,F)
 8 -  F ~ (20,F)
 9 -  F ~ (21,F)
```

The circuit correctly recognizes that all prefixes of the given string, except the last three characters, belong to the target language. Note that we have given a single enable impulse, at time 0.

Suppose we were to ignore the prohibition on placing an expression containing the empty word inside the scope of a "star"; let's say we try to compile $\tau.(t^{**})$:

```
rc (tau (star (star (char t))));
```

The system responds with

```
  ERROR: unbroken loop in {OR,OR,D_F,AND}
```

Thus theory and implementation agree that such a circuit is not correct.

8.3 The ρ design

A few auxiliary definitions:

```
let plumb = wiring (   list [list [wire 1; wire 2];
                             list [wire 1; wire 3]]
                     ,  list [list [wire 1; wire 2];
                             wire 3]);

let str R =
    let ll = wiring (   wire 1
                      ,  list [wire 2;
                               list [wire 2; wire 1]])
    in  ll .. (rid !! R) .. (inv outl);
```

About *term*, the definition in section 3 cannot be accurately translated in the language accepted by the Ruby interpreter, since the $\top\top$ relation is not among its primitives.

Trying to cope with this problem, we first observe that in our programs, the value on the right side of a *term* is never "used"; in fact *term* either appears "at the outer level" in circuits of the form $R \circ term$, where we are only interested in the values on the left domain, or in subcircuits of the shape $str.(R \circ term)$, where by the definition of *str* the value on the right domain of *term* is ignored. Hence we are free to regard the right domain of *term* as an input or an output; the choice will not affect the interpretation of our circuits.

We exploit this freedom of choice by regarding the right domain of *term* as an output. Now, we'd like to be able to express that this output value is always chosen at random. But since Hutton's interpreter does not allow for nondeterministic circuits (for a precise definition of what it means to be deterministic for a Ruby program see Hutton's thesis [23]), we arbitrarily choose the value to be equal to the ones on the left domains. In other words, we replace $\iota \vartriangle \iota \circ \top\top$ by $\iota \vartriangle \iota$. This choice amounts to fixing one out of all possible behaviours; again, this choice will not affect the interpretation of the circuits.

```
let term = wiring (list [wire 1; wire 1], wire 1);
```

We can now represent ρ and υ (upsilon). The first clause of the definition of `rho` rewrites a three-places sequence so that the operators associate to the left. This is needed for the translation to be done correctly. Apart from this, the LML definition is very close to the original mathematical definition.

```
let rec rho (seq E (seq F G)) = rho (seq (seq E F) G)
||      rho (seq E F)         = (ups F) .. plumb .. (rho E)
||      rho E                 = (ups E) .. term

and rec ups (choice E F) =
            rid !! (   ORG
                    .. (split (str (rho E))
                              (str (rho F))  ))
||      ups (star E)     =
            rid !! (feedback (   ORG
                              .. (rid !! (str (rho E)))
                              .. reorg))
||      ups (char t)     =
               (inv busdelay)
            !! (bdelay .. ANDG .. ((EQL t) !! rid))
;
```

We may now again test a few examples. Let's start with a simple string pattern: $t; u; v$.

```
rc (rho (seq (seq (char t) (char u)) (char v)));
```

The output is:

```
   Name      Domain              Range
   ---------------------------------
   D_0       w1                  w2
   D_F       w3                  w4
   EQ        <21,w2>             w5
   D_0       w2                  w6
   D_F       w4                  w7
   EQ        <20,w6>             w8
   D_0       w6                  w9
   D_F       w7                  w10
   EQ        <19,w9>             w11
   ---------------------------------
   AND       <w5,w12>            w13
   AND       <w8,w14>            w15
   AND       <w11,w10>           w16
   ---------------------------------
   D_F       w13                 w17
   D_F       w15                 w12
```

```
D_F       w16                w14
----------------------------------

Primitives   -  6
Delays       -  9
Longest path -  3
Parallelism  -  28%

Directions -  <<in,in>,out> ~ <out,out>

Wiring -  <<w1,w3>,w17> ~ <w9,w10>

Inputs -  w1 w3
```

The "Directions" section shows that both input and output occur on the left side of the circuits — the right side should be be ignored, as discussed above.

Since ρ is a slowed circuit, the input must be interleaved with a second sequence (see [25]). For this execution, the second sequence is composed of "don't care" values.

```
rsim (feed_data [t;0;u;0;v;0;0;0]);
```

The output is:

```
 0 -  ((19,T),F) ~ (0,F)
 1 -  ((0,F),F) ~ (0,F)
 2 -  ((20,F),F) ~ (0,F)
 3 -  ((0,F),F) ~ (19,T)
 4 -  ((21,F),F) ~ (0,F)
 5 -  ((0,F),F) ~ (20,F)
 6 -  ((0,F),T) ~ (0,F)
 7 -  ((0,F),F) ~ (21,F)
```

As we said before, the values on the right domain should be ignored. The string is recognized at step 6, as the left domain is `((0,F),T)`, the output part being the second element of the pair.

To demonstrate that a slow circuit performs two separate, interleaved computations, we compile the recognizer for $(t; u; v) + (z; z)$. For clarity, we put the whole circuit inside the scope of `str`, so that input occurs on the right and output on the left side. This will make it easier to read the execution transcript.

```
rc (str (rho (choice (seq (seq (char t) (char u)) (char v))
                     (seq (char z) (char z)) )));
```

The tail of the output is:

```
   Primitives   -  11
   Delays       -  15
   Longest path -  3
   Parallelism  -  30%

   Directions -  out ~ <in,in>

   Wiring -  w3 ~ <w4,w6>

   Inputs -  w4 w6
```

We now demonstrate the circuit with an interleaving of the sequences $z; z; 0 \ldots$ and $t; u; v; 0 \ldots$:

```
rsim (feed_data1 [z;t;z;u;0;v;0;0]);
```

(Note that we've used `feed_data1` so that the enable input be asserted for the first two clock ticks.) The output is:

```
    0 -  F ~ (25,T)
    1 -  F ~ (19,T)
    2 -  F ~ (25,F)
    3 -  F ~ (20,F)
    4 -  T ~ (0,F)
    5 -  F ~ (21,F)
    6 -  F ~ (0,F)
    7 -  T ~ (0,F)
```

8.4 The η design

We define the "shape" of a κ cell:

```
let ccn n R =    (rid !! ORG)
             .. rsh
             .. ((split (busdelays n) R) !! rid);

let cc R    = ccn 0 R;
```

And now the left-inverse of `cc`:

```
let ccli R =    outr
             .. R
             .. second (bcon false)
             .. (inv outl);
```

The κ cell:

```
let rec kappa (char t)     = cc (tau (char t))
||      kappa (choice E F) = (kappa E) .. (kappa F)
||      kappa (seq E F)    =
              cc (   (ccli (kappa F))
                  .. (split outl
                            (ccli (kappa E))  ))
||      kappa (star E)     =
              cc (feedback (   ORG
                            .. (rid !! (ccli (kappa E)))
                            .. reorg))
;
```

Finally, here is the definition of the latency function ℓ, and of the pipelined recognizer, η.

```
let rec ell (char t)     = 0
||      ell (choice E F) = ell E + (ell F) + 1
||      ell (seq E F)    = ell E + (ell F)
||      ell (star E)     = 0
;

let rec eta (char t)     = kappa (char t)

||      eta (choice E F) =
               eta E
            .. (busdelay !! bdelay)
            .. (eta F)

||      eta (seq E F)    =
            let n = ell E
            and m = ell F
            in  ccn (n+m) (   (ccli (kappa F))
                           .. (split (idelays n .. outl)
```

```
                                        (ccli (kappa E))    ))

||      eta (star E)     = kappa (star E)
;
```

We compile now the pipelined recognizer for $t + (u; u) + (v; v; v)$.

```
let e = choice (char t)
               (choice (seq (char u) (char u))
                       (seq (seq (char v) (char v)) (char v)) );

rc (eta e);
```

The tail of the output is

```
   Primitives   -  21
   Delays       -  12
   Longest path -  4
   Parallelism  -  22%

   Directions -  <<out,out>,out> ~ <<in,in>,in>

   Wiring -  <<w4,w8>,w3> ~ <<w15,w16>,w31>

   Inputs -  w15 w16 w31
```

The circuit has the shape of a κ: as the "Directions" line shows, there are two more outputs and one more input than needed. The extra input should be always set to *true*, and the extra output are just copies of the (a, e) bus. The purpose of the *cc* left inverse, *ccli*, is to convert a κ-shape recognizer to the usual `Directions -  out ~ <in,in>` shape, taking care of these details. Therefore, we compile instead

```
rc (ccli (eta e));
```

and obtain, as expected,

```
   Primitives   -  21
   Delays       -  12
   Longest path -  4
   Parallelism  -  22%
```

```
Directions -  out ~ <in,in>

Wiring -  w3 ~ <w15,w16>

Inputs -  w15 w16
```

Since this is a pipelined recognizer, we should expect some extra latency. Executing

```
ell e;
```

we obtain 2. Hence the results will appear two clock ticks later than expected. We demonstrate this recognizer:

```
rsim (feed_data [v;v;v;0;0;0;0]);
```

produces, as expected,

```
0 -  F ~ (21,T)
1 -  F ~ (21,F)
2 -  F ~ (21,F)
3 -  F ~ (0,F)
4 -  F ~ (0,F)
5 -  T ~ (0,F)
6 -  F ~ (0,F)
```

and

```
rsim (feed_data [u;u] @ ";" @ (feed_data [t;0;0;0]));
```

produces

```
0 -  F ~ (20,T)
1 -  F ~ (20,F)
2 -  F ~ (19,T)
3 -  F ~ (0,F)
4 -  T ~ (0,F)
5 -  T ~ (0,F)
```

(Note that in step 2, the tick after the pattern $u;u$ is input, we input the pattern t and we assert again the enable signal. This results in the pattern t being recognized, while the output of the previous elaboration is still in the pipeline. the `T` result in step 4 signals that the pattern $u;u$ was recognized; the `T` result in step 5 is for the t pattern.)

Let's now demonstrate our claims about combinatorial path length. We claimed that our η design results in circuits such that the length of longest combinational path does not increase as the number of "+" operators in the regular expression increases. According to the compiler output, the longest path is 4 units long. We now add more choice branches to the previous regular expression:

```
let e1 = (choice e (choice (char z)
                           (choice (char u) (char v))));
```

thus `e1` corresponds to the regular expression $t+(u;u)+(v;v;v)+z+u+v$.

```
rc (ccli (eta e1));
```

the interpreter responds with

```
Primitives   -  30
Delays       -  24
Longest path -  4
Parallelism  -  23%

Directions -  out ~ <in,in>

Wiring -  w3 ~ <w38,w39>

Inputs -  w38 w39
```

The price for this is an increased latency: in fact, the latency of this circuit is `ell e1` $= 5$. On the other hand, increasing the number of ";" operators may increase the longest path length: let `e2` $= (t+(u;u)+(v;v;v));t;t$

```
let e2 = seq e (seq (char t) (char t));
```

then

```
rc (ccli (eta e2));
```

results in

```
Primitives   -  29
Delays       -  14
Longest path -  6
Parallelism  -  14%

Directions -  out ~ <in,in>

Wiring -  w40 ~ <w5,w6>

Inputs -  w5 w6
```

Hence the longest combinational path was 4 for `e`, but increases to 6 for `e1`.

8.5 Conclusions

We have demonstrated how the regular language recognizers from chapter 7 can be straightforwardly implemented using an existing Ruby interpreter. While no new technical details are shown, the implementation has some interest from a practical point of view, and it demonstrates that the circuits that we derived are indeed implementable.

The Ruby interpreter does not have a parser, so that ML terms representing circuits must be directly constructed by the user. This fact is not an hindrance at all; in fact it greatly helps since it makes it very easy to write Ruby-generating programs. Our designs (7.1), (7.6) and (7.17) are actually functional programs that transform regular expressions into circuits, so the translation from the mathematical notation to the interpreter's syntax was very simple and direct. Although there are surface differences, as all ML programs are ASCII strings in the end, the structure of the Ruby interpreter implementations of `tau`, `rho` and `eta` is very close to the structure of the corresponding equations.

Chapter 9

A Tangram implementation

We present a simple compiler for the language of Ruby terms into Tangram circuits. Tangram is a language for the description of asynchronous circuits that was devised by Van Berkel [51]. Tangram programs can be compiled to asynchronous circuits by means of a compiler developed under van Berkel's supervision at the Philips research laboratories in Eindhoven. The Tangram compiler `tg2hc` translates Tangram programs into an intermediate notation called *handshake circuits*, which can then be interpreted (i.e., simulated) or compiled to silicon. Tangram as a programming language is derived from Hoare's CSP [21]. A similar language is Handel, developed by Ian Page [41]; the main difference is that Handel is compiled to synchronous circuits.

Our compiler performs a syntax-directed translation from Ruby circuits to Tangram. Part of it is derived from Hutton's interpreter [23]; see chapter 8. The compiler is written in Gofer[0], a functional programming language very similar to Haskell, devised and implemented by Mark P. Jones.

It must be noted that although Tangram is implemented in asynchronous circuits, the Tangram programs we produce are still essentially synchronous. So they do not take advantage of many of the features that Tangram offers. We use Tangram as a means of showing that our designs are actually implementable; but we do not claim that the Tangram programs we produce are the best possible Tangram implementation for the task at hand.

As is usual for this kind of experimental compiler, there is no parser. Ruby terms are just a defined datatype, and the user is expected to construct the element of this datatype that represents the Ruby term he wants to compile. Besides eliminating the problem of inventing a concrete syntax and writing

[0] `ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer`

a parser, this approach has the advantage to make it easy to write programs that write programs, such as the definitions of τ, ρ and η in section 9.2.

The compilation is better described as a sequence of phases. In the first phase the Ruby term is translated to a datastructure we call "Network" along with Hutton [23]. This first phase is a modification of the code Hutton presented in his thesis.

In the second phase, the network is checked against a number of requirements to see that it is implementable. The requirements are:

- All wires should be assigned a type.
- All wires should be driven by at most 1 gate. Wires that are led by no gate are inputs, and must appear in the list of external symbols.

(Ideally, we should also check that there are no cycles that are not interrupted by delays, but presently the compiler does not enforce this. Simulating a program with such a cycle, such as the Tangram program produced for $\dot{\wedge}^{\sigma}$, results in an error from the Tangram tools.) There is a further restriction that our compiler enforces, that is that all output wires must be outputs of delays. It would be possible to modify the compiler to remove this restriction, but it would add complexity to the compiler code. If the above requirements are met, then compilation proceeds to the third phase.

In the third phase, the network is translated to a string of characters which happens to be a Tangram program.

Once we have obtained a Tangram program, we can simulate its operation with the tangram simulator `hcsim`. We will try the same examples we tried with the Ruby interpreter in chapter 8. We will use the definitions of τ and ρ, from that section, (hand-)translated from Lazy ML to Gofer.

By running our examples with the Tangram simulator we have the guarantee that our examples could be compiled to silicon, should we want to. In addition, the simulator provides a host of data about the circuit's performance in terms of area, speed, and energy consumption.

As an example of what the compiled Tangram programs look like, consider the translation of $(\lhd \circ \dot{\wedge}) \vartriangle (\lhd \circ \dot{\vee})$ in figure 9 on the next page All our Tangram programs have this structure: they communicate with the external environment through channels; there is one channel for each wire entering or exiting the circuit. The body of the program consists of a "forever" loop that executes two commands in sequence. The first command performs communication on all the channels; the second updates the contents of all local

```
(c1!bool & c6!bool & c2?bool & c3?bool).
begin
    v1: var bool
  & v2: var bool
  & v3: var bool
  & v6: var bool
|
  forever do
      c1!v1 || c6!v6 || c2?v2 || c3?v3
    ; <<v1,v6>>
      := begin
             w0 = val v2 * v3
           & w5 = val v2 + v3
         | <<w0,w5>>
         end
  od
end
```

Figure 9.0: A Tangram program example

variables, except for those that are used to hold the inputs (`v2` and `v3` in the above example).

9.0 Tangram

A simple introduction to Tangram is given here, where we only mention the language features that we use in this thesis. For a complete description of the language, see Van Berkel [51] or Schalij [48].

A Tangram program is composed of an "external declaration", followed by a command. The external declaration is a list of the channels that the program uses to communicate with the environment, together with their types. For instance

```
(a?bool & b![0..127])
```

is an external declaration of two channels, one of which (`a`) is an input channel of type boolean; (this means that the external environment writes data on

it.) The other channel, `b`, is an output channel of type `[0..127]`, that is a channel that the environment is supposed to read, and carries values of type integer between 0 and 127, included.

Boolean and integer subranges are the only scalar types available in Tangram. Among the structured types, the only one we use here is the tuple, written

```
<<E0, E1, E2, ..., En>>
```

The simplest kind of command is the assignment command, which has the ordinary Pascal syntax and meaning. Communication commands have the shape `a?x` or `b!E`, where `a` is an input channel, `x` is a variable, `b` is an output channel and `E` is an expression. The effect of a communication command is to communicate a value through a channel. The environment must "collaborate" for a communication to occur; no communication occurs unless the other party is ready to participate in it. This is the usual meaning of the communication actions in CSP.

Commands can be composed in sequence or in parallel; if `C0` and `C1` are commands, then `C0 ; C1` is the sequential composition, while `C0 || C1` is the parallel composition of the two commands.

The `forever do` command implements unbounded repetition. The syntax for the unbounded repetition of command `C` is

```
forever do C od
```

The block command permits the introduction of local declarations. The syntax is

```
begin D | C end
```

where `D` is a sequence of declarations, and `C` is a command. In our circuits, the block command will only be used for local variables, although other kinds of declarations are allowed.

The syntax for expressions is standard, except that the operators `*`, `+` and unary `-` are overloaded. When applied to integers, they have their usual meaning of multiplication, addition, and negation respectively. When applied to booleans, they mean conjunction, disjunction and logic negation. It should be noted that integer multiplication is only allowed if one of the two operands is a constant.

A particular kind of expression is the block expression, which is used to declare local names within an expression: for `D` a list of declarations and `E` an expression,

```
begin D | E end
```

The main purpose of block expression is to factor common subexpressions, in order to save area. This is done by means of a particular declaration called *value declaration*, that is only allowed in block expressions. For instance, the expression

```
f(x+y, x+y)
```

has the same value as

```
begin a = val x+y | f(a, a) end,
```

except that for the latter a single adder is generated, instead of two.

9.1 The compiler in detail

9.1.0 Type declarations

We begin the description of the compiler with the type declarations.

```
data Tree a = Nil | LEAF a | PAIR (Tree a) (Tree a)

type Wiretree = Tree Wire

type Wire = Int
```

Trees are defined in the usual way. A tree of wires, or `Wiretree`, models what we call "arbitrary collection of wires" in section 3. We identify each wire with a number.

```
data Ruby = PRIM Prim
          | WIRING (Wiretree, Wiretree)
          | SEQ Ruby Ruby
          | PAR Ruby Ruby
          | INV Ruby
```

```
data Prim = AND | OR | NOT | EQL | CONST TValue
          | DELAY | NOP | TYPE TType | TOP

data TType = TBool | TRange Int

data TValue = TFalse | TTrue | TInt Int
```

A Ruby program is described by cases: it is either a primitive, or a wiring relation, or the sequence, parallel composition or inverse of other Ruby programs.

Recall that a wiring relation captures all components that only perform a rearrangement of wires, such as, for instance ι, *fork*, or *rsh*. Wirings are represented by a pair of wiretrees. Essentially, a wiring represents a set of equations on wires. For instance, $fork_2$ is represented by

```
WIRING (PAIR (LEAF 0) (LEAF 0), LEAF 0).
```

In actual use the numbers that appear in the definition of a wiring are consistently substituted for the wire numbers of the circuits that the wiring is connected to.

The primitives comprise all the standard combinational components such as $\dot{\wedge}$, $\dot{\vee}$ and $\doteq$, the constant circuit K_n, the delay $\lhd$, and top $\top\!\!\top$. The `TYPE` primitive is special: it is used to introduce type constraints on the wires. When we did our calculations in the preceding chapters we could reason about character-valued streams without specifying what the set of characters was. But in order to obtain a Tangram implementation, we must assign each wire a type. The only scalar types supported by Tangram are the booleans, and (finite) subranges of the integers. All wires connected to boolean primitives like $\dot{\wedge}$ are automatically assigned boolean type. But the inputs to the $\doteq$ primitive could be of any subrange type. The natural way to assign a constraint in relation algebra is by composition with a monotype. Therefore we introduce the primitive `TYPE` to be the monotype such that

$$m\langle\texttt{TYPE (TRange n)}\rangle m \;\equiv\; 0 \leq m < 2^n$$

and

$$m\langle\texttt{TYPE TBool}\rangle m \;\equiv\; m = \texttt{TFalse} \vee m = \texttt{TTrue}$$

(The `T` prefix in `TBool` and other identifiers stands for "Tangram" and is meant to avoid name clashes with Gofer standard types.) A `Node` is a triple

that associates a primitive to a left and right domain, represented by two wiretrees.

```
type Node = (Wiretree , Prim , Wiretree)
```

A `Network` is a triple where the first element is a list of nodes, and the second and third element are the left and right domain of the network. The domains are wiretrees rather than simply lists of wires because the shape information is needed when two networks are to be combined to implement a sequential composition. In that case it is necessary to check that the domains of the two networks are compatible, i.e., that they have the same shape.

```
type Network = ([Node], Wiretree, Wiretree)
```

9.1.1 The main program

The structure of the compiler is evident in its top function, `r2t` (from "Ruby to Tangram").

```
r2t :: Ruby -> String
r2t prog =
  let (net,_) = translate (prog, 0)
      types   = type_assign net
      clnet   = clean net
      ws      = wires clnet
      und     = undefs ws types
      od      = over_driven clnet
      ud      = under_driven clnet
      err str = str ++ "\n" ++ pp_network clnet
  in if not (null und)
     then err ("*** Error: wires " ++ show und ++ " have no type constraint")
     else if not (null od)
     then err ("*** Error: wires " ++ show od ++ " driven more than once")
     else if not (null ud)
     then err ("*** Error: wires " ++ show ud ++ " internal and not driven")
     else tgskel (net2tangram clnet types)
```

The compiler returns a string of characters, which contains a Tangram program in case the compilation was successful, or an error message otherwise. Compilation of a Ruby program consists of first translating it into a network with `translate`; then types are assigned to all wires with `type_assign`; hence the network is "cleaned" by removing all nodes of type `NOP` (no operation)

or `TYPE` (type constraint), which are not needed in subsequent phases. Then a number of lists of wires are built: `ws` is the list of all wires, `und` is the list of wires that were not assigned a type. List `od` contains the wires that are driven more than once; this list would be non-empty when trying to compile, for instance, $\dot{\wedge}^{\cup} \circ \dot{\wedge}$ (since the outputs of the two $\dot{\wedge}$ components are connected together). List `ud` contains the wires that are not driven, and do not occur in the external interface of the network. If all of the three lists `und`, `od` and `ud` are empty, then compilation proceeds with the call to `net2tangram`, which produces a compact representation of the Tangram program, to be then expanded by `tgskel` into a legal Tangram program.

9.1.2 From Ruby to networks

The first step is the translation from Ruby programs to networks. This is done by means of a modification of Hutton's Ruby interpreter [23]. For instance, the translation of a simple program consisting of a single "$\dot{\wedge}$" gate is:

```
? translate (PRIM AND, 0)
(([(LEAF 2,AND,PAIR (LEAF 0) (LEAF 1))],LEAF 2,PAIR (LEAF 0) (LEAF 1)),3)
```

(Here and in the following information typed by the user is right after the `?` prompt, while the text in the lines below are output by the gofer system.) The second argument given to `translate`, a `0` in the example above, is needed to generate wire "names": we associate a unique number to each wire. Function `translate` returns the network together with a number, which by the definition of `translate` is "fresh", i.e. it is not associated to any wire in the network that is returned.

The above example is hard to read, and the "fresh wire" numbers are cumbersome for interactive use; for this reason, in the following examples the shortcut function `ppn` (short for Pretty Print Network) will be used:

```
? ppn (PRIM AND)
[
  (w2, AND, (w0, w1))
]
w2 ~ (w0, w1)
```

Function `translate` is defined by cases. For combinational gates, all cases share a similar structure. For instance, the definition for $\dot{\wedge}$ is:

```
translate :: (Ruby , Wire) -> (Network , Wire)
translate (PRIM AND, x) =
  let rdom = PAIR (LEAF x) (LEAF (x+1))
      ldom = LEAF (x+2)
      node = (ldom, PRIM AND, rdom)
  in  ( ([node], ldom, rdom), x+3 )
```

The definition for product is recursive; it is very simple, and the only point to note is that care must be taken to handle correctly the fresh wire numbers.

```
translate (PAR p0 p1, x) =
  let ((l0, ld0, rd0), y) = translate (p0, x)
      ((l1, ld1, rd1), z) = translate (p1, y)
      rdom = PAIR rd0 rd1
      ldom = PAIR ld0 ld1
      nodes = l0 ++ l1
  in  ( (nodes, ldom, rdom), z )
```

The definition for sequential composition is more interesting, since the right domain of the first network and the left domain of the second network must be unified. The unification algorithms that we employ is rather simplistic, since it fails whenever two wiretrees differ in shape (unless one of the two is `Nil`). Unification, when successful, returns an Eqnset, which is defined as a list of pair of wires. This is interpreted as a set of equations on wires.

```
unify :: Wiretree -> Wiretree -> Eqnset
unify t0 t1 = unify' t0 t1 empty_eqnset
  where unify' Nil _ s = s
        unify' _ Nil s = s
        unify' (LEAF x0) (LEAF x1) s =
          if x0 == x1 then s
          else let t = (x0,x1):s
               in unify' (apply t (LEAF x0)) (apply t (LEAF x1)) t
        unify' (PAIR v0 v1) (PAIR w0 w1) s =
          let t = unify' v0 w0 s
          in unify' v1 w1 t
        unify' x y _ = match_error x y
```

"Applying" an Eqnset e to a wiretree t means to substitute for every wire w occurring in t a canonical element of the equivalence class of w with respect to e.

```
apply :: Eqnset -> Wiretree -> Wiretree
apply s = tree_map f
  where f w = head (equiv_class w s)
```

The unify procedure has the property that, when applied to a pair of non-`Nil` wiretrees, it produces an Eqnset that when applied to either wiretrees produces the same result.

Having introduced unification, the translation of sequential composition can be defined as:

```
translate (SEQ p0 p1, x) =
  let ((l0, ld0, rd0), y) = translate (p0, x)
      ((l1, ld1, rd1), z) = translate (p1, y)
      s                   = unify rd0 ld1
      ldom                = apply s ld0
      rdom                = apply s rd1
      nodes               = map (apply_node s) (l0 ++ l1)
  in  ( (nodes, ldom, rdom), z )
```

A point to note is that this definition assumes `unify` always succeeds; failure of unification is reported via Gofer's exception reporting mechanism, function `error`.

For instance, $\dot{\wedge} \circ \dot{\wedge}$ is not a well-formed program, because the right domain of $\dot{\wedge}$ is a pair while the left domain is not:

```
? ppn (SEQ (PRIM AND) (PRIM AND))
[
  (w
Program error: *** Wire structures incompatible: (w0, w1) and w5
```

(The partial output before the error message is due to Gofer's lazy evaluation: some of the output is available before the expression is fully evaluated.) Conversely, $\dot{\neg} \circ \dot{\wedge}$ can be nicely translated to a network:

```
? ppn (SEQ (PRIM NOT) (PRIM AND))
[
  (w1, NOT, w4)
  (w4, AND, (w2, w3))
]
w1 ~ (w2, w3)
```

Note how wire `w4` is both on the right domain of the $\dot{\neg}$ and on the left domain of the $\dot{\wedge}$, thanks to unification.

The `CONST` primitive is translated to a network by

```
translate (PRIM (CONST n), x) =
  let ldom  = (LEAF x)
      rdom  = Nil
      node  = (ldom, CONST n, rdom)
  in  ( ([node], ldom, rdom), x+1 )
```

The point to note here is that the right domain is `Nil`. This allows to connect any wire shape to the right of a constant component. The `TOP` primitive is translated similarly:

```
translate (PRIM TOP, x) = ( ([(Nil, NOP, Nil)], Nil, Nil), x )
```

9.1.3 Syntactic sugar

Typing Ruby programs in terms of the type constructors `SEQ` and `PAR` is tedious, and the result is hard to read. For this reason some "syntactic sugar" may help:

```
x r s = PAR r s
o r s = SEQ r s
andg  = PRIM AND
org   = PRIM OR
notg  = PRIM NOT
eqlg  = PRIM EQL
delay = PRIM DELAY
width n  = PRIM (TYPE (TRange n))
pair n m = PAIR (LEAF n) (LEAF m)
false = PRIM (CONST TFalse)
true  = PRIM (CONST TTrue)
k n   = PRIM (CONST (TInt n))
```

With these definitions, term $R{\times}S \circ T{\times}U$ can be written more naturally as `r ‘x‘ s ‘o‘ t ‘x‘ u`.

9.1.4 Assigning type information

For a network to be implementable in Tangram, it is necessary to assign a type to all wires. Some nodes always carry their own type information; for instance, an `AND` node can only be connected to boolean wires. Other primitives have more flexible type constraints; notably the `EQL` primitive where the output must be boolean, but the inputs could have any type as

long as both have the same type. Type assignation is then a matter of propagating the constraints to all wires.

Function `type_assign` has type

```
type_assign :: Network -> Map Wire TType
```

where type `Map` is a mapping defined, for arbitrary types `a` and `b`, by

```
type Map a b = [(a,b)]
```

(that is, a mapping is represented by a list of pairs). The definition of `type_assign` is then

```
type_assign net =
  fixpoint (type_assign' net) empty_map
  where  type_assign' ([],_,_) map = map
         type_assign' (node:rest,l,r) map =
           type_assign' (rest,l,r) (type_assign_node node map)
```

Here `fixpoint f x` $= \lim_{n\to\infty}$ `f`^{n} `x`. The auxiliary function `type_assign_node` extends a map with the type information provided by a single node; it is defined by cases over the primitives.

```
type_assign_node :: Node -> Map Wire TType -> Map Wire TType
type_assign_node node map =
  case node of
    (LEAF l, AND, PAIR (LEAF r0) (LEAF r1)) ->
       extend_many [l,r0,r1] TBool map
    (LEAF l, OR, PAIR (LEAF r0) (LEAF r1)) ->
       extend_many [l,r0,r1] TBool map
    (LEAF l, NOT, LEAF r) ->
       extend_many [l,r] TBool map
    (LEAF l, EQL, PAIR (LEAF r0) (LEAF r1)) ->
       let map' = extend map (l, TBool)
       in if defined r0 map'
          then extend map' (r1,(value r0 map))
          else if defined r1 map'
          then extend map' (r0,(value r1 map))
          else map'
    (LEAF l, DELAY, LEAF r) ->
       if defined l map
       then extend map (r, (value l map))
       else if defined r map
```

```
      then extend map (l, (value r map))
      else map
  (_, NOP, _) -> map
  (_, TOP, _) -> map
  (LEAF l, CONST TFalse, Nil) -> extend map (l, TBool)
  (LEAF l, CONST TTrue, Nil) -> extend map (l, TBool)
  (LEAF l, CONST _, Nil) -> map
  (LEAF l, TYPE t, LEAF r) -> extend_many [l,r] t map
```

Functions `extend` and `extend_many` are used to update a map. They do not allow redefinition of a value, i.e. a map can only be extended in a point where it is not defined, unless the extension leaves the map unchanged. The error message is over-specific; we are relying on the fact that `extend` is only used in the context of assigning type constraints to wires in a network.

```
extend :: (Eq a, Eq b) => Map a b -> (a,b) -> Map a b
extend m (a,b) =
  if not (defined a m)
  then (a,b):m
  else if value a m == b then m
  else error "Type constraints incompatible"
```

The error message would be generated, for instance, when trying to compile `eqlg ‘o‘ (width 1) ‘x‘ (width 2)`. Function `extend_many` is a shortcut: it extends a map `m` with the pair `(x,b)` for all `x` in the given list.

```
extend_many :: (Eq a, Eq b) => [a] -> b -> Map a b -> Map a b
extend_many [] b m = m
extend_many (a:as) b m = extend_many as b (m ‘extend‘ (a,b))
```

9.1.5 Code generation

The Tangram programs we generate are all instances of the following schema:

```
(𝒪 & ℐ).
begin
  𝒟
| forever do
      ℐ𝒪
    ; ℒ := begin 𝒱 | ℛ end
  od
end
```

where $\mathcal{I}$ and $\mathcal{O}$ are lists of input and output channel declarations, respectively; $\mathcal{D}$ is a list of variable declarations; $\mathcal{IO}$ is a parallel composition of communication statements; $\mathcal{L}$ is a tuple of variables, $\mathcal{V}$ is a list of value declarations, and $\mathcal{R}$ is a tuple of value names.

An instance of this schema is represented in a compact way by a sixtuple (i, o, l, a, n, t), where i is a list of input wires, o is a list of output wires, l is a list of wires that have a corresponding Tangram variable, a is a list of pair of wires that is used to generate the tuples $\mathcal{L}$ and $\mathcal{R}$ above, n is a list of nodes, and t is a map that assigns a type to each wire. Such tuples are represented in Gofer by the type `TangramP`:

```
type TangramP =
   ([Wire], [Wire], [Wire], [(Wire, Wire)], [Node], Map Wire TType)
```

Function `net2tangram` has type

```
net2tangram :: Network -> Map Wire TType -> TangramP
```

and is defined by

```
net2tangram net types =
  let (nodes, ldom, rdom) = net
      in_wires  = inputs net
      out_wires = outputs net
      dels      = sort (filter is_delay_node nodes)
      del_outs  = map (\(LEAF w, _, _) -> w) dels
      var_wires = sort (del_outs ++ in_wires)
      val_nodes = sort (nodes \\ dels)
      assign    = map (\(LEAF wl, _, LEAF wr) -> (wl, wr)) dels
  in (in_wires, out_wires, var_wires, assign, val_nodes, types)
```

Finally, function `tgskel` transforms a TangramP sixtuple into a full Tangram program:

```
tgskel :: TangramP -> String
tgskel tp =
        (write_header tp) ++ "."
     ++ "\nbegin"
     ++ "\n    " ++ (write_vardecls tp)
     ++ "\n|"
     ++ "\n  forever do"
     ++ "\n      " ++ (write_iocmds tp)
```

```
    ++ "\n    ; " ++ (write_lhs tp)
    ++ "\n      := begin "
    ++ "\n             " ++ (write_valdefs tp)
    ++ "\n         | " ++ (write_rhs tp)
    ++ "\n         end"
    ++ "\n  od"
    ++ "\nend"
```

In the above definition a number of auxiliary functions are used to generate the concrete syntax for certain Tangram syntactic categories. Here is the function that produces the external declaration part:

```
write_header :: TangramP -> String
write_header (inputs, outputs, _,_,_, types) =
  let in_decls          = map (\x -> input_decl x (value x types)) inputs
      out_decls         = map (\x -> output_decl x (value x types)) outputs
      input_decl w typ  = chan_name w ++ "?" ++ show typ
      output_decl w typ = chan_name w ++ "!" ++ show typ
  in
      "(" ++ (cat_with " & " (out_decls ++ in_decls)) ++ ")"
```

The following produces the list of variable declarations:

```
write_vardecls :: TangramP -> String
write_vardecls (_, _, local_vars, _, _, types) =
  let var_decls = map (\x -> var_decl x (value x types)) local_vars
  in (cat_with "\n  & " var_decls)
```

This function returns the parallel composition of the communication commands:

```
write_iocmds :: TangramP -> String
write_iocmds (inputs, outputs, local_vars, _, _, _) =
  let in_cmds  = map f inputs
      out_cmds = map g outputs
      f x      = chan_name x ++ "?" ++ var_name x
      g x      = if x `notElem` local_vars
                 then error "*** Output wires must be driven by delays"
                 else chan_name x ++ "!" ++ var_name x
  in (cat_with " || " (out_cmds ++ in_cmds))
```

The following functions produce the strings that take the roles of $\mathcal{V}$, $\mathcal{L}$ and $\mathcal{R}$ in the above Tangram program schema, respectively.

```
write_valdefs :: TangramP -> String
write_valdefs (inputs, _, local_vars, _, nodes, _) =
  cat_with "\n           & " (map (val_def local_vars) nodes)

write_lhs :: TangramP -> String
write_lhs (_, _, _, assign, _, _) =
  let left_vals = map fst assign
  in "<<" ++ (cat_with "," (map var_name left_vals)) ++ ">>"

write_rhs :: TangramP -> String
write_rhs (_, _, local_vars, assign, _, _) =
  let right_vals = map snd assign
      f x = if x `elem` local_vars
            then var_name x
            else wire_name x
  in "<<" ++ (cat_with "," (map f right_vals)) ++ ">>"
```

This concludes the exposition of the program code of the Ruby to Tangram compiler.

9.2 Regular language recognizers

9.2.0 The τ design

Before we can compile the recognizers in Tangram, we need a small library of standard Ruby components:

```
rid   = WIRING (LEAF 0, LEAF 0)
rid2  = WIRING (pair 0 1, pair 0 1)
swap  = WIRING (pair 0 1, pair 1 0)
fork  = WIRING (pair 0 0, LEAF 0)
fork2 = WIRING (PAIR (pair 0 1) (pair 0 1), pair 0 1)
rsh   = WIRING (PAIR (LEAF 0) (pair 1 2), PAIR (pair 0 1) (LEAF 2))
lsh   = INV rsh
rsh2  = WIRING (  PAIR (pair 0 1) (pair 2 3)
               ,  PAIR (PAIR (pair 0 1) (LEAF 2)) (LEAF 3)  )
lsh2  = INV rsh2
outl  = WIRING (LEAF 0, pair 0 1)
outr  = WIRING (LEAF 1, pair 0 1)

split  r s = r `x` s  `o`  fork
```

```
split2 r s = r `x` s  `o`  fork2

feedback r = (INV fork) `o` (r `x` rid) `o` rrr
  where rrr = WIRING (PAIR (pair 1 2) (LEAF 2), LEAF 1)

feedback2 r = (INV fork) `o` (r `x` rid) `o` rrr2
  where rrr2 = WIRING (PAIR (PAIR (pair 0 1) (LEAF 2)) (LEAF 2),  pair 0 1)
```

Because of the simple unification algorithm that we employ, we cannot connect a single wire to a pair of wires. In other words, our `rid` primitive corresponds to $\bar{\iota}$ rather than ι. This means that we must define different versions of various wiring primitives: for instance, `rid2` corresponds to $\bar{\iota}\times\bar{\iota}$.

The type of regular expression is defined this way:

```
data E = Char Char | Choice E E | Seq E E | Star E
```

We define a function to encode uppercase characters into integers:

```
char_encode :: Char -> Int
char_encode c = if isUpper c
                then ord c - (ord 'A')
                else error
                     "char_encode is only defined on uppercase characters"
```

The following defines `eqlc t` to be the correspondent of $(\dot{=}\, t)$:

```
eqlc :: Char -> Ruby
eqlc c = eqlg `o` rid `x` (k (char_encode c)) `o` (INV outl) `o` (width 5)
```

Note that this definition includes a type constraint (`width`). Now the definition of τ is straightforward:

```
reorg = WIRING (PAIR (LEAF 1) (pair 0 2), PAIR (pair 0 1) (LEAF 2))

tau :: E -> Ruby
tau (Char t)     =      delay
                   `o`  andg
                   `o`  (eqlc t) `x` rid
tau (Choice e f) = org `o` ((tau e) `split2` (tau f))
tau (Seq e f)    = tau f `o` (outl `split2` (tau e))
tau (Star e)     = feedback2 (org  `o`  rid `x` (tau e)  `o`  reorg)
```

```
? r2t (tau (Seq (Char 'T') (Char 'T')))

(c1!bool & c5?[0..31] & c19?bool).
begin
    v1: var bool
  & v3: var bool
  & v5: var [0..31]
  & v19: var bool
|
  forever do
      c1!v1 || c5?v5 || c19?v19
    ; <<v1,v3>>
      := begin
             w0 = val w2 * v3
           & w2 = val v5 = w6
           & w6 = val 19
           & w20 = val w22 * v19
           & w22 = val v5 = w26
           & w26 = val 19
         | <<w0,w20>>
         end
  od
end
```

Figure 9.1: The recognizer for $\tau.(t\,;t)$.

We can now test an example: we compile the recognizer for $\tau.(t\,;t)$; see figure 9.2.0 We compile the Tangram program with the Tangram compiler `tg2hc`, and then simulate it with `hcsim`. The output is as follows:

```
% hcsim tau0 out in_a in_e

  HC2HC V1.0.3
  Copyright 1997 Philips Electronics N.V.
  All rights reserved


  HCSIM V1.0.3
  Copyright 1997 Philips Electronics N.V.
```

```
  All rights reserved

Reading handshake circuit.
Starting simulation.
End of input file in_e
End of input file in_a
Simulation finished.

Simulation took 0.00 seconds CPU time.
Simulated 983 communication events.
Simulation speed: Inf events per second.

Total time:    198 ns
Total energy:  2.211 nJ
Average power: 11.131 mW
```

The command that starts the simulation is the one after the `%` prompt. The arguments are the name of the Tangram program, followed by a filename for each of the program's channels. The contents of the files `out`, `in_a` and `in_e` for this simulation are collated together in the following table:

```
% paste out in_a in_e
0       19      0
0       19      1
0       19      0
1       19      0
0       19      1
0       20      0
0       19      0
0
```

(Here `paste` is the Unix command that adjoins a number of files line by line.) The output corresponds to what we expect. Compared with the table on page 112 obtained from Hutton's Ruby interpreter there is an extra line (the last one). The Tangram simulator performs an extra output action, since the Tangram program is ready to perform it. Then it stops, since the input actions cannot be performed, having exhausted the input.

9.2.1 The ρ design

A few auxiliary definitions that we need are:

```
plumb = WIRING ( PAIR (pair 1 2) (pair 1 3)
               , PAIR (pair 1 2) (LEAF 3)  )

str :: Ruby -> Ruby
str r = let ll = WIRING (   LEAF 0
                        ,  PAIR (pair 1 2) (PAIR (pair 1 2) (LEAF 0)) )
            rr = WIRING (   PAIR (pair 1 2) (LEAF 3)
                        ,  (pair 1 2) )
        in  ll `o` (rid2 `x` r) `o` rr

term2 = fork2 `o` (PRIM TOP)
```

The only place where we need the `TOP` primitive is in the definition of `term2`, above. The definitions of ρ and υ are then:

```
rho :: E -> Ruby
rho (Seq e (Seq f g)) = rho (Seq (Seq e f) g)
rho (Seq e f)         = (ups f) `o` plumb `o` (rho e)
rho e                 = (ups e) `o` term2

ups (Choice e f) = rid2 `x` (    org
                             `o` (split2 (str (rho e))
                                         (str (rho f))  ))
ups (Star e)     =
            rid2 `x` (feedback2 (    org
                                 `o` (rid `x` (str (rho e)))
                                 `o` reorg))
ups (Char t)     =
                (INV (delay `x` delay))
            `x` (delay `o` andg `o` ((eqlc t) `x` rid))
```

This definition is very close to the one we used for the Hutton interpreter, except for minor syntactic differences. Again we may try an example: we define `test0` to be the Gofer representation of the regular expression $(t; u; v) + (z; z)$

```
test0 = Choice (Seq (Seq (Char 'T') (Char 'U')) (Char 'V'))
                    (Seq (Char 'Z') (Char 'Z'))
```

the compiled Tangram program is in figure 9.2 on the next page. Note that we cannot compile `rho test0` directly, since this would lead to a Tangram program whose output is not driven by a delay (the main regular expression operator is choice, hence the output is driven by an "or" gate. So we have to compile `(rid2 `x` delay `o` (rho test0)` instead. The result of simulating this circuit with the inputs used on page 118 is as follows:

```
(c5!bool & c0?[0..31] & c1?bool).
begin
    v0: var [0..31]
  & v1: var bool
  & v5: var bool
  & v10: var bool
  & v11: var bool
  & v24: var [0..31]
  & v26: var bool
  & v30: var bool
  & v52: var [0..31]
  & v54: var bool
  & v58: var bool
  & v80: var [0..31]
  & v82: var bool
  & v122: var [0..31]
  & v124: var bool
  & v128: var bool
  & v150: var [0..31]
  & v152: var bool
|
  forever do
      c5!v5 || c0?v0 || c1?v1
    ; <<v5,v10,v11,v24,v26,v30,v52,v54,v58,v80,v82,v122,v124,v128,v150,v152>>
      := begin
             w4 = val v10 + v11
           & w27 = val w29 * v30
           & w29 = val v24 = w33
           & w33 = val 21
           & w55 = val w57 * v58
           & w57 = val v52 = w61
           & w61 = val 20
           & w83 = val w85 * v82
           & w85 = val v80 = w89
           & w89 = val 19
           & w125 = val w127 * v128
           & w127 = val v122 = w131
           & w131 = val 25
           & w153 = val w155 * v152
           & w155 = val v150 = w159
           & w159 = val 25
         | <<w4,w27,w125,v0,v1,w55,v24,v26,w83,v52,v54,v0,v1,w153,v122,v124>>
         end
  od
end
```

Figure 9.2: The Tangram program for `rid2 ‘x‘ delay ‘o‘ (rho test0)`

```
% paste out in_a in_e
0       25      1
0       19      1
0       25      0
0       20      0
0       0       0
1       21      0
0       0       0
0       0       0
1
```

The results correspond to the ones on page 118, except that the outputs appear one clock tick later due to the extra delay we placed on the output wire.

9.2.2 The η design

We start again with a few auxiliary definitions:

```
delays 0 = rid
delays n = delay `o` (delays (n-1))

busdelays n = (delays n) `x` (delays n)
```

The following definitions are entirely similar to the one in section 8.4 on page 118. The "shape" of a κ cell:

```
ccn n r =     (rid2 `x` org)
          `o` rsh2
          `o` ((split2 (busdelays n) r) `x` rid)

cc r    = ccn 0 r
```

The left-inverse of `cc`:

```
ccli r  = let outr2 = WIRING (LEAF 2     , PAIR (pair 0 1) (LEAF 2) )
              outl2 = WIRING ((pair 0 1) , PAIR (pair 0 1) (LEAF 2) )
          in     outr2
             `o` r
             `o` rid2 `x` false
             `o` (INV outl2)
```

The κ cell:

```
kappa (Char t)     = cc (tau (Char t))
kappa (Choice e f) = (kappa e) `o` (kappa f)
kappa (Seq e f)    =
              cc (    (ccli (kappa f))
                  `o` (split2 outl
                              (ccli (kappa f))  ))
kappa (Star e)     =
              cc (feedback2 (    org
                             `o` (rid `x` (ccli (kappa e)))
                             `o` reorg))
```

Finally, the definitions of ℓ and η:

```
ell (Char t)     = 0
ell (Choice e f) = ell e + (ell f) + 1
ell (Seq e f)    = ell e + (ell f)
ell (Star e)     = 0


eta (Char t)     = kappa (Char t)

eta (Choice e f) =     eta e
                   `o` ((delay `x` delay) `x` delay)
                   `o` (eta f)

eta (Seq e f)    =
            let n = ell e
                m = ell f
            in  ccn (n+m) (    (ccli (kappa f))
                           `o` (split2 ((delays n) `o` outl)
                                       (ccli (kappa e))      ))

eta (Star e)     = kappa (Star e)
```

Let's now define the regular expression $t + u\,; u + v\,; v\,; v$:

```
test1 = Choice (Char 'T')
               (Choice (Seq (Char 'U') (Char 'U'))
                       (Seq (Seq (Char 'V') (Char 'V')) (Char 'V')) )
```

The output of compiling `delay `o` (ccli (eta test1))` can be seen in figure 9.3 on the next page.

```
(c1!bool & c210?[0..31] & c212?bool).
begin
    v1: var bool
  & v3: var [0..31]
  & v4: var bool
  & v10: var bool
  & v11: var bool
  & v50: var [0..31]
  & v52: var bool
  & v61: var bool
  & v83: var bool
  & v146: var bool
  & v210: var [0..31]
  & v212: var bool
  & v243: var bool
  & v329: var bool
  & v390: var bool
|
  forever do
      c1!v1 || c210?v210 || c212?v212
    ; <<v1,v3,v4,v10,v11,v50,v52,v61,v83,v146,v243,v329,v390>>
      := begin
            w0 = val v10 + v11
          & w25 = val w27 * v4
          & w27 = val v3 = w31
          & w31 = val 19
          & w54 = val w60 + v61
          & w60 = val v83 + w84
          & w77 = val v146 + w147
          & w84 = val false
          & w98 = val w100 * w77
          & w100 = val v50 = w104
          & w104 = val 20
          & w147 = val false
          & w161 = val w163 * v52
          & w163 = val v50 = w167
          & w167 = val 20
          & w214 = val w220 + w221
          & w220 = val v243 + w244
          & w221 = val false
          & w237 = val w306 + w307
          & w244 = val false
          & w258 = val w260 * w237
          & w260 = val v210 = w264
          & w264 = val 21
          & w306 = val v329 + w330
          & w307 = val false
          & w323 = val v390 + w391
          & w330 = val false
          & w344 = val w346 * w323
          & w346 = val v210 = w350
          & w350 = val 21
          & w391 = val false
          & w405 = val w407 * v212
          & w407 = val v210 = w411
          & w411 = val 21
        | <<w0,v50,v52,w25,w54,v210,v212,w214,w98,w161,w258,w344,w405>>
        end
  od
end
```

Figure 9.3: The Tangram program for `delay ‘o‘ (ccli (eta test1))`

Chapter 10

A machine-checked derivation

In this section we use a proof checker called PVS (Rushby et al. [40]) to verify part of the carré derivation. This entails the definition and verification of part of the theory of chapters 2 and 3, as well as the introductory material of section 4. The work consists in rewriting the relevant definitions and theorems in the language of PVS, and in leading the theorem prover to verify all theorems. The resulting body of equality lemmas can be used as a calculus that allows us to reproduce the proof of (5.1).

The work described in this chapter is not a report on a complete and practically usable embedding of Ruby in PVS, but rather a demonstration that such an embedding is feasible and useful. Rasmussen [44] describes a much more in-depth work, using the Isabelle theorem prover (Paulson [42]). The present exposition may be useful for didactic purposes, since it is small enough to be quickly understood.

PVS is a system designed to assist in the job of formally verifying theorems. Its main parts are a specification language, a type-checker and a theorem prover. The specification language is based on a higher order logic, similar to the one used by HOL (Gordon [10]). The main difference with respect to HOL is that it is possible to define arbitrary subtypes, by restricting a type with a predicate. For instance, the set of non-zero integers is defined in PVS by

```
NZI: TYPE = { x:int | x /= 0 }
```

It is then possible to declare integer division as a total function:

```
div: [int,NZI -> int]
```

This flexibility makes type-checking undecidable, since to type-check a formula may involve arbitrary theorem proving. In this respect the high automation of the theorem prover is helpful, since many type-checking conditions can be discharged by the theorem prover without human assistance.

The theorem prover makes use of decision procedures to allow many simple facts about arithmetic to be automatically discharged. PVS comes with a number of theories and theorems pre-proved, the so-called "prelude". It is possible to define tree-like datatypes of the kind provided in Gofer or ML; for each such datatype a number of axioms is generated, such as rules for structural induction.

The general philosophy of PVS is to allow the maximum ease of use at the expense of flexibility (theorem proving systems are well-known to be awkward to use). By contrast, systems like HOL are somewhat harder to use, but allow greater scope for customization. Both HOL and PVS have their own logic language; but HOL has a programmable meta-language called ML (this is in fact a variant of the general-purpose ML programming language described e.g., in [43]). Terms in the logic language are just elements of the ML type "term"; interaction with the theorem prover happens through interaction with the ML interpreter prompt. This is analogous to the use of Lazy ML to interact with the Ruby interpreter [23] or the use of Gofer to interact with the Ruby-to-Tangram compiler of chapter 9.

Having a programming language as a meta-language makes it much more natural to automate various things such as proof procedures, or the parsing and pretty-printing of embedded languages. The downside is that it makes the syntax more cluttered; for instance, the definition of an "and" gate may look like the following:

```
let AND_DEF = new_definition
    (‘AND_DEF‘, "AND_DEF(a:num->bool,b:num->bool,out:num->bool)
    = !t. out t = ((a t) AND (b t))");;
```

As is evident by this example, one is forced to mix the logic language and the meta-language at all times.

PVS instead lacks a meta-language. This makes it much more difficult to embed custom languages in the PVS language; for instance, it is difficult to write a term representing an imperative program using a syntax that resembles the traditional Algol syntax. On the other hand, one does not have to learn and use two languages at the same time. All interaction with PVS is done through the Emacs editor window; commands to the theorem prover are given by means of commands defined in Emacs. This makes interaction

simpler. Proof procedures can still be coded when needed (they are called "strategies" in PVS jargon.) We'll see an example of a custom PVS strategy later.

The main reason for doing this work is to get a chance to look at the theory with fresh eyes. Working within the limitations of a fixed specification language forces one to think about the definitions under a different light. For instance, compare the definition of delay from chapter 2 with the one given below. A second advantage is that small details that may be overlooked in paper-and-pencil work come to the forefront. For instance, before starting this work we didn't realize that the "lifting" operation described in chapter 3 is actually a collection of operations, one for each arity of the operation to be lifted.

10.0 The theories

Our verification effort is structured in a number of "theories", a theory being simply a collection of definitions and theorems (the PVS equivalent of programming languages "modules"). Theories can be parameterized, so that one can, for instance, build a theory of relations over arbitrary types. Theories can "import" other theories; this means that all the definitions and theorems of the imported theory are made available to the theory that imports them. Theories are a means of keeping one's work organized; it would be possible to work with a single, large theory. Our work is divided into six theories; six other theories were automatically generated (those whose name begins with `tree_adt` or `plus_adt`), and one, `sets`, is from the PVS prelude. The diagram in figure 10.0 on page 180 shows the inclusion relation between the theories. The main theorem we aim to prove is enunciated in theory `carre`.

All the theorems and lemmas enunciated in the theories that follow were proved with PVS.

In the rest of this section we examine the contents of each theory.

10.0.0 The theory of relations

The first step is to define a theory of relations; this is done in theory `rel`, which in turn is based on the PVS prelude theory `sets`. Sets are represented in PVS by predicates. Binary relations are just a special case of predicates. In this we follow closely the exposition of relational calculus given in chapter 2.

The main notational difference is that we are forced to write `R(x,y)`, where we previously wrote $x\langle R\rangle y$. The theory is parameterized by a type `t`, and our relations are relations between arbitrary pairings of elements of `t`. The notion of "arbitrary pairing" is captured by binary trees. The definition of the tree datatype is as follows:

```
tree[t:TYPE+] : DATATYPE
  BEGIN
    scalar (scalar: t): scalar?
    pair   (left: tree, right: tree): pair?
  END tree
```

For any type `X`, this defines `tree[X]` to be a type, with the unary constructor `scalar` and the binary constructor `pair`, with corresponding predicates `scalar?` and `pair?` and accessor functions `scalar`, `left` and `right`. Theory `tree_props` defines simple lemmas about trees that we needn't describe in detail.

The theory starts with the declaration of the parameters, and of the imported theories. We declare `U` (for "Universe") an abbreviation for the set of trees of elements of `t`. Theory `set` is imported with argument `[U,U]`, which is the cartesian product of `U` with itself. This means that whenever we refer to an element of the type `set`, we mean "a subset of `[U,U]`". Type `rel` is declared as a synonym for `set`.

```
rel [ t : TYPE+ ] : THEORY
  BEGIN
  IMPORTING tree[t]
  U:        TYPE = tree
  IMPORTING sets[[U,U]]
  rel:      TYPE = set
```

Next we declare some variables; these declarations mean, for instance: "`R` is a variable ranging over `rel`", and allow us to use the name `R` freely without need to declare its type in every formula it occurs.

```
  R,S,T,U: VAR rel
  x,y,z,v: VAR U
  f:       VAR [U -> U]
```

Next we declare `o` as the operator symbol for relational composition, so that we can write `R o S` for $R \circ S$.

```
  ; o(R,S)(x,y): bool = EXISTS z: R(x,z) AND S(z,y)
```

The definition of inverse, identity, $\top\top$ and $\bot\bot$ are natural:

```
inv(R)(x,y): bool = R(y,x)

I(x,y)     : bool = (x = y)

top(x,y)   : bool = TRUE
bot(x,y)   : bool = FALSE
```

We define a predicate that states that a relation is deterministic:

```
determ?(R) : bool = FORALL x,y,z: R(y,x) AND R(z,x) IMPLIES y = z
```

Now a number of basic facts is enunciated about the definitions so far:

```
id0:          THEOREM R o I = R
id1:          THEOREM I o R = R

comp0:        THEOREM R o union(S,T) = union(R o S , R o T)
comp1:        THEOREM union(R,S) o T = union(R o T , S o T)
comp_assoc:   THEOREM (R o S) o T = R o (S o T)

comp_determ: THEOREM (determ?(R) AND determ?(S)) IMPLIES determ?(R o S)

det_rel_absorb_r: LEMMA (FORALL x,y: R(x,y) = (x=f(y)))
                           IMPLIES (S o R)(z,v) = S(z,f(v))
det_rel_absorb_l: LEMMA (FORALL x,y: R(x,y) = (y=f(x)))
                         IMPLIES (R o S)(z,v) = S(f(z),v)

inv_inv:    THEOREM inv(inv(R)) = R
inv_comp:   THEOREM inv(R o S) = inv(S) o inv(R)
inv_id:     THEOREM inv(I) = I
inv_eq:     THEOREM (inv(R) = inv(S)) = (R = S)
inv_inter: THEOREM inv(intersection(R,S)) = intersection(inv(R), inv(S))
inv_union: THEOREM inv(union(R,S)) = union(inv(R), inv(S))
```

Now we define product and split. For product we choose the symbol * since it is the closest to $\times$ among the ones that can be used in infix notation. For split, we use functional notation.

```
; *(R,S)(x,y): bool =
         IF pair?(x) AND pair?(y)
         THEN R(left(x),left(y)) AND S(right(x),right(y))
```

```
           ELSE FALSE  ENDIF

  split(R,S)(x,y): bool =
           IF pair?(x) THEN R(left(x), y) AND S(right(x), y)
           ELSE FALSE ENDIF
```

Again, we have the enunciation of a number of basic facts:

```
  inv_prod: THEOREM inv(R*S) = inv(R)*inv(S)
  fusion:   THEOREM (R o S) * (T o U) = (R * T) o (S * U)

  split_fusion0: THEOREM (R * S) o split(T, U) = split(R o T, S o U)

  split_fusion1: THEOREM
    determ?(T) IMPLIES split(R, S) o T = split(R o T, S o T)

  split_ldomain: THEOREM split(R,S) = (I*I) o split(R,S)

  split_determ:  THEOREM
    determ?(R) AND determ?(S) IMPLIES determ?(split(R,S))

  END rel
```

This concludes the theory of relations.

10.0.1 The theory of circuits

Theory `circuits` is mainly concerned with delay and its properties. It also defines the lifting operation for unary and binary functions. It defines type `stream` to be the type of functions from the integers to the theory parameter `t`. By importing theory `rel` with parameter `stream` we obtain that our circuits are relations over trees of streams. (The `t` parameter in theory `rel` is bound to type `stream`.)

```
circuits [ t: TYPE+ ] : THEORY
  BEGIN
  stream: TYPE = [int -> t]
  IMPORTING rel[stream]
  IMPORTING tree_props[stream]

  R,S,T,U: VAR rel
  x,y,z,w: VAR tree[stream]
  l,m,n:   VAR int
```

The following predicates characterize particular classes of trees. Theorem pair_of_pairs_eta facilitates manipulation of pair-of-pairs.

```
pair_of_scalars?(x): bool =
      pair?(x) AND scalar?(left(x)) AND scalar?(right(x))

pair_of_pairs?(x): bool =
      pair?(x) AND pair?(left(x)) AND pair?(right(x))

pair_of_pairs_eta: THEOREM
      (FORALL (x: (pair_of_pairs?)):
         pair(
             pair(left(left(x)), right(left(x)))
           , pair(left(right(x)), right(right(x))) )
         = x)
```

While we talked in previous chapters of lifting as a single operation, to be more precise we should define a different lifting operation for each arity of the lifted relation. For our needs it suffices to define lifting for binary deterministic relations; lifting of unary functions is also given for demonstration.

```
lift1(f: [t -> t])(x,y): bool =
  IF scalar?(x) AND scalar?(y)
  THEN FORALL n: scalar(x)(n) = f(scalar(y)(n))
  ELSE FALSE ENDIF

lift2(f: [t,t -> t])(x,y): bool =
  IF pair_of_scalars?(y) AND scalar?(x)
  THEN FORALL n: scalar(x)(n) = f(scalar(left(y))(n), scalar(right(y))(n) )
  ELSE FALSE ENDIF
```

Delay and antidelay are deterministic relations; it helps to define them in such a way that the functional dependence between the left and right arguments is easily shown. So, we first define a function that when applied to a tree of streams it applies primitive delay to all the leaves of the tree; then we use these functions in the definition of delay as a relation. As we said, the definition of delay given here is syntactically much different from the one given in chapter 3, making use of tree map rather than fixed points. The tree map can be defined recursively, in a way that is probably more readily understandable by people with a knowledge of computer programming than a definition using fixed points. On the other hand, the meaning of recursive definitions is usually given by means of fixed point, so the difference between the two ways of defining delay is not great. Basic theorems about delays follow.

```
delayed: [U -> U] = map (LAMBDA (f:stream): LAMBDA n: f(n-1))
antidelayed: [U -> U] = map (LAMBDA (f:stream): LAMBDA n: f(n+1))

delay(x,y): bool = (x = delayed(y))
yaled(x,y): bool = (y = delayed(x))

del_antidel_inverse: THEOREM delayed(antidelayed(x)) = x
antidel_del_inverse: THEOREM antidelayed(delayed(x)) = x

delay_invertible: THEOREM delay(x,y) = (y = antidelayed(x))
yaled_invertible: THEOREM yaled(x,y) = (x = antidelayed(y))

inv_delay: THEOREM inv(delay) = yaled
inv_yaled: THEOREM inv(yaled) = delay

delay_poly0: THEOREM delay o (I*I) = delay*delay
delay_poly1: THEOREM (I*I) o delay = delay*delay

delay_yaled: THEOREM delay o yaled = I
yaled_delay: THEOREM yaled o delay = I

delay_determ: THEOREM determ?(delay)
```

Finally, we have some theorems about lifted functions:

```
lift2_rdom: THEOREM FORALL (f: [t,t -> t]): lift2(f) o (I*I) = lift2(f)
lift2_determ: THEOREM FORALL (f: [t,t -> t]): determ?(lift2(f))
lift2_delayed: LEMMA FORALL (f: [t,t -> t]):
   lift2(f)(x,pair(delayed(y),delayed(z))) = lift2(f)(antidelayed(x),pair(y,z))

retiming_lift2: THEOREM FORALL (f: [t,t -> t]):
                        lift2(f) o delay = delay o lift2(f)

END circuits
```

So much for the theory of circuits.

10.0.2 The theory of tuples

Theory `tuples` reproduces part of the material from section 4. It defines a way to interpret trees as tuples. Type `upfrom(1)` used below is the type of positive integers.

```
tuples [t: TYPE+] : THEORY
  BEGIN
  IMPORTING circuits[t]

  n,m: VAR upfrom(1)
  R,S,T,U: VAR rel
  a,b,c,d: VAR U
```

The `max_width` of a tree is the maximum tuple width of a tree; i.e., the number of times we can take the "right" path from the root of the tree.

```
  max_width(a): RECURSIVE upfrom(1) =
    IF scalar?(a) THEN 1
    ELSE 1+max_width(right(a)) ENDIF
    MEASURE a BY <<
```

The following predicates characterise certain classes of trees; a tree that satisfies `n_tuple?(n)` can be interpreted as a tuple of (at least) n elements.

```
  n_tuple?(n)(a): bool = n <= max_width(a)

  n_tuple_of_scalars?(n)(a): RECURSIVE bool =
    IF n = 1
    THEN scalar?(a)
    ELSE     pair?(a)
         AND scalar?(left(a))
         AND n_tuple_of_scalars?(n-1)(right(a)) ENDIF
    MEASURE n
```

Next are the definitions of various combinators, all of which closely follow the ones in section 4.

```
  map(n,R): RECURSIVE rel =
    IF n = 1 THEN R ELSE R * map(n-1, R) ENDIF
    MEASURE n

  arity(n): rel = map(n, I)

  fold(n, R): RECURSIVE rel =
    IF n = 1 THEN I ELSE R o (I * fold(n-1, R)) ENDIF
    MEASURE n

  fork(n): RECURSIVE rel =
```

```
    IF n = 1 THEN I ELSE split(I, fork(n-1)) ENDIF
    MEASURE n

  tri(n,R): RECURSIVE rel =
    IF n = 1 THEN I ELSE I * (tri(n-1, R) o map(n-1, R)) ENDIF
    MEASURE n
```

The label CHALLENGE is formally equivalent to THEOREM; it can be used to label things that can be used to "challenge" a definition; when a definition is complicated it may take some care to become convinced that what was defined is indeed what one had in mind. One way to "test" a definition is to see if some simple expected consequences are indeed provable. Here are a few such challenges.

```
  map:  CHALLENGE map(2,R) = R*R
  fold: CHALLENGE R o (I*I) = R IMPLIES fold(2, R) = R
  fork: CHALLENGE fork(2) = split(I,I)
  tri:  CHALLENGE tri(2,R) = I * R
```

Then we have a list of theorems; all of these were proved by induction on n.

```
  map_step: THEOREM map(n+1, R) = R*map(n,R)
  tri_step: THEOREM tri(n+1, R) = I*(tri(n,R) o map(n,R))

  map_fusion: THEOREM map(n, R) o map(n, S) = map(n, R o S)

  fold_map: THEOREM R o (S*S) = S o R
                    IMPLIES fold(n, R) o map(n, S) = S o fold(n, R)

  fork_fusion1: THEOREM determ?(R)
                        IMPLIES fork(n) o R = map(n,R) o fork(n)

  fork_ldom: THEOREM map(n, I) o fork(n) = fork(n)

  map_map: THEOREM T o S = S o R
                     IMPLIES map(n,T) o map(n,S) =  map(n,S) o map(n,R)

  tri_map: THEOREM T o S = S o R
                   IMPLIES tri(n,T) o map(n,S) =  map(n,S) o tri(n,R)

  horner: THEOREM R o (S*S) = S o R
                   IMPLIES fold(n, R) o tri(n, S) = fold(n, R o (I*S))
  END tuples
```

This concludes the theory of tuples.

10.0.3 The theory of zip_n

The definitions and proofs for zip_n are complicated enough to deserve a theory of their own. The theory begins with the usual declarations, and then defines predicates that are later used as subtypes of trees.

```
zip [t: TYPE+] : THEORY
  BEGIN
  IMPORTING tuples[t]

  n,m: VAR upfrom(1)
  R,S,T,U: VAR rel
  a,b,c,d: VAR U

  pair_of_n_tuples?(n)(a): bool =
    pair?(a) AND n_tuple?(n)(left(a)) AND n_tuple?(n)(right(a))

  tuple_of_pairs?(n)(a): RECURSIVE bool =
    IF n = 1 THEN pair?(a)
    ELSE pair?(a) AND pair?(left(a)) AND tuple_of_pairs?(n-1)(right(a))
    ENDIF
    MEASURE n

  tuple_of_pairs_pair: LEMMA tuple_of_pairs?(n)(a) => pair?(a)
  tuple_of_pairs0:     LEMMA (n /= 1 AND tuple_of_pairs?(n)(a))
                             => pair?(left(a))
  tuple_of_pairs1:     LEMMA (n /= 1 AND tuple_of_pairs?(n)(a))
                             => tuple_of_pairs?(n-1)(right(a))
  tuple_of_pairs2:     LEMMA FORALL (n: upfrom(2)):
                             tuple_of_pairs?(n)(a)
                             => tuple_of_pairs?(n-1)(a)
```

We observed earlier that delay, being a deterministic relation, is better defined by means of a function; for this makes it easier to manipulate it in proofs. The same applies to *zip*, so here we define `fzip` (for "functional zip") in much the same way as it is defined in functional programs. Note that the predicate `pair_of_n_tuples?(n)` defined earlier is used to type the second argument.

```
  fzip(n: upfrom(1), a: (pair_of_n_tuples?(n)))
   : RECURSIVE (tuple_of_pairs?(n)) =
    IF n = 1 THEN a
    ELSE LET b = left(a), c = right(a)
```

```
        IN pair(
                 pair(left(b), left(c))
               , fzip(n-1, pair(right(b), right(c)))
               )
  ENDIF
  MEASURE n
```

Now `funzip` is the inverse of `fzip`

```
funzip(n: upfrom(1), a: (tuple_of_pairs?(n)))
 : RECURSIVE (pair_of_n_tuples?(n)) =
  IF n = 1 THEN a
  ELSE LET b = funzip(n-1, right(a))
       IN pair(pair(left(left(a)), left(b)), pair(right(left(a)), right(b)) )
  ENDIF
  MEASURE n
```

Now we can give the definition of *zip* seen as a relation. Then challenges to the definition and properties of *zip* are are declared. As usual, properties that depend on n are proved by induction on n. Theorem `zip_fold` is not necessary for the proof of the carré theorem. It is included because it is interesting in its own right.

```
zip(n)(a,b): bool = IF pair_of_n_tuples?(n)(b)
                    THEN a = fzip(n, b)
                    ELSE FALSE ENDIF


fzip2_fzip2: THEOREM pair_of_pairs?(a) IMPLIES fzip(2, fzip(2, a)) = a

fzip_funzip_inverse: THEOREM tuple_of_pairs?(n)(a)
                             IMPLIES fzip(n, funzip(n, a)) = a
funzip_fzip_inverse: THEOREM pair_of_n_tuples?(n)(a)
                             IMPLIES funzip(n, fzip(n, a)) = a

zip_invertible: THEOREM pair_of_n_tuples?(n)(b)
                        AND tuple_of_pairs?(n)(a)
                        IMPLIES zip(n)(a,b) = (b = funzip(n, a))

fzip2: CHALLENGE   fzip(2, pair(pair(a,b),pair(c,d)))
                 = pair(pair(a,c),pair(b,d))

zip1: CHALLENGE zip(1) = I*I
```

```
  zip2: CHALLENGE zip(2)( pair(pair(a,b),pair(c,d)) ,
                          pair(pair(a,c),pair(b,d)) )

  zip2_inv: THEOREM inv(zip(2)) = zip(2)

  zip_step:   THEOREM zip(n+1) = (I*zip(n)) o zip(2)
  zip2_rdom0: THEOREM zip(2) o (I*I) = zip(2)
  zip2_rdom1: THEOREM zip(2) o ((I*I)*(I*I)) = zip(2)
  zip2_ldom1: THEOREM ((I*I)*(I*I)) o zip(2) = zip(2)
  zip2_lemma0: LEMMA  (zip(2) o R)(pair(pair(a,b),pair(c,d)), e)
                      = R(pair(pair(a,c),pair(b,d)), e)
  zip2_lemma1: LEMMA  (R o zip(2))(e, pair(pair(a,b),pair(c,d)))
                      = R(e, pair(pair(a,c),pair(b,d)))

  zip2_comm:  THEOREM zip(2) o ((R * S)*(T * U)) = ((R * T)*(S * U)) o zip(2)

  zip2_split: THEOREM   zip(2) o split(split(R, S), split(T, U))
                      = split(split(R, T), split(S, U))

  zip_ldom:   THEOREM map(n, I*I) o zip(n) = zip(n)
  zip_fold:   THEOREM FORALL (n:upfrom(2)): zip(n) = inv(fold(n, zip(2)))

  zip_map: THEOREM zip(n) o (map(n,R) * map(n,S)) = map(n,R*S) o zip(n)
  zip_tri: THEOREM zip(n) o (tri(n,R) * tri(n,S)) = tri(n,R*S) o zip(n)

  zip_split1: THEOREM   zip(n) o split(fork(n),fork(n))
                      = map(n, split(I,I)) o fork(n)

END zip
```

So much for the theory of *zip*.

10.0.4 The theory of carré

Finally, we have theory `carre`, where the main theorem is enunciated. This is the only non-parameterized theory that we use, since we are able to fix the universe of values that we need to the disjoint union of the booleans and the naturals. The `plus` datatype defines the disjoint sum of two sets in the manner that is usual in functional programming languages like Gofer or SML:

```
plus  [T,S: TYPE]: DATATYPE
```

```
  BEGIN
  inl (fst: T): inl?
  inr (snd: S): inr?
  END plus
```

(A more traditional way to define the disjoint union of sets A and B is $A + B = \{(0, a) \mid a \in A\} \cup \{(1, b) \mid b \in B\}$.) So, when `plus` is instantiated with the types `bool` and `nat`, as in `plus[bool,nat]` we have that, for instance, `inl(FALSE)` is an element that satisfies `inl?(inl(FALSE))` and falsifies `inr?(inl(FALSE))`.

```
carre : THEORY
  BEGIN
  IMPORTING plus[bool,nat]
  inlFALSE: plus = inl(FALSE)
  IMPORTING zip[plus]

  R,S,T,U: VAR rel
  a,b,c,d: VAR U
  x,y,z:   VAR plus
  n,m:     VAR upfrom(1)
```

Next we have the definitions of $\lhd^n$, $\doteq$ and $\dot{\wedge}$; `andg` stands for "and gate". The name "and" is a reserved keyword of PVS.

```
  delays(n): RECURSIVE rel =
    IF n = 1 THEN delay ELSE delay o (delays(n-1)) ENDIF
    MEASURE n

  eqlg: rel = lift2(LAMBDA x,y: inl (x = y))
  andg: rel = lift2(LAMBDA x,y:
                      inl(inl?(x) AND inl?(y) AND fst(x) AND fst(y)) )
```

Now a few lemmas are introduced; since they are meaningful on their own it makes sense to declare and prove them separately rather than proving them in the main proof. Another reason for declaring them separately is that some of these are used more than once in the proof of the carré theorem.

```
  eqlg_rdom: LEMMA eqlg o (I*I) = eqlg
  andg_rdom: LEMMA andg o (I*I) = andg
  retiming_eqlg: LEMMA eqlg o (delay*delay) = delay o eqlg
  retiming_andg: LEMMA andg o (delay*delay) = delay o andg
  delays_determ: LEMMA determ?(delays(n))
```

Finally, we enunciate the main theorem.

```
  carre: THEOREM   fold(n, andg)
                 o map(n, eqlg)
                 o zip(n)
                 o (tri(n, delay)*tri(n,delay))
                 o split(fork(n), fork(n) o delays(n))
               =
                   fold(n, andg o (I * delay))
                 o fork(n)
                 o eqlg
                 o split(I, delays(n))

END carre
```

This concludes the exposition of the PVS theories.

10.1 Proofs

When the PVS theorem prover is invoked on a formula, the formula is transformed into a sequent, that is a pair (Γ, Δ) where Γ is an (initially empty) list of antecedent formulæ, and Δ is a list of consequent formulæ, which initially contains just the formula to be proved. Informally, a sequent is "true" if the conjunction of the antecedents implies the disjunction of the consequents. A proof consists in the application of rules of inference, that manipulate in various way the sequent, or reduce it to a list of simpler sequents. A proof attempt then has the shape of a tree, where the root is the original sequent, and the leaves are sequents whose truth implies the truth of the original sequent. A proof is complete when every leaf sequent is trivially true (or can be shown to be by use of decision procedures.) The primitive rules of inference can be combined in arbitrary proof procedures called "strategies" or "tactics". Many tactics are provided with PVS, and the users can write their own.

By way of example, we now see the proof of $R \circ \iota = R$, theorem `id0` in theory `rel`. Figure 10.1 on page 181 shows the tree structure of the proof.

The proof begins with the following sequent:

```
id0 :

  |-------
{1}    (FORALL (R: rel): R o I = R)
```

We use `(skolem!)` to eliminate the quantification, introducing the skolem constant `R!1` (think of it as `R` primed, or `R'`). (Usually, by skolemization is meant a rule that allows one to replace an existentially quantified *antecedent* variable with a fresh constant. The PVS rule can also perform the symmetric operation of replacing an universally quantified *consequent* variable with a constant.)

```
Rule? (skolem!)
Skolemizing,
this simplifies to:
id0 :

  |-------
{1}    R!1 o I = R!1
```

We are going to do a pointwise proof. The command `(apply-extensionality)` applies the rule $R = S \Leftarrow \forall(x, y : x\langle R\rangle y \equiv x\langle S\rangle y)$ and then skolemizes the universal quantification. The `hide? t` flag means that we want to "hide" from the resulting sequent the original formula. Next we expand the definitions of composition and identity.

```
Rule? (apply-extensionality :hide? t)
Applying extensionality,
this simplifies to:
id0 :

  |-------
{1}    (R!1 o I)(x!1, x!2) = R!1(x!1, x!2)

Rule? (expand* "o" "I")
Expanding the definition(s) of (o I),
this simplifies to:
id0 :

  |-------
{1}    ((EXISTS (z: U): (R!1(x!1, z) AND (z = x!2))) = R!1(x!1, x!2))
```

PVS rules do not allow the manipulation of quantifiers, unless the quantified term is the "top" term in the formula. The main function symbol in the current sequent is boolean equality; the only way to get rid of it is to change it into a propositional connective by `(iff)`, and then do a proof by mutual implication.

```
Rule? (iff)
Converting top level boolean equality into IFF form,
Converting equality to IFF,
this simplifies to:
id0 :

  |-------
{1}    (EXISTS (z: U): (R!1(x!1, z) AND (z = x!2))) IFF R!1(x!1, x!2)

Rule? (ground)
Applying propositional simplification and decision procedures,
this yields  2 subgoals:
id0.1 :

{-1}    EXISTS (z: U): (R!1(x!1, z) AND (z = x!2))
  |-------
{1}    R!1(x!1, x!2)
```

Now we must deal with two subproofs, which are both very easy. The first is dealt with by skolemizing the antecedent existential, and then applying the general-purpose tactic `(ground)`:

```
Rule? (skolem!)
Skolemizing,
this simplifies to:
id0.1 :

{-1}    (R!1(x!1, z!1) AND (z!1 = x!2))
  |-------
[1]    R!1(x!1, x!2)

Rule? (ground)
Applying propositional simplification and decision procedures,

This completes the proof of id0.1.
```

For the second subproof it is sufficient to instantiate the existential, and then calling `(assert)`, which tells the prover that the sequent "follows easily" from the decision procedures.

```
id0.2 :

{-1}    R!1(x!1, x!2)
```

```
  |-------
{1}    EXISTS (z: U): (R!1(x!1, z) AND (z = x!2))

Rule? (inst 1 "x!2")
Instantiating the top quantifier in 1 with the terms:
 x!2,
this simplifies to:
id0.2 :

[-1]    R!1(x!1, x!2)
  |-------
{1}    (R!1(x!1, x!2) AND (x!2 = x!2))

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of id0.2.

Q.E.D.
```

This proof is very simple, and was conducted at a fine level of detail for the purpose of demonstration; however, it is often convenient to leave such simple proofs to the automated proof procedures. The proof of `id1` below makes use of the powerful `(grind)` tactic and is considerably shorter:

```
id1 :

  |-------
{1}    (FORALL (R: rel): I o R = R)

Rule? (skolem!)
Skolemizing,
this simplifies to:
id1 :

  |-------
{1}    I o R!1 = R!1

Rule? (apply-extensionality :hide? t)
Applying extensionality,
this simplifies to:
id1 :
```

```
  |-------
{1}    (I o R!1)(x!1, x!2) = R!1(x!1, x!2)

Rule? (grind)
I rewrites I(x!1, z)
  to  (x!1 = z)
O rewrites (I o R!1)(x!1, x!2)
  to  EXISTS (z: U): (x!1 = z) AND R!1(z, x!2)
I rewrites I(x!1, z)
  to  (x!1 = z)
O rewrites (I o R!1)(x!1, x!2)
  to  EXISTS (z: U): (x!1 = z) AND R!1(z, x!2)
Trying repeated skolemization, instantiation, and if-lifting,
Q.E.D.
```

* *
*

The style of proof that we'd like to use when verifying Ruby equalities would ideally be similar to the one we employ with paper and pencil. This ideal is not fully attainable with current theorem proving technology; but we try to get as close as possible to it. In particular, in the supporting theories `rel`, `circuits`, `tuples` and `zip` we prove a series of lemmas that will make it possible to verify Ruby equalities in point-free style. The proofs of the lemmas, however, are mostly pointwise, and equalities of relations are often proved by mutual inclusion. The objective here is to relegate proofs by induction, mutual inclusion and by pointwise reasoning to the proofs of lemmas as much as possible, since these proof styles generally do not lead to concise calculational proofs.

One of the problems we find in applying equational lemmas to a circuit term is that the rewriting rules in PVS do not take into account the associativity of composition. As a result, some steps which one finds obvious are complicated by the need to rewrite several times the goal term with the associativity lemma. To obviate this problem, we devised a simple rewriting rule "modulo associativity" which we called `assoc-rewrite`. It works by first rewriting the term with associativity, associating all compositions to the left; then it alternately tries to rewrite with the given lemma and then with the associativity rule, until all compositions are associated to the right. The code for `assoc-rewrite` is in figure 10.2 on page 182.

10.2 The carre proof

The main theorem we aim to prove is theorem `carre` from the `carre` theory:

```
carre: THEOREM   fold(n, andg)
                 o map(n, eqlg)
                 o zip(n)
                 o (tri(n, delay)*tri(n,delay))
                 o split(fork(n), fork(n) o delays(n))
               =
                 fold(n, andg o (I * delay))
                 o fork(n)
                 o eqlg
                 o split(I, delays(n))
```

This corresponds to a proof of (5.1). The PVS proof we obtain has the shape of a tree (see figure 10.3 on page 183). Whenever a rule with provisos is used, the current sequent is decomposed into a list of sequents where the first is the result of applying the rule, and the others are the provisos that must hold for the rule to be valid. For instance, the rule `(rewrite "fork_fusion1")`, which rewrites the current goal with the named lemma, has a proviso, which is easily discharged by the appeal to lemma `delays_determ`. As a result, one can follow the main line of reasoning by traversing the tree from the root and taking always the leftmost inferior node.

The carré proof starts with the following sequent:

```
carre :

  |-------
{1}    (FORALL (n: upfrom(1)):
          fold(n, andg) o map(n, eqlg) o zip(n)
              o (tri(n, delay) * tri(n, delay))
              o split(fork(n), fork(n) o delays(n))
              = fold(n, andg o (I * delay)) o fork(n) o eqlg
                o split(I, delays(n)))
```

The first thing to do is to get rid of the quantification by skolemizing. We supply the name `n` for the skolem constant, since it's more readable than automatically generated names like `n!1`.

```
Rule? (skolem 1 "n")
For the top quantifier in 1, we introduce Skolem constants: n,
```

```
this simplifies to:
carre :

  |-------
{1}    fold(n, andg) o map(n, eqlg) o zip(n) o (tri(n, delay) * tri(n, delay))
          o split(fork(n), fork(n) o delays(n))
          = fold(n, andg o (I * delay)) o fork(n) o eqlg o split(I, delays(n))
```

We now follow the structure of the derivation of page 60: we apply (4.7), by rewriting modulo associativity of composition.

```
Rule? (assoc-rewrite "zip_tri")
rewriting with zip_tri modulo associativity,
this simplifies to:
carre :

  |-------
{1}    (fold(n, andg) o map(n, eqlg)) o tri(n, delay * delay) o zip(n)
          o split(fork(n), fork(n) o delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

We next want to commute $tri_n.(\lhd\times\lhd)$ with $map_n.\doteq$; this is possible by means of theorem `tri_map` and because $\lhd\times\lhd$ commutes with $\doteq$. Since `tri_map` is a conditional rewrite rule, we first introduce the proviso as an antecedent. After rewriting, we hide the proviso since it's no longer needed.

```
Rule? (lemma "retiming_eqlg")
Applying retiming_eqlg
this simplifies to:
carre :

{-1}    eqlg o (delay * delay) = delay o eqlg
  |-------
[1]    (fold(n, andg) o map(n, eqlg)) o tri(n, delay * delay) o zip(n)
          o split(fork(n), fork(n) o delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (assoc-rewrite "tri_map" :dir rl :subst ("T" "delay"))
rewriting with tri_map modulo associativity,
this simplifies to:
carre :

[-1]    eqlg o (delay * delay) = delay o eqlg
```

```
  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o split(fork(n), fork(n) o delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (hide -1)
Hiding formulas:  -1,
this simplifies to:
carre :

  |-------
[1]    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o split(fork(n), fork(n) o delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

Now we rewrite with (4.3); since we didn't supply the proviso as an antecedent, this time we'll get two goals: the first one is the original goal after rewriting, the second is the antecedent of the rewriting rule. We concentrate first on our main line of reasoning, leaving the second goal for later. Note that while our original sequent name is `carre`, the two sequents we obtain after rewriting are called `carre1` and `carre2`.

```
Rule? (rewrite "fork_fusion1")
Found matching substitution:
R gets delays(n),
n gets n,
Rewriting using fork_fusion1,
this yields  2 subgoals:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o split(fork(n), map(n, delays(n)) o fork(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

Here we want to apply the split fusion rule `split_fusion0` (4.3); in order to do so, we must have a $map_n.\iota$ term appear by applying `fork_ldom` backwards (the `:dir rl` flag means to apply the rule from right to left). This will rewrite more than necessary, bringing about more $map_n.\iota$ terms than needed; we'll take care of that later by applying `fork_ldom` in the other direction.

```
Rule? (rewrite "fork_ldom" :dir rl)
Found matching substitution:
```

```
n gets n,
Rewriting using fork_ldom,
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o
         split(map(n, I) o fork(n), map(n, delays(n)) o (map(n, I) o fork(n)))
           = (fold(n, andg o (I * delay)) o (map(n, I) o fork(n))) o eqlg
             o split(I, delays(n))

Rule? (rewrite "split_fusion0" :dir rl)
Found matching substitution:
U gets map(n, I) o fork(n),
S gets map(n, delays(n)),
T gets fork(n),
R gets map(n, I),
Rewriting using split_fusion0,
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o
         ((map(n, I) * map(n, delays(n))) o split(fork(n), map(n, I) o fork(n)))
           = (fold(n, andg o (I * delay)) o (map(n, I) o fork(n))) o eqlg
             o split(I, delays(n))

Rule? (apply (repeat (rewrite "fork_ldom")))
Applying
   (REPEAT (REWRITE "fork_ldom")),
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o ((map(n, I) * map(n, delays(n))) o split(fork(n), fork(n)))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

Next we rewrite with `zip_map` (4.6), `zip_split1`, `map_fusion` (4.1), `split_fusion0` (4.3), and finally simplify with the rule of composition identity `id0`.

```
Rule? (assoc-rewrite "zip_map")
```

```
rewriting with zip_map modulo associativity,
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o map(n, I * delays(n))
          o zip(n)
          o split(fork(n), fork(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (assoc-rewrite "zip_split1")
rewriting with zip_split1 modulo associativity,
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o map(n, I * delays(n))
          o map(n, split(I, I))
          o fork(n)
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (assoc-rewrite "map_fusion")
rewriting with map_fusion modulo associativity,
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay))
          o map(n, (eqlg o (I * delays(n))) o split(I, I))
          o fork(n)
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (assoc-rewrite "split_fusion0")
rewriting with split_fusion0 modulo associativity,
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay))
          o map(n, eqlg o split(I o I, delays(n) o I))
          o fork(n)
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

```
Rule? (apply (repeat (rewrite "id0")))
Applying
   (REPEAT (REWRITE "id0")),
this simplifies to:
carre.1 :

  |-------
{1}    (fold(n, andg) o tri(n, delay)) o map(n, eqlg o split(I, delays(n)))
          o fork(n)
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

Once again we want to `assoc-rewrite` with `fork_fusion1`, a conditional rewriting rule; as before, we want to provide the proviso as an antecedent, for otherwise `assoc-rewrite` would produce redundant subgoals. This time we do not have a ready-made lemma for the proviso so we introduce it by case analysis. We prove the theorem assuming that the proviso holds, and leave to a separate goal sequent the task of proving that it's indeed true; or, equivalently, that the negation of the proviso leads to a contradiction.

```
Rule? (case "determ?(eqlg o split(I, delays(n)))")
Case splitting on
   determ?(eqlg o split(I, delays(n))),
this yields  2 subgoals:
carre.1.1 :

{-1}    determ?(eqlg o split(I, delays(n)))
  |-------
[1]    (fold(n, andg) o tri(n, delay)) o map(n, eqlg o split(I, delays(n)))
          o fork(n)
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (assoc-rewrite "fork_fusion1" :dir rl)
rewriting with fork_fusion1 modulo associativity,
this simplifies to:
carre.1.1 :

[-1]    determ?(eqlg o split(I, delays(n)))
  |-------
{1}    (fold(n, andg) o tri(n, delay)) o fork(n) o eqlg o split(I, delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

Finally, we apply Horner's rule. The result of rewriting with `horner` is a sequent that contains Horner's rule proviso as a consequent; by appeal to the appropriate lemma, the main line of reasoning is concluded.

```
Rule? (assoc-rewrite "horner")
rewriting with horner modulo associativity,
this simplifies to:
carre.1.1 :

[-1]    determ?(eqlg o split(I, delays(n)))
  |-------
{1}    andg o (delay * delay) = delay o andg
{2}    (fold(n, andg) o tri(n, delay)) o fork(n) o eqlg o split(I, delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (lemma "retiming_andg")
Applying retiming_andg
this simplifies to:
carre.1.1 :

{-1}    andg o (delay * delay) = delay o andg
[-2]    determ?(eqlg o split(I, delays(n)))
  |-------
[1]    andg o (delay * delay) = delay o andg
[2]    (fold(n, andg) o tri(n, delay)) o fork(n) o eqlg o split(I, delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

which is trivially true.

This completes the proof of carre.1.1.
```

We are now left with the task of proving the provisos for some of the rules we applied. The first one we deal with is the one resulting from the case analysis.

```
carre.1.2 :

  |-------
{1}    determ?(eqlg o split(I, delays(n)))
[2]    (fold(n, andg) o tri(n, delay)) o map(n, eqlg o split(I, delays(n)))
          o fork(n)
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))
```

Formula 2 is not needed, so for clarity we hide it from the sequent. This is because formula 1, the proviso, is always true. (In general, when performing a case analysis, all of the context may be needed; note that the sequent $([], [A, B])$ is equivalent to the sequent $([\neg A], [B])$, which amounts to proving that from the negation of A, the thesis B follows.)

```
Rule? (hide 2)
Hiding formulas:  2,
this simplifies to:
carre.1.2 :

  |-------
[1]    determ?(eqlg o split(I, delays(n)))
```

We have a rule that says that to prove that the composition of two relations is deterministic, it suffices to prove that both relations are deterministic. This results in the splitting of the goal in two sequents.

```
Rule? (rewrite "comp_determ")
Found matching substitution:
S gets split(I, delays(n)),
R gets eqlg,
Rewriting using comp_determ,
this yields  2 subgoals:
carre.1.2.1 :

  |-------
{1}    determ?(eqlg)
[2]    determ?(eqlg o split(I, delays(n)))
```

The first subgoal is solved by expanding the definition of `eqlg` ($\doteq$) and then applying the general lemma about retiming of a lifted relation.

```
Rule? (apply (then (hide 2) (expand "eqlg")))
Applying
   (THEN (HIDE 2) (EXPAND "eqlg")),
this simplifies to:
carre.1.2.1 :

  |-------
{1}    determ?(lift2(LAMBDA x, y: inl(x = y)))

Rule? (rewrite "lift2_determ")
Found matching substitution:
f gets LAMBDA x, y: inl(x = y),
Rewriting using lift2_determ,

This completes the proof of carre.1.2.1.
```

The second subgoal is further split in two separate subgoals, one for each argument of `split`.

```
carre.1.2.2 :

  |-------
{1}    determ?(split(I, delays(n)))
[2]    determ?(eqlg o split(I, delays(n)))

Rule? (apply (then (hide 2) (rewrite "split_determ")))
Applying
   (THEN (HIDE 2) (REWRITE "split_determ")),
this yields  2 subgoals:
carre.1.2.2.1 :

  |-------
{1}    determ?(I)
[2]    determ?(split(I, delays(n)))
```

Now we must prove that the identity relation is deterministic, which is easily done by the general purpose tactic `(grind)`.

```
Rule? (apply (then (hide 2) (grind)))
I rewrites I(y, x)
  to  (y = x)
I rewrites I(z, x)
  to  (z = x)
determ? rewrites determ?(I)
  to  FORALL (x: U[stream[plus]]), (y: U[stream[plus]]), (z: U[stream[plus]]):
        (y = x) AND (z = x) IMPLIES y = z
Applying
   (THEN (HIDE 2) (GRIND)),

This completes the proof of carre.1.2.2.1.
```

Finally, we must prove that $\lhd^n$ is deterministic. This is fairly easy to do by induction; but since the other remaining subgoal also calls for the same result, it is better to have this proven as a separate lemma, rather than duplicating the argument in the present proof. By appealing twice to `delays_determ`, the proof of `carre` is complete.

```
carre.1.2.2.2 :
```

```
  |-------
{1}    determ?(delays(n))
[2]    determ?(split(I, delays(n)))

Rule? (rewrite "delays_determ")
Found matching substitution:
n gets n,
Rewriting using delays_determ,

This completes the proof of carre.1.2.2.2.


This completes the proof of carre.1.2.2.


This completes the proof of carre.1.2.


This completes the proof of carre.1.

carre.2 :

  |-------
{1}    determ?(delays(n))
[2]    (fold(n, andg) o tri(n, delay)) o map(n, eqlg) o zip(n)
          o split(fork(n), fork(n) o delays(n))
          = (fold(n, andg o (I * delay)) o fork(n)) o eqlg o split(I, delays(n))

Rule? (rewrite "delays_determ")
Found matching substitution:
n gets n,
Rewriting using delays_determ,

This completes the proof of carre.2.

Q.E.D.
```

This concludes our work on the verification of (part of) the carré problem.

10.3 Conclusions

We have shown how to formalize the theory of relations within the specification language of PVS, which is based on higher order logic. We proved from first principles all the rules of relational calculus, and Ruby, that we needed. We demonstrated how a general-purpose verification system can be used to do proofs in a point-free relational style.

Some of the proofs of the rules of the calculus and lemmas were for the most part conducted by letting the automated proof procedures take care of most of the work, and then leading the theorem prover step-by-step to prove the remaining sequents that it could not prove automatically. The proofs conducted in this manner are not compelling from the point of view of human readers; however they require less labour than full step-by-step proofs; and the theorems for which this strategy works are often the ones that can be easily seen to be correct. Another proof strategy, that leads to proofs that are more readable, is to lead the prover step-by-step until the task is decomposed into sequents that are easy to prove, and then letting the automated proof procedures take care of them.

But the main point of this work is to show that once the needed lemmas are available, the proof of Ruby-style derivations can be done in a fashion that is not too dissimilar to the corresponding paper-and-pencil proof.

One of the main obstacles to the use of systems like PVS is the ASCII notation, that is considerably less readable than mathematical notation (compare the terms that appear in the PVS proof of the carre theorem to the ones that appear in the corresponding derivation in page 60). The main difficulty in following the verification is in the parsing of the terms. PVS has a facility for producing LaTeX versions of proofs and theories; but this can only be used to embellish the presentation of proofs, while all interaction with the system is done in ASCII. Since we already gave a mathematical-style justification of the carré derivation, it seemed better to show the PVS material in the same style as one would see it by working with the system; this may be more useful to the reader who wants to reproduce our work.

It must be noted that PVS offers good assistance in the *verification* of theorems, but very little assistance for the *construction* of solutions. In our view, there are two main advantages in the use of systems like PVS. One is the uncovering of all unstated assumptions (not to mention errors!) in one's work. In this respect, the work with PVS was somewhat disappointing, for no errors were actually found in our work. The other advantage is that the effort to express things in a language, that is more formal and poorer than con-

ventional mathematical language, may lead to simpler definitions and better understanding.

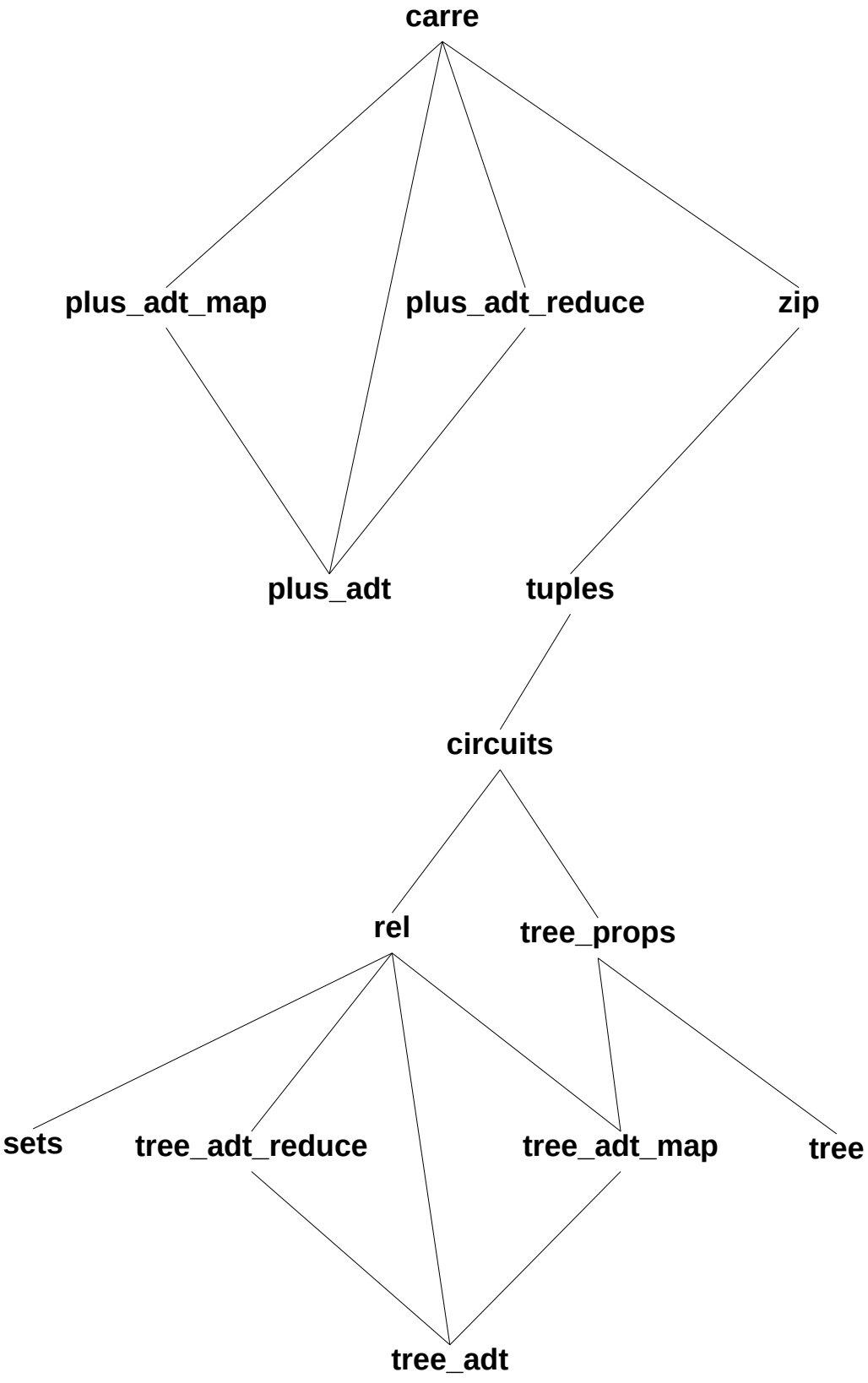


Figure 10.0: The hierarchy of theories by inclusion

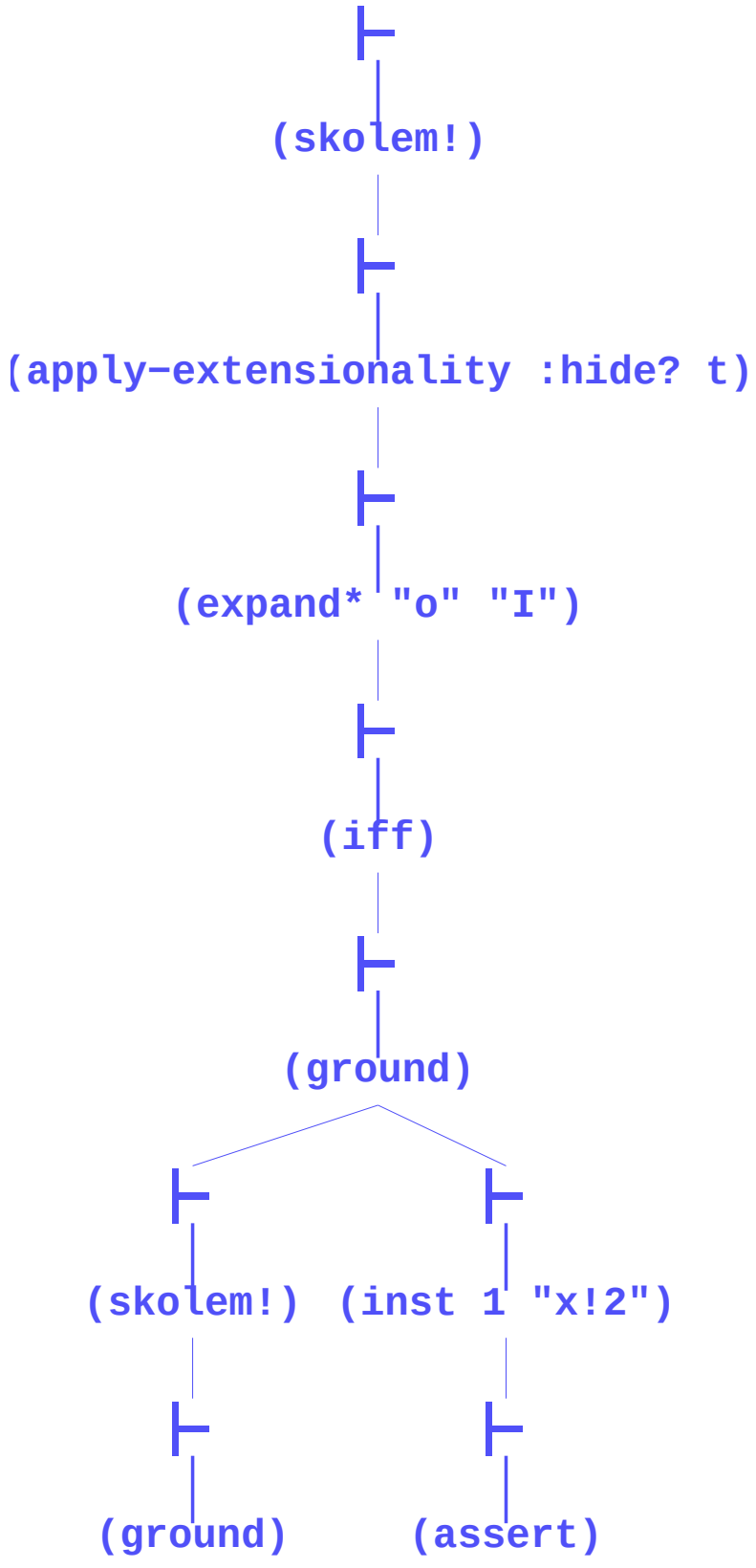


Figure 10.1: The structure of the proof of `id0`

```
(defstep assoc-rewrite (lemma &optional (fnums *)
                              subst (target-fnums *)
                              (dir LR) (order IN))
  (apply
   (then*
    (repeat (rewrite "comp_assoc" :dir rl :target-fnums target-fnums))
    (repeat (then (rewrite lemma
                           :subst subst :fnums fnums :dir dir
                           :target-fnums target-fnums)
                  (rewrite "comp_assoc")))
    (repeat (rewrite "comp_assoc" :dir rl)) ))
  "Applies a rewriting rule modulo associativity of composition"
  "rewriting with ~a modulo associativity" )
```

Figure 10.2: The `assoc-rewrite` proof strategy

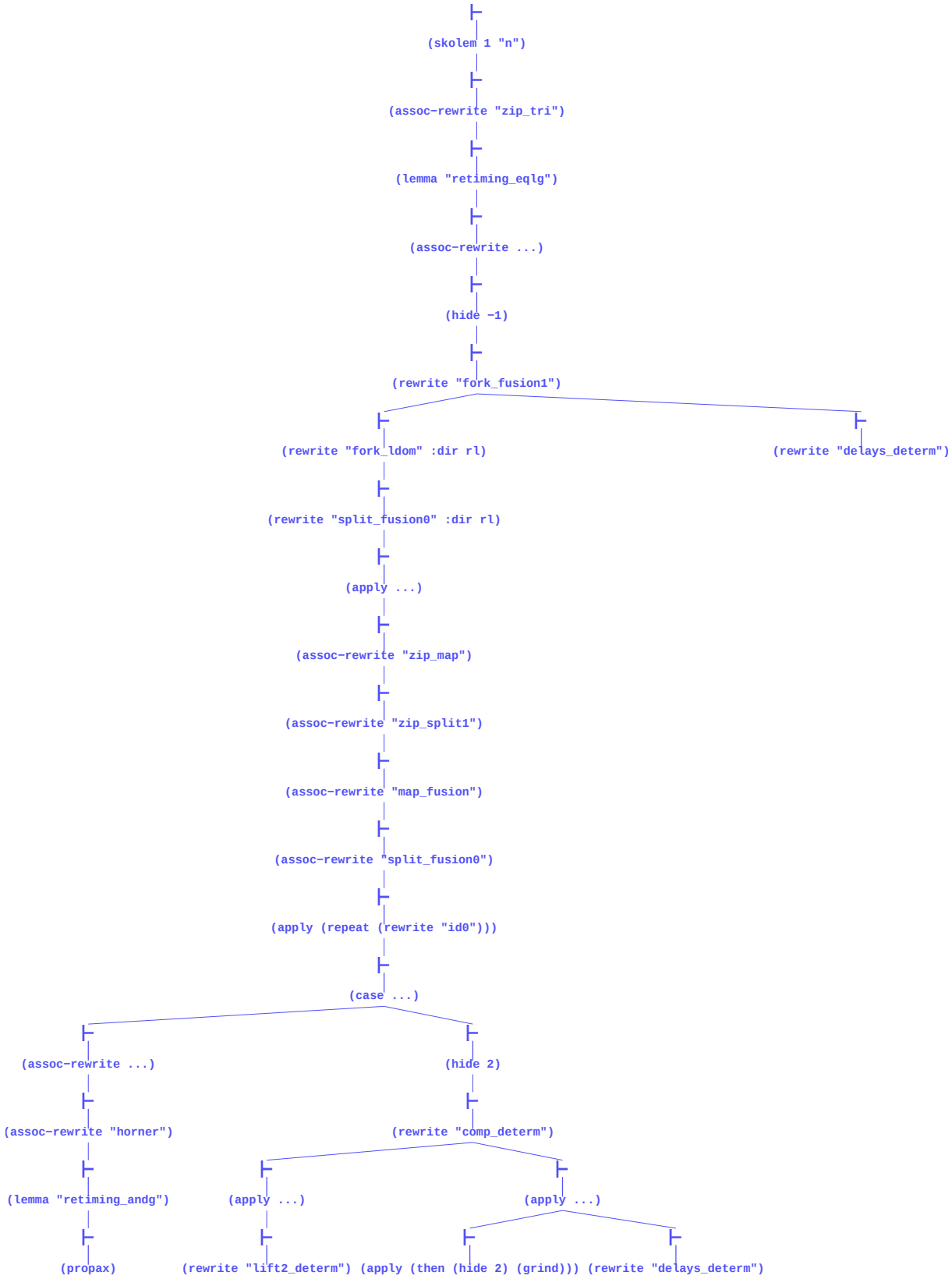


Figure 10.3: The proof tree of the `carre` theorem

Chapter 11

Conclusions and discussion

It is customary to define Ruby circuits as relations between two streams, which may be tuple-valued. In this thesis instead we define circuits as relations between trees of streams; so we may have a relation between, say, two pairs of streams, whereas in Ruby literature one would find a relation between two pair-valued streams. The reason for this different choice is the wish to depart as little as possible from relational calculus as defined in [0]. The standard relational definition of product, as defined in chapter 2, is

$$(11.0) \quad (x,y)\langle R{\times}S\rangle(z,v) \;\equiv\; x\langle R\rangle z \;\wedge\; y\langle S\rangle v.$$

But then the product of two relations is a relation between two *pairs* of values. In Ruby this is taken care of by using this variant definition of product:

$$(11.1) \quad [R,S] \;=\; \widetilde{zip} \circ R{\times}S \circ \widetilde{zip}^{\cup},$$

where $\widetilde{zip}$ is the relation that transforms a pair of streams into a stream of pairs, defined by (3.10) at page 36. A similar thing applies to split. Now, it is perfectly acceptable to build a calculus of circuits that uses (11.1) and doesn't use (11.0). But suppose we want the universe $\mathcal{U}$ to contain not just streams but also other values such as, say, the integers. This is necessary in order to define, e.g., *mem* (section 7.0). Then we have that product and split are no longer defined over all relations, but only on relation between streams. To overcome this undesirable fact, one can then define two products, the ordinary one (11.0) and the product for circuits (11.1). Then one needs two versions of the split operator; then two versions of the projections. This causes a proliferation of operator symbols, which is unfortunate since the relational calculus uses many operators already; and does not change the

problem of (11.1) not being defined over all relations. For this reason we prefer not to use (11.1) and accept the fact that a circuit is a relation between arbitrary pairings (that is trees) of streams.

There is another possible advantage to employing (11.1) for circuits. Sandum [47] notes that some Ruby circuits involving bundle are not implementable. For instance, $[\bar{\iota}, \overline{\mathcal{B}}]$ satisfies

$$\begin{aligned} a\langle[\bar{\iota}, \overline{\mathcal{B}}]\rangle b \;\equiv\; \forall(n :: \quad & \ll.(a.n) \;=\; \ll.(b.n) \\ & \wedge \gg.(a.2n) \;=\; \ll.\gg.(b.n) \\ & \wedge \gg.(a.(2n+1)) \;=\; \gg.\gg.(b.n)), \end{aligned}$$

and any implementation of $[\bar{\iota}, \overline{\mathcal{B}}]$ would require unbounded memory. This is due to the fact that the two devices in parallel are supposed to read a pair-valued stream. Hence they must work in synchrony. But $\bar{\iota}\times\overline{\mathcal{B}}$ reads from a pair of streams, and an implementation where $\bar{\iota}$ and $\overline{\mathcal{B}}$ work asynchronously is possible.

However, it is not clear how to make use of this apparent greater generality. If the two streams read by a device $R\times S$ are read at different speeds, then there should be some way to relate the state of R with the state of S; in other words, we must have a way to define some T that when composed as in $T \circ R\times S$ can make use of the outputs of both R and S. Such a T should be some kind of arbiter, that is able to accept actions from R and S asynchronously. This is a line of investigation that we haven't pursued; it look promising as it hints at a way to define a version of Ruby for asynchronous circuits.

Now we must admit that defining product as in (11.0) bears the disadvantage that many definition are more complicated than what we'd have when defining product by (11.1), notably the definition of delay and bundle; and the proofs of the related properties are similarly more complicated. But this is a problem only with the proofs of the rules of the calculus of Ruby; circuit derivations are largely unaffected by our different choice of in the definition of product, since most rules of the calculus, like fusion, hold both for (11.1) and (11.0).

It is interesting to note that there is at least one calculus equality that does *not* hold for $\times$, while it holds for $[-,-]$: (see the discussion at page 196): it holds

$$\mathcal{B}^{\cup} \circ \mathcal{B} \;=\; [\iota, \iota], \tag{11.2}$$

but not $\mathcal{B}^{\cup} \circ \mathcal{B} = \iota \times \iota$. Given that we wish to avoid (11.1), the best alternative rule we can formulate is

(11.3) $\mathcal{B}^{\cup} \circ \mathcal{B} \circ A = A,$

for A a shape-symmetric monotype; see theorem (B.2) at page 196. It is not clear whether this fact is an argument for or against our style of Ruby, since there is a good reason, in terms of circuits interpretation, for *not* wanting (11.2) to hold. What the circuit on the lhs of (11.2) does is to take a pair of values and sending them one at a time through a single wire, and then recompose the pair. But there is no restriction on the type of the elements of the pair; for instance, the first element may be a scalar wire, while the second is a pair. This makes it difficult to imagine the interpretation of the internal wire that connects the $\mathcal{B}^{\cup}$ component with the $\mathcal{B}$ in the circuit $\mathcal{B}^{\cup} \circ \mathcal{B}$, for it must be a wire able to transmit values of widely differing shapes. On the other hand, theorem (11.3) tells us the much weaker fact that, whenever we can prove that all elements of a pair of wires recursively share the same pairing structure, then it is possible to cancel composition with $\mathcal{B}^{\cup} \circ \mathcal{B}$.

It may be countered that terms like $\mathcal{B}^{\cup} \circ \mathcal{B}$ may arise in the course of calculations, but need never be implemented; and certainly theorem (11.2) is useful for simplifying calculations. In conclusion, whether it is desirable or not to define product so that (11.2) holds is a matter of opinion.

* *
*

The informal picture interpretation of circuit terms that we used is essentially monodimensional: the circuit extends horizontally. This is due to the fact that the relations of our calculus are *binary*, so they are like a circuit component with just two "ports". In contrast, traditional circuit diagrams extend in two dimensions: components may well have connections entering and exiting in all directions.

It is common in Ruby to extend the picture interpretation to two dimensions, by interpreting the second element of the left domain and the first element of the right domain as "vertical". Nevertheless, it's nice to see that a lot of ground can be covered by the simpler monodimensional picture interpretation, at least for the kind of circuits that we consider in this thesis.

A problem with the current state of Ruby is the absence of a textbook. Existing expositions are in the form of lecture notes, and often quote laws without proof. Laws that are often used without proof are the retiming

and slowdown laws. We give full and detailed proofs for these laws in the appendix.

Many Ruby laws are proved to hold for combinational circuits, and then extended to arbitrary circuits by the "lifting theorem", which states that for "timeless" circuits (i.e., a circuit as defined by definition 3.9), many laws can be so extended. We feel no need for such theorem, since we found most Ruby laws can be easily proved by standard relational calculus.

In conclusion, it is difficult to assess the usefulness of a calculus. In this thesis there are no deep or difficult new theorems; on the contrary, we tried to make everything uniformly straightforward. This work is about method and notation: two things of paramount importance in Computing Science. The whole discipline of "programming language design" is about finding convenient notations to express programming concepts.

Much of this work was driven by the desire to see how Ruby could handle problems somewhat different from those one finds in the literature. It is clear that Ruby shines in regular array computations, such as matrix multiplications or convolution filters; but the real limits of Ruby applicability have not yet been clearly identified. A difficulty one often encounters when working with Ruby is that wiring relations can quickly become unmanageable.

In our opinion, the strength of Ruby is in the fact that memory elements are not named; this makes laws like the retiming law or the slowdown theorem very simple to state and apply; in contrast, stating these laws for a notation like Tangram would be difficult. On the other hand, this strength is also a weakness, for the risk of getting bogged down with overly complicated wiring relations. CSP-like notations have their own set of identity laws (Hoare [22]); identifying the overlap and differences between CSP and Ruby laws could be interesting further work.

One area where Ruby could be improved is in the handling of multiplexers, or "if-then-else". In its full generality this kind of component is shunned in Jones [24], for the reason that it enjoys few useful laws. However, the kind of circuits one can obtain without a choice component is limited. A first step in using choice within Ruby is our "junc" component; it could be interesting to try to generalize it, so to have a combinator that is more similar to "junc" as defined by Aarts et al. [0].

It is arguable that computing with relations is a different paradigm that is not nearly fully explored. It seems there are many tricks and techniques and methods yet to be discovered; I look forward with curiosity and anticipation to the results of future research.

Appendix A

Proofs of the delay and retiming laws

This section contains proofs of the properties of delay and, in particular, the retiming laws. The proof of (3.4) is as follows:

$$
\begin{array}{ll}
 & \lhd^< \;=\; \iota \\
\equiv & \qquad \{ \quad \text{definition of } \lhd,\ \iota \quad \} \\
 & (\mu(X \mapsto \partial \cup X \times X))^< \;=\; \mu(X \mapsto \bar{\iota} \cup X \times X) \\
\Leftarrow & \qquad \{ \quad \mu\text{-fusion} \quad \} \\
 & \forall (X ::\ (\partial \cup X \times X)^< \;=\; \bar{\iota} \cup X^< \times X^<) \\
\Leftarrow & \qquad \{ \quad \text{domains distribute through } \cup \text{ and } \times \quad \} \\
 & \partial^< \;=\; \bar{\iota} \;\wedge\; X^< \times X^< \;=\; X^< \times X^< \\
\equiv & \qquad \{ \quad \text{definition of } \partial,\ \bar{\iota} \quad \} \\
 & \mathit{true}
\end{array}
$$

The proof of the first part of (3.2) is as follows:

$$
\begin{array}{ll}
 & \lhd \circ \iota \times \iota \\
= & \qquad \{ \quad \text{definition of } \lhd \quad \} \\
 & (\partial \cup \lhd \times \lhd) \circ \iota \times \iota \\
= & \qquad \{ \quad \text{composition distributes over union} \quad \} \\
 & \partial \circ \iota \times \iota \;\cup\; \lhd \times \lhd \circ \iota \times \iota \\
= & \qquad \{ \quad \partial \text{ is not defined for pairs, (2.1)} \quad \} \\
 & \lhd \times \lhd
\end{array}
$$

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

The next calculation establishes property (3.14).

$$
\begin{array}{ll}
 & \triangleright \circ \triangleleft \;=\; \iota \\
\equiv & \quad \{ \quad \text{definition of } \iota \text{ and } \triangleleft \quad \} \\
 & \triangleright \circ \mu(X \mapsto \partial \cup X{\times}X) \;=\; \mu(X \mapsto \bar{\iota} \cup X{\times}X) \\
\Leftarrow & \quad \{ \quad \mu\text{-fusion} \quad \} \\
 & \forall(X :: \; \triangleright \circ (\partial \cup X{\times}X) \;=\; \bar{\iota} \cup (\triangleright \circ X){\times}(\triangleright \circ X)) \\
\Leftarrow & \quad \{ \quad \text{calculus} \quad \} \\
 & \triangleright \circ \partial \;=\; \bar{\iota} \;\wedge\; \forall(X :: \; \triangleright \circ X{\times}X \;=\; (\triangleright \circ X){\times}(\triangleright \circ X)) \\
\equiv & \quad \{ \quad \text{property (3.3)} \quad \} \\
 & \triangleright \circ \partial \;=\; \bar{\iota} \\
\equiv & \quad \{ \quad \triangleright \;=\; \triangleleft^{\cup} \;=\; (\partial \cup \triangleleft{\times}\triangleleft)^{\cup} \;=\; \partial^{\cup} \cup \triangleright{\times}\triangleright \quad \} \\
 & (\partial^{\cup} \cup \triangleright{\times}\triangleright) \circ \partial \;=\; \bar{\iota} \\
\equiv & \quad \{ \quad \partial \text{ is not defined on pairs} \quad \} \\
 & \partial^{\cup} \circ \partial \;=\; \bar{\iota}
\end{array}
$$

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

$$
\begin{array}{ll}
 & a\langle \partial^{\cup} \circ \partial \rangle b \\
\equiv & \quad \{ \quad \text{composition} \quad \} \\
 & \exists(c :: \; a\langle \partial^{\cup} \rangle c \;\wedge\; c\langle \partial \rangle b) \\
\equiv & \quad \{ \quad \text{definition of primitive delay; calculus} \quad \} \\
 & \exists(c :: \; \forall(n :: \; a.n \;=\; c.(n+1) \;\wedge\; c.(n+1) \;=\; b.n)) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & \forall(n :: \; a.n \;=\; b.n)
\end{array}
$$

The second equality in (3.14) can be proved by means of a very similar proof.

The proof of the retiming laws, equation (3.13) is by structural induction on definition 3.9. We begin by proving that (3.13) holds for any lifted relation $\dot{R}$. For all streams a and b :

$$
\begin{array}{ll}
 & a\langle \triangleleft \circ \dot{R} \rangle b \\
\equiv & \quad \{ \quad \text{composition} \quad \} \\
 & \exists(c :: \; a\langle \triangleleft \rangle c \;\wedge\; c\langle \dot{R} \rangle b)
\end{array}
$$

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

Next we show that (3.13) holds for the left projection. First, we need a little lemma:

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

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,

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

The proof that (3.13) holds for the right projection is entirely similar. That (3.13) holds for $R := \lhd$ is trivial; for $R := \rhd$ it is a consequence of (3.14). For *term* we have:

$$
\begin{array}{ll}
 & \lhd \circ \mathit{term} \\
= & \quad \{ \quad (3.16) \quad \} \\
 & \mathit{term} \\
= & \quad \{ \quad \lhd^{\rhd} = I \quad \} \\
 & \mathit{term} \circ \lhd^{\rhd} \\
= & \quad \{ \quad \top\!\!\top \circ R^{\rhd} = \top\!\!\top \circ R \quad \} \\
 & \mathit{term} \circ \lhd
\end{array}
$$

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 parts of (3.13) hold for lifting, projections, delays and *term*.

Suppose now that (3.13) holds 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:

$$
\begin{array}{ll}
 & \lhd \circ R \times S \\
= & \quad \{ \quad \text{equation (3.3)} \quad \} \\
 & (\lhd \circ R) \times (\lhd \circ S) \\
= & \quad \{ \quad \text{hypothesis on } R \text{ and } S \quad \} \\
 & (R \circ \lhd) \times (S \circ \lhd) \\
= & \quad \{ \quad \text{equation (3.3)} \quad \} \\
 & R \times S \circ \lhd
\end{array}
$$

Similarly, for split:

$$
\begin{array}{ll}
 & \lhd \circ R \vartriangle S \\
= & \quad \{ \quad \text{equation (3.3)} \quad \} \\
 & (\lhd \circ R) \vartriangle (\lhd \circ S) \\
= & \quad \{ \quad \text{hypothesis on } R \text{ and } S \quad \} \\
 & (R \circ \lhd) \vartriangle (S \circ \lhd)
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \quad \text{equation (3.5)} \quad \} \\
& R \vartriangle S \circ \lhd
\end{array}
$$

For converse, we have

$$
\begin{array}{ll}
& \lhd \circ R^{\cup} \\
= & \{ \quad \text{converse; } \rhd \text{ is the converse of } \lhd \quad \} \\
& (R \circ \rhd)^{\cup} \\
= & \{ \quad \text{by hypothesis (3.13) holds for } R \quad \} \\
& (\rhd \circ R)^{\cup} \\
= & \{ \quad \text{converse} \quad \} \\
& R^{\cup} \circ \lhd
\end{array}
$$

The last part of the proof makes use of the properties of loop and feedback, equations (3.8):

$$
\begin{array}{ll}
& \lhd \circ R^{\sigma} \\
= & \{ \quad \text{loop-feedback} \quad \} \\
& \lhd \circ (\iota \vartriangle \iota \circ R)^{\varpi} \\
= & \{ \quad \text{loop fusion, eq. (2.1)} \quad \} \\
& (\lhd \vartriangle \iota \circ R)^{\varpi} \\
= & \{ \quad \text{equations (3.5) and (3.14)} \quad \} \\
& (\iota \vartriangle \rhd \circ \lhd \circ R)^{\varpi} \\
= & \{ \quad \text{hypothesis on } R \quad \} \\
& (\iota \vartriangle \rhd \circ R \circ \lhd)^{\varpi} \\
= & \{ \quad \text{equation (2.1)} \quad \} \\
& (\iota \times \rhd \circ \iota \vartriangle \iota \circ R \circ \lhd)^{\varpi} \\
= & \{ \quad \text{loop leapfrog} \quad \} \\
& (\iota \vartriangle \iota \circ R \circ \lhd \circ \iota \times \rhd)^{\varpi} \\
= & \{ \quad \text{equations (3.3) and (3.14)} \quad \} \\
& (\iota \vartriangle \iota \circ R \circ \lhd \times \iota)^{\varpi} \\
= & \{ \quad \text{loop fusion} \quad \} \\
& (\iota \vartriangle \iota \circ R)^{\varpi} \circ \lhd \\
= & \{ \quad \text{loop-feedback} \quad \} \\
& R^{\sigma} \circ \lhd
\end{array}
$$

This concludes the proof of (3.13).

Appendix B

Facts about bundle and *slow*

An essential property of bundle is

(B.o) $\quad \mathcal{B} \circ \mathcal{B}^{\cup} = \iota$

The proof is:

$$
\begin{array}{ll}
 & \mathcal{B} \circ \mathcal{B}^{\cup} = \iota \\
\equiv & \quad \{ \quad \text{definitions} \quad \} \\
 & \mu(X \mapsto \overline{\mathcal{B}} \cup X{\times}X \circ \mathit{zip}) \circ \mathcal{B}^{\cup} = \mu(X \mapsto \bar{\iota} \cup X{\times}X) \\
\Leftarrow & \quad \{ \quad \mu\text{-fusion} \quad \} \\
 & \forall(X :: (\overline{\mathcal{B}} \cup X{\times}X \circ \mathit{zip}) \circ \mathcal{B}^{\cup} = \bar{\iota} \cup (X \circ \mathcal{B}^{\cup}){\times}(X \circ \mathcal{B}^{\cup}))
\end{array}
$$

and for all X,

$$
\begin{array}{ll}
 & (\overline{\mathcal{B}} \cup X{\times}X \circ \mathit{zip}) \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \mu \text{ computation rule;} \\
 & \qquad \text{converse over union and composition} \quad \} \\
 & (\overline{\mathcal{B}} \cup X{\times}X \circ \mathit{zip}) \circ (\overline{\mathcal{B}}^{\cup} \cup \mathit{zip}^{\cup} \circ \mathcal{B}^{\cup} \times \mathcal{B}^{\cup}) \\
= & \quad \{ \quad \text{distributivity; } \overline{\mathcal{B}} \text{ is not defined on pairs} \quad \} \\
 & \overline{\mathcal{B}} \circ \overline{\mathcal{B}}^{\cup} \cup X{\times}X \circ \mathit{zip} \circ \mathit{zip}^{\cup} \circ \mathcal{B}^{\cup} \times \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{pointwise calculations} \quad \} \\
 & \bar{\iota} \cup X{\times}X \circ \mathcal{B}^{\cup} \times \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & \bar{\iota} \cup (X \circ \mathcal{B}^{\cup}){\times}(X \circ \mathcal{B}^{\cup})
\end{array}
$$

This completes the proof of (B.0).

One could hypothesize that $\mathcal{B}^{\cup} \circ \mathcal{B} \stackrel{?}{=} \iota\times\iota$ should hold; yet it does *not* (see chapter 11). In fact, for any two monotype A, we have that $\mathcal{B}^{\cup} \circ \mathcal{B} \circ A \neq \bot\!\!\!\bot$ only if A has a "symmetric shape". For example, it holds

(B.1) $\quad \mathcal{B}^{\cup} \circ \mathcal{B} \circ \bar{\iota}\times\bar{\iota} \;=\; \bar{\iota}\times\bar{\iota}$

and also

$$\mathcal{B}^{\cup} \circ \mathcal{B} \circ (\bar{\iota}\times\bar{\iota})\times(\bar{\iota}\times\bar{\iota}) \;=\; (\bar{\iota}\times\bar{\iota})\times(\bar{\iota}\times\bar{\iota})$$

but:

$$\begin{array}{ll} & \mathcal{B}^{\cup} \circ \mathcal{B} \circ (\bar{\iota}\times\bar{\iota})\times\bar{\iota} \\ = & \quad \{ \quad \text{definition} \quad \} \\ & \mathcal{B}^{\cup} \circ (\overline{\mathcal{B}} \cup \mathcal{B}\times\mathcal{B} \circ \mathit{zip}) \circ (\bar{\iota}\times\bar{\iota})\times\bar{\iota} \\ = & \quad \{ \quad \text{distributivity; } \overline{\mathcal{B}} \text{ is not defined on } (\bar{\iota}\times\bar{\iota})\times\bar{\iota} \quad \} \\ & \mathcal{B}^{\cup} \circ \mathcal{B}\times\mathcal{B} \circ \mathit{zip} \circ (\bar{\iota}\times\bar{\iota})\times\bar{\iota} \\ = & \quad \{ \quad \text{zip is only defined on pairs of pairs} \quad \} \\ & \bot\!\!\!\bot \end{array}$$

Let's formalize the concept of "having a symmetric shape". Let A be a monotype. We say that A is shape-symmetric, if

- $A \subseteq \bar{\iota}\times\bar{\iota}$, or
- there exist monotypes B and C such that $A = B\times C$ and both B and C are shape-symmetric.

Now we claim that, if A is a shape-symmetric monotype,

(B.2) $\quad \mathcal{B}^{\cup} \circ \mathcal{B} \circ A \;=\; A$

This can be shown by structural induction on the definition of shape symmetry. If $A \subseteq \bar{\iota}\times\bar{\iota}$, then

$$\begin{array}{ll} & \mathcal{B}^{\cup} \circ \mathcal{B} \circ A \\ = & \quad \{ \quad A \subseteq \bar{\iota}\times\bar{\iota} \quad \} \\ & \overline{\mathcal{B}}^{\cup} \circ \overline{\mathcal{B}} \circ A \\ = & \quad \{ \quad \text{by pointwise reasoning} \quad \} \\ & A \end{array}$$

Suppose now A can be written $B{\times}C$, for shape symmetric B and C:

$$
\begin{array}{ll}
 & \mathcal{B}^{\cup} \circ \mathcal{B} \circ B{\times}C \\
= & \quad \{ \quad \text{a shape-symmetric monotype can always} \\
 & \qquad \text{be written as a product of two monotypes:} \\
 & \qquad \text{say, } B = B_0{\times}B_1 \text{ and } C = C_0{\times}C_1 \quad \} \\
 & \mathcal{B}^{\cup} \circ \mathcal{B} \circ (B_0{\times}B_1){\times}(C_0{\times}C_1) \\
= & \quad \{ \quad \text{definition of bundle on pair of pairs} \quad \} \\
 & \mathit{zip} \circ (\mathcal{B}^{\cup} \circ \mathcal{B}){\times}(\mathcal{B}^{\cup} \circ \mathcal{B}) \circ \mathit{zip} \circ (B_0{\times}B_1){\times}(C_0{\times}C_1) \\
= & \quad \{ \quad \text{property of } \mathit{zip} \quad \} \\
 & \mathit{zip} \circ (\mathcal{B}^{\cup} \circ \mathcal{B}){\times}(\mathcal{B}^{\cup} \circ \mathcal{B}) \circ (B_0{\times}C_0){\times}(B_1{\times}C_1) \circ \mathit{zip} \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & \mathit{zip} \circ (\mathcal{B}^{\cup} \circ \mathcal{B} \circ B_0{\times}C_0){\times}(\mathcal{B}^{\cup} \circ \mathcal{B} \circ B_1{\times}C_1) \circ \mathit{zip} \\
= & \quad \{ \quad B_0{\times}C_0 \text{ and } B_1{\times}C_1 \text{ must be shape-symmetric;} \\
 & \qquad \text{by induction} \quad \} \\
 & \mathit{zip} \circ (B_0{\times}C_0){\times}(B_1{\times}C_1) \circ \mathit{zip} \\
= & \quad \{ \quad \text{property of } \mathit{zip} \quad \} \\
 & (B_0{\times}B_1){\times}(C_0{\times}C_1) \\
= & \quad \{ \quad \text{by the definitions of } B_0,\ B_1,\ C_0 \text{ and } C_1 \quad \} \\
 & B{\times}C
\end{array}
$$

This concludes the proof of (B.2).

One interesting theorem about $\mathcal{B}$ is the following:

$$\mathcal{B} \circ \lhd = \lhd \circ \lhd \circ \mathcal{B}$$

Proof: let's begin by supposing the left domain is not a pair. We have:

$$
\begin{array}{ll}
 & a\langle \overline{\mathcal{B}} \circ \partial{\times}\partial \rangle (b, c) \\
\equiv & \quad \{ \quad \text{definitions of bundle, primitive delay and composition} \quad \} \\
 & \exists (d, e :: \quad \forall (n :: \ a.2n = d.n \ \wedge \ a.(2n+1) = e.n) \\
 & \qquad \wedge \forall (n :: \ d.n = b.(n-1) \ \wedge \ e.n = c.(n-1))) \\
\equiv & \quad \{ \quad \text{single-point rule} \quad \} \\
 & \forall (n :: \ a.2n = b.(n-1) \ \wedge \ a.(2n+1) = c.(n-1)) \\
\equiv & \quad \{ \quad \text{change of dummy: } n := m+1 \quad \} \\
 & \forall (m :: \ a.(2m+2) = b.m \ \wedge \ a.(2m+3) = c.m)
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{single-point rule} \quad \} \\
& \exists(d :: \quad \forall(n :: \ a.n \ = \ d.(n-2)) \\
& \qquad \wedge \forall(m :: \ d.2m \ = \ b.m \ \wedge \ d.(2m+1) \ = \ c.m)) \\
\equiv & \{ \quad \text{definitions of bundle, primitive delay and composition} \quad \} \\
& a\langle \partial \circ \partial \circ \overline{\mathcal{B}} \rangle (b, c)
\end{array}
$$

Hence, since $\partial \circ \overline{\mathcal{B}} = \lhd \circ \overline{\mathcal{B}}$ and $\overline{\mathcal{B}} \circ \partial \times \partial = \overline{\mathcal{B}} \circ \lhd$, we have

(B.3) $\quad \lhd \circ \lhd \circ \overline{\mathcal{B}} = \overline{\mathcal{B}} \circ \lhd$

Armed with this lemma, we return to the general proof. We will try to find some function H such that

(B.4) $\quad \lhd \circ \lhd \circ \mathcal{B} = \mu H = \mathcal{B} \circ \lhd$

The first part of this equation expands as follows:

$$
\begin{array}{ll}
& \lhd \circ \lhd \circ \mathcal{B} = \mu H \\
\equiv & \{ \quad \text{definition} \quad \} \\
& \lhd \circ \lhd \circ \mu(X \ \mapsto \ \overline{\mathcal{B}} \cup X \times X \circ \mathit{zip}) \ = \ \mu H \\
\Leftarrow & \{ \quad \mu\text{-fusion} \quad \} \\
& \forall(X :: \ \lhd \circ \lhd \circ (\overline{\mathcal{B}} \cup X \times X \circ \mathit{zip}) = H.(\lhd \circ \lhd \circ X)) \\
\equiv & \{ \quad \text{calculus} \quad \} \\
& \forall(X :: \ \lhd \circ \lhd \circ \overline{\mathcal{B}} \cup \lhd \circ \lhd \circ X \times X \circ \mathit{zip} \ = \ H.(\lhd \circ \lhd \circ X))
\end{array}
$$

and the second part is

$$
\begin{array}{ll}
& \mathcal{B} \circ \lhd = \mu H \\
\equiv & \{ \quad \text{definition} \quad \} \\
& \mu(X \ \mapsto \ \overline{\mathcal{B}} \cup X \times X \circ \mathit{zip}) \circ \lhd \ = \ \mu H \\
\Leftarrow & \{ \quad \mu\text{-fusion} \quad \} \\
& \forall(X :: \ (\overline{\mathcal{B}} \cup X \times X \circ \mathit{zip}) \circ \lhd = H.(X \circ \lhd)) \\
\equiv & \{ \quad \text{calculus} \quad \} \\
& \forall(X :: \ \overline{\mathcal{B}} \circ \lhd \cup X \times X \circ \mathit{zip} \circ \lhd \ = \ H.(X \circ \lhd))
\end{array}
$$

Let's define

$$H.X \ = \ \overline{\mathcal{B}} \circ \lhd \cup X \times X \circ \mathit{zip}$$

We obtain, for all X:

$$
\begin{array}{ll}
 & H.(X \circ \lhd) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & (X \circ \lhd)\times(X \circ \lhd) \circ zip \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & X\times X \circ \lhd\times\lhd \circ zip \\
= & \quad \{ \quad zip \text{ is defined on pairs of pairs} \quad \} \\
 & X\times X \circ (\lhd\times\lhd)\times(\lhd\times\lhd) \circ zip \\
= & \quad \{ \quad \text{pointwise reasoning} \quad \} \\
 & X\times X \circ zip \circ (\lhd\times\lhd)\times(\lhd\times\lhd) \\
= & \quad \{ \quad \text{delay} \quad \} \\
 & X\times X \circ zip \circ \lhd
\end{array}
$$

and

$$
\begin{array}{ll}
 & H.(\lhd \circ \lhd \circ X) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & (\lhd \circ \lhd \circ X)\times(\lhd \circ \lhd \circ X) \circ zip \\
= & \quad \{ \quad \text{fusion} \quad \} \\
 & \lhd\times\lhd \circ \lhd\times\lhd \circ X\times X \circ zip \\
= & \quad \{ \quad \text{delay} \quad \} \\
 & \lhd \circ \lhd \circ X\times X \circ zip
\end{array}
$$

These last two calculations, together with (B.3), prove (B.4).

Next we see a very useful theorem: *slow* distributes through product, split, composition, converse, loop and feedback.

$$
\begin{array}{lcl}
slow.(R \circ S) & = & slow.R \circ slow.S \\
slow.(R\times S) & = & slow.R \times slow.S \\
slow.(R \vartriangle S) & = & slow.R \vartriangle slow.S \\
slow.(R^{\cup}) & = & (slow.R)^{\cup} \\
slow.(R^{\varpi}) & = & (slow.R)^{\varpi} \\
slow.(R^{\sigma}) & = & (slow.R)^{\sigma}
\end{array} \tag{B.5}
$$

The proof for composition is:

$$
\begin{array}{ll}
 & slow.R \circ slow.S \\
= & \quad \{ \quad \text{definition} \quad \}
\end{array}
$$

$$
\begin{array}{ll}
 & \mathcal{B} \circ R{\times}R \circ \mathcal{B}^{\cup} \circ \mathcal{B} \circ S{\times}S \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{(B.2)} \quad \} \\
 & \mathcal{B} \circ R{\times}R \circ S{\times}S \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{identity, fusion} \quad \} \\
 & \mathcal{B} \circ (R \circ S){\times}(R \circ S) \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \mathit{slow}.(R \circ S)
\end{array}
$$

For product, we have:

$$
\begin{array}{ll}
 & \mathit{slow}.(R{\times}S) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \mathcal{B} \circ (R{\times}S){\times}(R{\times}S) \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \mathcal{B} \text{ on pairs of pairs} \quad \} \\
 & \mathcal{B}{\times}\mathcal{B} \circ \mathit{zip} \circ (R{\times}S){\times}(R{\times}S) \circ \mathit{zip} \circ \mathcal{B}^{\cup} \times \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{properties of } \mathit{zip} \text{ (3.12)} \quad \} \\
 & \mathcal{B}{\times}\mathcal{B} \circ (R{\times}R){\times}(S{\times}S) \circ \mathcal{B}^{\cup} \times \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{fusion; definition} \quad \} \\
 & \mathit{slow}.R \times \mathit{slow}.S
\end{array}
$$

For split, it is enough to show that $\mathit{slow}.(\iota \vartriangle \iota) = \iota \vartriangle \iota$; the result then follows from the distributivity of *slow* through composition and product.

$$
\begin{array}{ll}
 & \mathit{slow}.(\iota \vartriangle \iota) \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \mathcal{B} \circ (\iota \vartriangle \iota){\times}(\iota \vartriangle \iota) \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \mathcal{B} \text{ on pairs of pairs} \quad \} \\
 & \mathcal{B}{\times}\mathcal{B} \circ \mathit{zip} \circ (\iota \vartriangle \iota){\times}(\iota \vartriangle \iota) \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{properties of } \mathit{zip} \text{ (3.12)} \quad \} \\
 & \mathcal{B}{\times}\mathcal{B} \circ (\iota{\times}\iota) \vartriangle (\iota{\times}\iota) \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \mathcal{B}^{\cup} \text{ is deterministic; fusion} \quad \} \\
 & (\mathcal{B} \circ \iota{\times}\iota \circ \mathcal{B}^{\cup}) \vartriangle (\mathcal{B} \circ \iota{\times}\iota \circ \mathcal{B}^{\cup}) \\
= & \quad \{ \quad \text{identity; equation (B.0)} \quad \} \\
 & \iota \vartriangle \iota
\end{array}
$$

For converse,

$$
\begin{array}{ll}
 & \mathit{slow}.R^{\cup} \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \mathcal{B} \circ R^{\cup} \times R^{\cup} \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{converse over product} \quad \} \\
 & \mathcal{B} \circ (R \times R)^{\cup} \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{converse is idempotent} \quad \} \\
 & (\mathcal{B} \circ (R \times R)^{\cup} \circ \mathcal{B}^{\cup})^{\cup\cup} \\
= & \quad \{ \quad \text{converse over composition} \quad \} \\
 & (\mathcal{B}^{\cup\cup} \circ (R \times R)^{\cup\cup} \circ \mathcal{B}^{\cup})^{\cup} \\
= & \quad \{ \quad \text{idempotency, twice} \quad \} \\
 & (\mathcal{B} \circ R \times R \circ \mathcal{B}^{\cup})^{\cup} \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & (\mathit{slow}.R)^{\cup}
\end{array}
$$

For loop,

$$
\begin{array}{ll}
 & \mathit{slow}.R^{\varpi} \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & \mathcal{B} \circ R^{\varpi} \times R^{\varpi} \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{loop (3.8)} \quad \} \\
 & \mathcal{B} \circ (\mathit{zip} \circ R \times R \circ \mathit{zip})^{\varpi} \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{loop fusion (3.8)} \quad \} \\
 & (\mathcal{B} \times \iota \circ \mathit{zip} \circ R \times R \circ \mathit{zip} \circ \mathcal{B}^{\cup} \times \iota)^{\varpi} \\
= & \quad \{ \quad \mathit{zip} \text{ is defined on pairs of pairs} \quad \} \\
 & (\mathcal{B} \times \iota \circ \iota \times (\iota \times \iota) \circ \mathit{zip} \circ R \times R \circ \mathit{zip} \circ \mathcal{B}^{\cup} \times \iota)^{\varpi} \\
= & \quad \{ \quad \text{lemma (B.2)} \quad \} \\
 & (\mathcal{B} \times \iota \circ \iota \times (\mathcal{B}^{\cup} \circ \mathcal{B} \circ \iota \times \iota) \circ \mathit{zip} \circ R \times R \circ \mathit{zip} \circ \mathcal{B}^{\cup} \times \iota)^{\varpi} \\
= & \quad \{ \quad \text{fusion, identity} \quad \} \\
 & (\mathcal{B} \times (\mathcal{B}^{\cup} \circ \mathcal{B}) \circ \mathit{zip} \circ R \times R \circ \mathit{zip} \circ \mathcal{B}^{\cup} \times \iota)^{\varpi} \\
= & \quad \{ \quad \text{fusion, loop leapfrog (3.8)} \quad \} \\
 & (\mathcal{B} \times \mathcal{B} \circ \mathit{zip} \circ R \times R \circ \mathit{zip} \circ \mathcal{B}^{\cup} \times \mathcal{B}^{\cup})^{\varpi} \\
= & \quad \{ \quad R \text{ is defined on pairs} \quad \} \\
 & (\mathcal{B} \circ R \times R \circ \mathcal{B}^{\cup})^{\varpi} \\
= & \quad \{ \quad \text{definition} \quad \} \\
 & (\mathit{slow}.R)^{\varpi}
\end{array}
$$

For feedback,

$$
\begin{array}{ll}
 & slow.R^{\sigma} \\
= & \quad \{ \quad \text{loop-feedback (3.8)} \quad \} \\
 & slow.(\iota \vartriangle \iota \circ R)^{\varpi} \\
= & \quad \{ \quad \text{distributivity of } slow \text{ (see above)} \quad \} \\
 & (\iota \vartriangle \iota \circ slow.R)^{\varpi} \\
= & \quad \{ \quad \text{loop-feedback (3.8)} \quad \} \\
 & (slow.R)^{\sigma}
\end{array}
$$

This concludes the proof of (B.5).

The main theorem about *slow* is that, for R a circuit, $slow.R$ is equal to R with all delays doubled. The proof is by induction on definition 3.9. We begin with delay; this is a corollary of theorem (B.4):

$$
\begin{array}{ll}
 & \mathcal{B} \circ \lhd\times\lhd \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{delays and (B.4)} \quad \} \\
 & \lhd \circ \lhd \circ \mathcal{B} \circ \mathcal{B}^{\cup} \\
= & \quad \{ \quad \text{equation (B.0)} \quad \} \\
 & \lhd \circ \lhd
\end{array}
$$

We continue with a lifted relation.

$$
\begin{array}{ll}
 & a\langle \dot{R} \circ \overline{\mathcal{B}} \rangle (b, c) \\
\equiv & \quad \{ \quad \text{composition} \quad \} \\
 & \exists (d ::\ a\langle \dot{R} \rangle d \ \wedge\ d\langle \overline{\mathcal{B}} \rangle (b, c)) \\
\equiv & \quad \{ \quad \text{definition} \quad \} \\
 & \exists (d ::\ a\langle \dot{R} \rangle d \ \wedge\ \forall (n ::\ b.n = d.(2n) \ \wedge\ c.n = d.(2n+1))) \\
\equiv & \quad \{ \quad \text{definition of lifted relation} \quad \} \\
 & \exists (d ::\ \ \forall (n ::\ \ b.n = d.(2n) \ \wedge\ c.n = d.(2n+1) \\
 & \qquad\quad \wedge \qquad a.(2n)\ \langle R \rangle\ b.n \ \wedge\ a.(2n+1)\ \langle R \rangle\ c.n)) \\
\equiv & \quad \{ \quad \text{single-point rule} \quad \} \\
 & \forall (n ::\ a.(2n)\ \langle R \rangle\ b.n \ \wedge\ a.(2n+1)\ \langle R \rangle\ c.n)
\end{array}
$$

and

$$
\begin{array}{ll}
 & a\langle\overline{\mathcal{B}} \circ \dot{R}{\times}\dot{R}\rangle(b,c) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & \exists(d,e ::\ a\langle\overline{\mathcal{B}}\rangle(d,e) \ \wedge\ (d,e)\langle\dot{R}{\times}\dot{R}\rangle(b,c)) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & \exists(d,e ::\ \forall(n ::\ d.n = a.2n \ \wedge\ e.n = a.(2n+1) \ \wedge\ d.n\,\langle R\rangle\, b.n \ \wedge\ e.n\,\langle R\rangle\, c.n)) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & \forall(n ::\ a.(2n)\,\langle R\rangle\, b.n \ \wedge\ a.(2n+1)\,\langle R\rangle\, c.n)
\end{array}
$$

Hence, $\dot{R} \circ \overline{\mathcal{B}} = \overline{\mathcal{B}} \circ \dot{R}{\times}\dot{R}$. A similar proof establishes the same result for a two-inputs lifted relation (i.e., a relation $\dot{R}$ such that $\dot{R} = \dot{R} \circ \iota{\times}\iota$).

The case of the identity relation is trivial, given equation (B.0):

$$\mathcal{B} \circ \iota{\times}\iota \circ \mathcal{B}^{\cup} = \mathcal{B} \circ \mathcal{B}^{\cup} = \iota$$

Next we see the left projection: for a through e arbitrary pairings of streams,

$$
\begin{array}{ll}
 & a\langle{\ll} \circ \mathcal{B}\rangle((b,c),(d,e)) \\
\equiv & \quad \{ \quad \text{composition} \quad \} \\
 & \exists(f,g ::\ a\langle{\ll}\rangle(g,f) \ \wedge\ (g,f)\langle\mathcal{B}\rangle((b,c),(d,e))) \\
\equiv & \quad \{ \quad \text{projection} \quad \} \\
 & \exists(f ::\ (a,f)\langle\mathcal{B}\rangle((b,c),(d,e))) \\
\equiv & \quad \{ \quad \text{definition of } \mathcal{B} \quad \} \\
 & \exists(f ::\ (a,f)\langle\mathcal{B}{\times}\mathcal{B} \circ \mathit{zip}\rangle((b,c),(d,e))) \\
\equiv & \quad \{ \quad \text{definition of } \mathit{zip} \quad \} \\
 & \exists(f ::\ (a,f)\langle\mathcal{B}{\times}\mathcal{B}\rangle((b,d),(c,e))) \\
\equiv & \quad \{ \quad \text{definition of product} \quad \} \\
 & \exists(f ::\ a\langle\mathcal{B}\rangle(b,d) \ \wedge\ f\langle\mathcal{B}\rangle(c,e)) \\
\equiv & \quad \{ \quad \mathcal{B} \text{ is deterministic, hence } f \text{ is uniquely determined} \quad \} \\
 & a\langle\mathcal{B}\rangle(b,d) \\
\equiv & \quad \{ \quad \text{calculus} \quad \} \\
 & a\langle\mathcal{B} \circ {\ll}{\times}{\ll}\rangle((b,c),(d,e))
\end{array}
$$

For *term* we have, for arbitrary collections of wires a,b,c and d:

$$(a,b)\langle\mathcal{B} \circ \mathit{term}{\times}\mathit{term}\rangle(c,d)$$

$$
\begin{array}{ll}
\equiv & \{ \quad \text{composition, definition of } \mathit{term} \quad \} \\
& \exists(e, f :: \ (a, b)\langle \mathcal{B} \rangle((e, e), (f, f))) \\
\equiv & \{ \quad \text{definition of } \mathcal{B} \quad \} \\
& \exists(e, f :: \ (a, b)\langle \mathcal{B} \times \mathcal{B} \circ \mathit{zip} \rangle((e, e), (f, f))) \\
\equiv & \{ \quad \text{definition of } \mathit{zip} \quad \} \\
& \exists(e, f :: \ (a, b)\langle \mathcal{B} \times \mathcal{B} \rangle((e, f), (e, f))) \\
\equiv & \{ \quad \text{product} \quad \} \\
& \exists(e, f :: \ a\langle \mathcal{B} \rangle(e, f) \ \wedge \ b\langle \mathcal{B} \rangle(e, f)) \\
\equiv & \{ \quad \mathcal{B} \text{ is deterministic} \quad \} \\
& a = b \\
\equiv & \{ \quad \text{single-point rule} \quad \} \\
& \exists(e :: \ a = b \ \wedge \ e\langle \mathcal{B} \rangle(c, d)) \\
\equiv & \{ \quad \text{definition of } \mathit{term} \quad \} \\
& \exists(e :: \ (a, b)\langle \mathit{term} \rangle e \ \wedge \ e\langle \mathcal{B} \rangle(c, d)) \\
\equiv & \{ \quad \text{composition} \quad \} \\
& (a, b)\langle \mathit{term} \circ \mathcal{B} \rangle(c, d)
\end{array}
$$

So much for the base of the induction over definition 3.9. The induction step is an immediate consequence of theorem (B.5).

Bibliography

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

[1] Roland Backhouse. Making formality work for us. *EATCS Bulletin*, 38:219–249, June 1989.

[2] Roland Backhouse. The calculational method. *Information Processing Letters*, 53(3):121, 1995.

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

[4] Roland C. Backhouse. *Program Construction and Verification*. Prentice-Hall, 1986.

[5] John Backus. Can programming be liberated from von Neumann's style? In *ACM Turing Award Lectures. The First Twenty Years. 1966-1985*. ACM Press, 1987.

[6] Richard S. Bird. A calculus of functions for program derivation. In D.A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series. Addison-Wesley, 1988.

[7] Richard S. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall International, 1996.

[8] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979.

[9] C. Brink and G. Schmidt, editors. *Relational Methods in Computer Science*. Springer, 1997.

[10] Avra Cohn and Mike Gordon. A mechanized proof of correctness of a simple counter. In K. McEvoy and J.V. Tucker, editors, *Theoretical Foundations of VLSI Design*, Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[11] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[12] Edsger W. Dijkstra, editor. *Formal Development of Programs and Proofs*. University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

[13] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Texts and monographs in Computer Science. Springer-Verlag, 1990.

[14] Henk Doornbos, Roland Backhouse, and Jaap van der Woude. A calculational approach to mathematical induction. *Theoretical Computer Science*, 179(1–2):103–135, 1 June 1997.

[15] Henk Doornbos, Netty van Gasteren, and Roland Backhouse. Programs and datatypes. In C. Brink and G. Schmidt, editors, *Relational Methods in Computer Science*. Springer, 1997.

[16] W.H.J. Feijen and Lex Bijlsma. Exercises in formula manipulation. In Edsger W. Dijkstra, editor, *Formal Development of Programs and Proofs*, University of Texas at Austin Year of Programming Series. Addison-Wesley, 1990.

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

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

[19] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[20] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

[21] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[22] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, 1987.

[23] Graham Hutton. *Between Functions and Relations in Calculating Programs*. PhD thesis, Dept. of Computer Science, University of Glasgow, June 1993. Available at `http://www.cs.nott.ac.uk/Department/Staff/gmh`.

[24] Geraint Jones. Designing circuits by calculation. Technical Report PRG-TR-10-90, Programming Research Group, Oxford University Computing Laboratory, April 1990. Available at `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications`.

[25] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design. IFIP WG 10.5 Lecture Notes*. North-Holland, 1990. A revised version is available at `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications`.

[26] Geraint Jones and Mary Sheeran. Deriving bit-serial circuits in Ruby. In Arne Halaas and Peter B. Denyer, editors, *IFIP Transactions A-1, VLSI 91*. North-Holland, 1992. Available at `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications`.

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

[28] Randy H. Katz. *Contemporary Logic Design*. Addison-Wesley, 1994.

[29] Thomas Kropf. IFIP WG10.5 benchmark circuits for hardware verification. Available at `http://goethe.ira.uka.de/hvg`, 1996.

[30] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufman, 1992.

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

[32] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–63, 1983.

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

[34] Wayne Luk. Specifying and developing regular heterogeneous designs. In Luc J. M. Claesen, editor, *Formal VLSI Specification and Synthesis. VLSI Design Methods-I*. North-Holland, 1990.

[35] Wayne Luk. Systematic serialisation of array-based architectures. *Integration*, 14(3):333–360, 1993. Available at `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Wayne.Luk`.

[36] Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[37] Lambert Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings CWI Symposium on Mathematics and Computer Science*, number 1 in CWI Monographs, pages 289–334. North-Holland, 1986.

[38] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

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

[40] Sam Owre, John Rushby, and Natarajan Shankar. PVS: a prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, number 607 in LNCS. Springer-Verlag, 1992.

[41] Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 1(12):87–107, 1996. Available at `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Ian.Page`.

[42] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[43] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[44] O. Rasmussen. A Ruby proof system. Technical Report ID–TR: 1995-161, Dept. of Computer Science, Technical University of Denmark, December 1995.

[45] Martin Rem. The carré problem. In *Simplex Sigillum Veri, Een Liber Amicorum voor prof. dr. F.E.J. Kruseman Aretz*, pages 279–284. Technische Universiteit Eindhoven, December 1995.

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

[47] 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`.

[48] Frits D. Schalij. Tangram manual. Technical Report Nat. Lab. Technical Note Nr. UR 008/93, Philips Electronics N. V., 1996.

[49] Robin Sharp and Ole Rasmussen. An introduction to Ruby. 2nd edition. Technical report, Dept. of Computer Science, Technical University of Denmark, 1995. Available at `ftp://ftp.it.dtu.dk/pub/Ruby/intro.ps.Z`.

[50] Matteo Vaccari and Roland Backhouse. Deriving a systolic regular language recognizer. In Richard Bird and Lambert Meertens, editors, *Algorithmic Languages and Calculi*. Chapman & Hall, 1997.

[51] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[52] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. Nat. Lab. Technical Note Nr. UR 005/94, Philips Electronics N.V., 1994.

[53] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments*. Number 445 in LNCS. Springer-Verlag, 1990.

Index