Between Functions and Relations in Calculating Programs

Graham Muir Hutton

A thesis submitted for the degree of Ph.D. at the
Department of Computing Science,
University of Glasgow

16th October 1992

Abstract

This thesis is about the *calculational* approach to programming, in which one derives programs from specifications. One such calculational paradigm is *Ruby*, the relational calculus developed by Jones and Sheeran for describing and designing circuits. We identify two shortcomings with derivations made using Ruby. The first is that the notion of a program being an implementation of a specification has never been made precise. The second is to do with types. Fundamental to the use of type information in deriving programs is the idea of having types as special kinds of programs. In Ruby, types are partial equivalence relations (pers). Unfortunately, manipulating some formulae involving types has proved difficult within Ruby. In particular, the preconditions of the 'induction' laws that are much used within program derivation often work out to be assertions about types; such assertions have typically been verified either by informal arguments or by using predicate calculus, rather than by applying algebraic laws from Ruby.

In this thesis we address both of the shortcomings noted above. We define what it means for a Ruby program to be an implementation, by introducing the notion of a *causal* relation, and the *network* denoted by a Ruby program. A relation is causal if it is functional in some structural way, but not necessarily from domain to range; a network captures the connectivity between the primitive relations in a program. Moreover, we present an interpreter for Ruby programs that are implementations. Our technique for verifying an assertion about types is to first express it using operators that give the *best* left and right types for a relation, and then verify this assertion by using algebraic properties of these operators.

Contents

Chapter 1

Introduction

In the *calculational* approach to programming, one derives programs from specifications. There are many calculational paradigms; examples include the *fold/unfold* style of Burstall and Darlington [14], the imperative *refinement calculus* of Back, Morgan, Morris and others [1, 52, 53], the functional *Squiggol* of Bird and Meertens [8, 49], the relational *Ruby* of Jones and Sheeran [41, 37], and recently, the categorical approach of Bird and de Moor [20, 9]. In this thesis we work with Ruby.

Ruby is a relational language that is used for designing programs that represent hardware circuits. Ruby programs denote binary relations, and programs are built-up inductively from primitive relations using a pre-defined set of operators. Ruby programs also have a pictorial interpretation using boxes and wires, which is important when circuit layout is considered in circuit design. The Ruby approach to circuit design is to derive executable terms from specifications in the following way. We first formulate a term that clearly expresses the desired relationships between inputs and outputs, but typically has no direct translation as a circuit. We then transform this term using algebraic laws for the operators of Ruby, aiming towards a term that does represent a circuit. There are several reasons why Ruby is based upon a calculus of relations rather than a calculus of functions. Relational languages offer a rich set of operators and laws for combining and transforming terms, and afford a natural treatment of non-determinism in specifications. Furthermore, many methods of combining circuits (viewed as networks of functions) are unified if the distinction between input and output is removed [58]. Ruby has been successfully used in the design of many different kinds of circuits, including systolic arrays [59], arithmetic circuits [43], and butterfly networks [60].

We identify two shortcomings with derivations made using Ruby. The first is that the notion of a program being an implementation of a specification has never been made precise. The second is to do with types. Fundamental to the use of type information in deriving programs is the idea of having types as special kinds of programs. In Ruby, types are partial equivalence relations (pers) [43, 38]. Unfortunately, manipulating some formulae involving types has proved difficult within Ruby. In particular, the preconditions of the 'induction' laws that are much used within program derivation often work out to be assertions about types; such assertions have typically been verified either by informal arguments or by using predicate calculus, rather than by applying algebraic laws from Ruby.

In this thesis we address both of the shortcomings noted above. We define what it means for a Ruby program to be an implementation, by introducing the notion of a *causal* relation, and the *network* denoted by a Ruby program. A relation is causal if it is functional in some structural way, but not necessarily from domain to range; a network captures the connectivity between the primitive relations in a program. Moreover, we present an interpreter for Ruby programs that are implementations. Our technique for verifying an assertion about types is to express it using operators that give the *best* left and right types for a relation, and then verify this assertion by using algebraic properties of these operators.

The thesis is structured as follows. We begin in chapter 2 with a brief survey of work that is related to our use of a relational language in deriving programs. Chapter 3 introduces our style for calculations, some useful concepts from lattice theory, the basic algebra of relations, and the relational language Ruby. In chapter 4 we introduce the notion of a causal relation, and the network denoted by a Ruby program, and define when a Ruby program is an implementation. In chapter 5 we present an interpreter for a natural sub-class of the implementations. Chapter 6 introduces the idea of pers as types; also introduced here are the *difunctional* relations, which generalise pers in a natural way. Finally, in chapter 7 everything is put into practice: we present a number of program derivations in Ruby, and run the resulting implementations on the interpreter; pers and difunctionals are much in evidence in our derivations. The thesis concludes in chapter 8 with a summary of our achievements, and some directions for future research.

The title “Between Functions and Relations in Calculating Programs” of this thesis reflects that causal relations and difunctional relations can both be viewed as natural generalisations of the notion of a functional relation.

Acknowledgements

Over the last 3 years, I have benefitted greatly from contact with a number of people. I am grateful in particular to Roland Backhouse, Carolyn Brown, Maarten Fokkinga, Netty van Gasteren, John Hughes, Geraint Jones, John Launchbury, Erik Meijer, David Murphy, Ed Voermans, and Jaap van der Woude. The hospitality of Backhouse's group in Eindhoven during my two visits is much appreciated. I would like to thank my thesis examiners (John Launchbury, Simon Peyton Jones, Colin Runciman) for pointing out errors and omissions, and for suggesting a number of improvements. Special thanks are due to my family, and to my supervisor Mary Sheeran. The work reported in this thesis was funded by SERC projects 'relational programming' and 'structured parallel programming'.

Authors current address: Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden. Email: graham@cs.chalmers.se.

Chapter 2

Survey

In this chapter we briefly survey some work that is related to our use of relational languages in deriving programs. First some notation. Recall that a (binary) relation on a universe $\mathcal{U}$ is a set of pairs of elements of $\mathcal{U}$. Throughout this thesis, letters $R, S, T, \ldots$ denote relations. If R is a relation, then $a\ R\ b$ means $(a, b) \in R$, and the *domain* and *range* of R are the sets defined by $dom\ R = \{a \mid \exists b.\ a\ R\ b\}$ and $rng\ R = \{b \mid \exists a.\ a\ R\ b\}$. Inclusion $\subseteq$, equality $=$, union $\cup$, intersection $\cap$, and complement $\neg$ are defined for relations just as for any other sets. We write $\perp\!\!\perp$ for the empty relation $\emptyset$, $\top\!\!\top$ for the full relation $\mathcal{U} \times \mathcal{U}$, and *id* for the identity (or diagonal) relation $\{(a, a) \mid a \in \mathcal{U}\}$. (Writing $\top\!\!\top$ rather than the more usual lattice-theoretic symbol $\top$ avoids confusion with the use of T to denote an arbitrary relation.) The *converse* operator for relations is defined by $R^{-1} = \{(b, a) \mid a\ R\ b\}$. Composition of relations is defined by $R\ ;\ S = \{(a, c) \mid \exists b.\ a\ R\ b\ \wedge\ b\ S\ c\}$.

The basic theory of binary relations was developed by Peirce, around 1870. Schröder extended the theory in a very thorough and systematic way around 1895. Tarksi's aim in writing his well–known article [62] in 1941 was to "awaken interest in a certain neglected logical theory", saying that "the calculus of relations deserves much more attention than it receives", having "an intrinsic charm and beauty which makes it a source of intellectual delight to all who become acquainted with it." Tarksi [62] proposes a few simple axioms as the definition for a relational algebra. Modern introductions to Tarski's calculus that may be of interest to calculational programmers are given by Dijkstra [22] and van Gasteren and Feijen [25]. Schmidt and Ströhlein have recently published a textbook on the calculus of relations [56]. An introduction to the algebraic properties of relations is given in section 3.3.

Several authors have proposed the use of a relational calculus as a programming language. Jones and Sheeran's relational language Ruby [58, 37, 41] is used to design circuits. Bird and de Moor [20, 9] are developing a (categorical) relational language, building on the work of Bird and Meertens on the functional language Squiggol [8, 49]. Backhouse et al [4] have similar aims with their Spec calculus. MacLennan's language RPL [47] is a relational generalisation of Backus' pioneering functional language FP [5]. Cattrall [17] extends the work of MacLennan on the implementation of RPL in the light of advances in functional programming languages, notably the use of lazy evaluation and polymorphic type inference. Haeberer and Veloso [68] derive a number of simple programs using a relational language.

Both Ruby and the Spec calculus have much in common with Bird and Meertens' functional language Squiggol. It is interesting then to briefly survey some developments of the basic Squiggol language. Programs in Squiggol work upon trees, lists, bags and sets, the so–called 'Boom hierarchy'. The framework was uniformly extended to cover recursive types by Malcolm [48], by means of the 'F–algebra' paradigm of type definition, and the resulting 'catamorphic' programming style. (A catamorphism is a unique homomorphism from an initial F–algebra.) Gibbons' recent thesis [29] is an extensive exploration of these ideas in deriving programs that operate upon different kinds of trees. Meijer's thesis [50] shows how Malcolm's work can be cast in a framework where types are cpo's rather than unstructured sets, with the advantage that finite and infinite objects can inhabit the same type; applications in deriving compilers from denotational semantics are given. Fokkinga's thesis [27] shows how the F–algebra paradigm may be extended to handle types whose constructors are required to satisfy laws.

Berghammer [7] proposes the use of relational algebra to specify types and programs; the idea of characterising types by a number of relational formulae is also explored in Desharnais' thesis [21]. Haeberer and Veloso [67] have returned to

Tarski's question of how the basic calculus of relations might be extended to gain the expressive power of first–order logic; section 2.0.1 gives more details. A categorical treatment of binary relations is given by Freyd and Scedrov in their recent book [28]; other relevant categorical work includes that of Barr [6], and Carboni, Kasangian and Street [16]. In his D.Phil. thesis [20] de Moor shows how categorical results about relations can be used in solving dynamic programming problems. In section 2.0.3 we summarise an interesting categorical result about relations, namely that functors on functions uniquely extend to functors on relations.

A simpler notion than a relational algebra that has received much attention recently is that of a *quantale* [55]. Brown and Gurr [12] show that relational quantales [13] are models for a special variant of linear logic; section 2.0.2 gives more details. Quantales have been proposed by Hoare and He [31] as models for the semantics of non–deterministic 'while' programs and for program specification.

The space of relations is isomorphic to the space of set–valued functions; de Moor [19] has experimented with manipulating set–valued functions. Every relation can be expressed as the composition of the converse of a functional relation and a functional relation; Riguet [54] explores the algebraic properties of the special class of relations that can be expressed the other way around, as the composition of a functional relation and the converse of a functional relation; such relations are called *difunctional*. All the programs that we derive in chapter 7 are difunctional.

In the remainder of this chapter we expand upon a few interesting references above that are perhaps not so well–known to calculational programmers.

2.0.1 Classical logic

Many properties of binary relations can be succinctly expressed using only the operators of Tarski's relational calculus. For example, that a relation R is a pre–ordering (reflexive and transitive) can be expressed as $(R \; ; \; R) \cup id \;\subseteq\; R$. One is led to ask whether all properties of relations that can be written using first–order classical logic can be expressed within relational calculus? The answer is no. The problem is with existential quantification. Logical terms involving $\exists$ are translated to relational terms involving "$;$", this being the only operator of the calculus defined using $\exists$. Observe that the $\exists$–bound variable b occurs twice within $R \; ; \; S \; = \; \{(a,c) \mid \exists b. \; a \; R \; b \; \wedge \; b \; S \; c\}$. Relational calculus provides no means to make extra copies of such a b if it is required at more than two occurrences, and hence such logical terms can't be translated into relational terms. The desire to extend relational calculus such that the answer to the translation question becomes yes lead to the development of 'cylindric algebras' and 'polyadic algebras'.

Veloso and Haeberer [67] present a simple new approach to gaining the expressive power of first–order logic, based upon the introduction of a product construction on relations (notably, a product operator × on relations, defined by $(a,b) \; R \times S \; (c,d)$ iff $a \; R \; c \wedge b \; S \; d$.) Since Jones and Sheeran's calculus Ruby provides such a product construction, this result tells us that the calculus that we use in this thesis to derive programs has at least the expressive power of first-order logic.

2.0.2 Linear logic

A *quantale* [55] is a 4–tuple $\langle \mathcal{Q}, \leq, \otimes, 1 \rangle$ such that $\langle Q, \leq \rangle$ is a complete lattice, $\langle Q, \otimes, 1 \rangle$ is a monoid, and $\otimes$ is universally disjunctive (distributes through all $\bigcup$'s) in both arguments. Quantales are relevant to us in this thesis because the space $P(\mathcal{A} \times \mathcal{A})$ of binary relations over a universe $\mathcal{A}$ forms a quantale, with $\leq$ as set inclusion, $\otimes$ as relational composition, and 1 as the identity relation on $\mathcal{A}$. In fact, such relational models are canonical, in the sense that every quantale is isomorphic to a quantale of relations on its underlying set [13].

That a function f on a complete lattice be universally disjunctive is precisely the condition for there being a unique function g satisfying $f \; x \leq y \; \equiv \; x \leq g \; y$. Such a g is known as the *right adjoint* to f. Adjoint functions are interesting to us because they have properties that can be useful in calculation; section 3.2.2 gives more details. That relational composition "$;$" is universally disjunctive in both arguments means that the functions $(R \; ; \; -)$ and $(- \; ; \; R)$ have right adjoints.

Backhouse [4] denotes the adjoint functions by $(R/-)$ and $(-\backslash R)$. The operators $/$ and $\backslash$ have proved to be very useful in manipulating relational formulae [4, 35].

In [12] Brown and Gurr extend their work on quantales, showing that relational quantales form a sound and complete class of models for 'non–commutative intuitionistic linear logic.' (Linear intuitionistic logic differs from intuitionistic logic primarily in the absence of the structural rules of weakening and contraction, with the effect that each premise in a sequent must be used precisely once in proving the single consequent. In non–commutative linear logic, the exchange rule is also discarded, with the effect that the position of a premise within a sequent becomes significant, and that two linear implication operators become necessary.) This result is interesting to us because the linear implication operators are modelled within a relational quantale as $/$ and $\backslash$, operators which are so useful in calculating with relations; moreover, the result suggests that studying relational algebra might be a useful stepping stone to understanding linear logic.

2.0.3 Extending functors

The *map* operator from functional programming [10] is a well–known example of a 'functor'. (Recall that a functor is a mapping from types to types, together with a mapping from functions to functions that must preserve identity functions and distribute through composition.) There is also a functor *map′*, defined below, that works with relations rather than functions:

$$\begin{array}{cccll} [\,] & \mathit{map}'\ R & [\,] & \equiv & \mathit{true}, \\ (x.xs) & \mathit{map}'\ R & (y.ys) & \equiv & x\ R\ y\ \wedge\ xs\ (\mathit{map}'\ R)\ ys. \end{array}$$

As one would expect, the relational *map′* is an extension of the functional *map*, in that *map′* behaves as *map* when applied to functional relations. Being precise,

$$\mathit{map}'\ (\mathit{lift}\, f)\ =\ \mathit{lift}\ (\mathit{map}\ f),$$

where *lift* converts a function to a relation in the obvious way, that is,

$$\mathit{lift}\, f = \{(a, f a) \mid a \in \mathit{dom}\ f\}.$$

An interesting result (due to Carboni, Kelly and Wood [15]) is that map' is in fact the *only* monotonic functor on relations that extends map in this way. The result is not specific to map: every functor on functions has at most one extension to a monotonic functor on relations; see [20] for a discussion of conditions under which the extension exists. In conjunction with another result about functors (concerning the existence of initial algebras), de Moor [20] cites the unique extension result as justification for the slogan "The generalisation of programming with total functions to programming with relations is free of surprises."

Chapter 3

Prelude

In this chapter we introduce some standard material that is used in the remainder of the thesis. Section 3.1 introduces our notation for proofs. Section 3.2 reviews some concepts from lattice theory and Galois theory. Section 3.3 gives the basic properties of the operators of relational calculus. And finally, section 3.4 introduces the relational language Ruby in which all our derivations are made.

3.1 Proofs

We adopt the style for calculations and proofs as developed by Dijkstra, Feijen, van Gasteren, and others [23, 66]. The reader should have little problem in following our calculations with no prior knowledge of this style. Let us prove, by way of an example, that functional relations are closed under composition: $\mathit{fn}.R \wedge \mathit{fn}.S \;\Rightarrow\; \mathit{fn}.(R\ ;\ S)$, where $\mathit{fn}.R \;\equiv\; x\ R\ y \wedge x\ R\ z \;\Rightarrow\; y = z$. Our proof in fact uses the equivalent point–free definition $\mathit{fn}.R \;\equiv\; R^{-1}\ ;\ R \subseteq \mathit{id}$.

$$
\begin{array}{ll}
 & \underline{\mathit{fn}.(R\ ;\ S)} \\
\equiv & \qquad \{\ \text{defn}\ \} \\
 & \underline{(R\ ;\ S)^{-1}}\ ;\ R\ ;\ S\ \subseteq\ \mathit{id} \\
\equiv & \qquad \{\ \text{law for converse}\ \} \\
 & S^{-1}\ ;\ \underline{R^{-1}\ ;\ R}\ ;\ S\ \subseteq\ \mathit{id}
\end{array}
$$

$$
\begin{array}{ll}
\Leftarrow & \quad \{ \text{ assumption: } \mathit{fn}.R \ \} \\
& S^{-1} \ ; \ \underline{id \ ; \ S} \ \subseteq \ id \\
\equiv & \quad \{ \text{ law for identity } \} \\
& S^{-1} \ ; \ S \ \subseteq \ id \\
\equiv & \quad \{ \text{ defn } \} \\
& \mathit{fn}.S
\end{array}
$$

Note that associativity of composition is used implicitly at a number of places to avoid parenthesising repeated compositions. Note also that monotonicity of "$;$" is used implicitly in the $\Leftarrow$ step. Such simple properties are used so frequently in calculations that they are rarely mentioned in the hints between steps.

One might have expected the proof above to proceed in the reverse direction. Experience has shown however [66] that in proving a proposition $A \Rightarrow B$, it is often much simpler to begin with the consequent B and work backwards to the assumption A; if there is more than one assumption (as is the case above) it is usually convenient not to work backwards to their conjunction, but to use one or more of the assumptions as hints during the proof.

As in the example proof above, we sometimes underline parts of a formulae. Such underlining has no formal meaning; rather it is used to draw the eye to those parts of the formulae that are being changed in moving to the next step.

3.2 Lattices

In this section we review some concepts from lattice theory [30]. A more detailed introduction that may be of interest to calculational programmers can be found as a prelude to van der Woude's notes on predicate transformers [64].

Recall that a *partially–ordered set* (poset) is a pair $\langle \mathcal{A}, \sqsubseteq \rangle$, where $\sqsubseteq$ is a reflexive, anti–symmetric and transitive relation on the set $\mathcal{A}$. For the remainder of this section $\langle \mathcal{A}, \sqsubseteq \rangle$ is assumed to be a poset. Sometimes in this thesis we find it preferable to use the dual ordering $\sqsupseteq$, defined by $a \sqsupseteq b \;\; \hat{=} \;\; b \sqsubseteq a$.

Definition 1: An *upper bound* of a set $\mathcal{X} \subseteq \mathcal{A}$ is an $a \in \mathcal{A}$ for which

$$\forall x \in \mathcal{X}.\; x \sqsubseteq a.$$

Element $a \in \mathcal{A}$ is the *least upper bound* (lub) of $\mathcal{X} \subseteq \mathcal{A}$ if

$$(\forall x \in \mathcal{X}.\; x \sqsubseteq y) \;\equiv\; a \sqsubseteq y.$$

If it exists, we denote the unique lub by $\sqcup\mathcal{X}$. Turning things around, an element $a \in \mathcal{A}$ is the *greatest lower bound* (glb) of $\mathcal{X} \subseteq \mathcal{A}$ if

$$(\forall x \in \mathcal{X}.\; y \sqsubseteq x) \;\equiv\; y \sqsubseteq a.$$

If it exists, we denote the unique glb by $\sqcap\mathcal{X}$.

Definition 2: A *lattice* is a poset for which lubs and glbs exist for all finite subsets of elements. Equivalently, a lattice is a poset for which there exist least and greatest elements $\bot\!\!\bot$ and $\top\!\!\top$, and binary lub and glb operators $\sqcup$ and $\sqcap$. The properties required of these elements and operators are captured in the following four axioms:

$$\bot\!\!\bot \sqsubseteq a,$$

$$a \sqsubseteq \top\!\!\top,$$

$$a \sqsubseteq c \;\wedge\; b \sqsubseteq c \;\;\equiv\;\; a \sqcup b \;\sqsubseteq\; c,$$

$$c \sqsubseteq a \;\wedge\; c \sqsubseteq b \;\;\equiv\;\; c \;\sqsubseteq\; a \sqcap b.$$

One can show from these axioms that the operators $\sqcup$ and $\sqcap$ are idempotent, commutative and associative. Moreover, $\bot\!\!\bot$ and $\top\!\!\top$ are respectivity the units for $\sqcup$ and $\sqcap$, and the zeros for $\sqcap$ and $\sqcup$.

We consider now some extra properties that a lattice might have:

Definition 3: A lattice is *complete* if lubs and glbs exist of all subsets of elements, not just finite subsets. A lattice is *distributive* if for all finite sets $\mathcal{X} \subseteq \mathcal{A}$ the following two equations hold:

$$\begin{aligned} a \ \sqcup \ (\sqcap \mathcal{X}) \ &= \ \sqcap \, \{a \sqcup x \mid x \in \mathcal{X}\}, \\ a \ \sqcap \ (\sqcup \mathcal{X}) \ &= \ \sqcup \, \{a \sqcap x \mid x \in \mathcal{X}\}. \end{aligned}$$

Equivalently, a lattice is distributive if

$$\begin{aligned} a \ \sqcup \ (b \ \sqcap \ c) \ &= \ (a \ \sqcup \ b) \ \sqcap \ (a \ \sqcup \ c), \\ a \ \sqcap \ (b \ \sqcup \ c) \ &= \ (a \ \sqcap \ b) \ \sqcup \ (a \ \sqcap \ c). \end{aligned}$$

A complete lattice is *completely distributive* if the distribution laws hold for all sets $\mathcal{X} \subseteq \mathcal{A}$, not just finite subsets. Elements a and b of a lattice are *complements* if $a \sqcup b = \top\!\!\top$ and $a \sqcap b = \bot\!\!\bot$. A lattice is *complemented* if every element a has a unique complement $\neg a$.

All the notions introduced above are combined in our final definition:

Definition 4: A *powerset lattice* (plat) is a complete boolean algebra, i.e. a complete, completely distributive, complemented lattice. The powerset $P(\mathcal{A}) = \{\mathcal{X} \mid \mathcal{X} \subseteq \mathcal{A}\}$ of any set $\mathcal{A}$ forms a powerset lattice under set inclusion $\subseteq$, hence the terminology.

3.2.1 Monotonicity and junctivity

Let $\mathcal{A}$ and $\mathcal{B}$ be complete lattices, $f : A \rightarrow B$, $\mathcal{X} \subseteq \mathcal{A}$, and $a, b \in \mathcal{A}$. In the following definitions, we abbreviate $\{fx \mid x \in \mathcal{X}\}$ by $f^*\mathcal{X}$.

Definition 5:

$$
\begin{array}{l}
f \text{ is } \textit{monotonic} \;\; \widehat{=} \;\; a \sqsubseteq b \;\Rightarrow\; fa \sqsubseteq fb, \\
f \text{ is } \textit{disjunctive} \;\; \widehat{=} \;\; \textit{finite}.\mathcal{X} \;\Rightarrow\; f(\sqcup\mathcal{X}) \;=\; \sqcup\,(f^*\mathcal{X}), \\
f \text{ is } \textit{conjunctive} \;\; \widehat{=} \;\; \textit{finite}.\mathcal{X} \;\Rightarrow\; f(\sqcap\mathcal{X}) \;=\; \sqcap\,(f^*\mathcal{X}), \\
f \text{ is } \textit{universally disjunctive} \;\; \widehat{=} \;\; f(\sqcup\mathcal{X}) \;=\; \sqcup\,(f^*\mathcal{X}), \\
f \text{ is } \textit{universally conjunctive} \;\; \widehat{=} \;\; f(\sqcap\mathcal{X}) \;=\; \sqcap\,(f^*\mathcal{X}).
\end{array}
$$

There are a number of other useful formulations of monotonicity:

Lemma 6:

$$
\begin{array}{lrcl}
\text{(a)} & \textit{monotonic}.f & \equiv & fa \;\sqsubseteq\; f(a \sqcup b). \\
\text{(b)} & \textit{monotonic}.f & \equiv & f(a \sqcap b) \;\sqsubseteq\; fa. \\
\text{(c)} & \textit{monotonic}.f & \equiv & \sqcup\,(f^*\mathcal{X}) \;\sqsubseteq\; f(\sqcup\mathcal{X}). \\
\text{(d)} & \textit{monotonic}.f & \equiv & f(\sqcap\mathcal{X}) \;\sqsubseteq\; \sqcap\,(f^*\mathcal{X}).
\end{array}
$$

3.2.2 Galois connections

In this section we review some concepts from Galois theory [30], and make some comments about how Galois connections are used in relational programming. Galois connections have many other interesting applications in Computing Science; see for example [51]. More recently, Galois connections play a key role in Hughes and Launchbury's work on reversing program analyses [32].

Definition 7: A *Galois connection* between complete lattices $\mathcal{A}$ and $\mathcal{B}$ is a pair of functions $f : \mathcal{B} \to \mathcal{A}$ and $g : \mathcal{A} \to \mathcal{B}$, satisfying

$$f\,b \;\sqsubseteq\; a \;\;\equiv\;\; b \;\sqsubseteq\; g\,a$$

for all elements $a \in \mathcal{A}$ and $b \in \mathcal{B}$.

We say that function f is *left adjoint* to function g, and conversely, that g is *right adjoint* to f. That $\mathcal{A}$ and $\mathcal{B}$ form lattices is not strictly necessary to define the notion of a Galois connection; that $\mathcal{A}$ and $\mathcal{B}$ be pre–ordered sets is sufficient. All our applications however involve complete lattices, an assumption that allows our introduction to Galois connections to be simplified. (To avoid confusion between left and right adjoints, just remember that the left adjoint $f : \mathcal{B} \to \mathcal{A}$ appears to the left of an inclusion $\sqsubseteq$ in definition 7; it might also be helpful to use a left–facing arrow $\leftarrow$ and think of the left adjoint f as having type $\mathcal{A} \leftarrow \mathcal{B}$.)

A Galois connection is a simple instance of the general categorical notion of an *adjunction*, hence the reference to functions f and g as adjoints. Studying the simple notion of a Galois connection is an excellent precursor to studying adjunctions. Briefly, an adjunction between categories $\mathcal{C}$ and $\mathcal{D}$ is a pair of functors $F : \mathcal{D} \to \mathcal{C}$ and $G : \mathcal{C} \to \mathcal{D}$, such that for all objects $B \in \mathcal{D}$ and $A \in \mathcal{C}$, the arrows of type $FB \to A$ in $\mathcal{C}$ are in one–to–one correspondence with arrows of type $B \to GA$ in $\mathcal{D}$, in a way that is independent of the choice of particular objects A and B. More formally, we require a 'natural isomorphism of hom-functors.' Fokkinga [26] introduces and proves many properties of adjunctions in a calculational style.

What is expressed in stating that f and g are adjoint functions?

Lemma 8:

$$\begin{array}{ll} & f\,b \sqsubseteq a \;\equiv\; b \sqsubseteq g\,a \\ \equiv & \\ & f \text{ and } g \text{ are monotonic } \;\wedge\; f \circ g \sqsubseteq id \;\wedge\; id \sqsubseteq g \circ f. \end{array}$$

We see now that a Galois connection is precisely a pair of monotonic functions that are *approximate inverses*, in the sense that the application $f\,(g\,a)$ always gives a result that approximates a, while the application $g\,(f\,b)$ always gives a result approximated by b. (Recall that functions f and g are true inverses precisely when $f\,(g\,a) = a$ and $g\,(f\,b) = b$ for all elements $a \in \mathcal{A}$ and $b \in \mathcal{B}$.)

The adjoint functions satisfy powerful junctivity properties:

Lemma 9:

$$\begin{array}{ll} & f \text{ and } g \text{ are adjoint functions} \\ \Rightarrow & \\ & f \text{ is universally disjunctive} \quad \wedge \quad g \text{ is universally conjunctive.} \end{array}$$

In a complemented lattice the adjoints approximately distribute over $\neg$:

Lemma 10:

$$\begin{array}{ll} & f \text{ and } g \text{ are adjoint functions} \\ \Rightarrow & \\ & f\,(\neg b) \;\sqsubseteq\; \neg(f\,b) \quad \wedge \quad \neg(g\,a) \;\sqsubseteq\; g\,(\neg a). \end{array}$$

Adjoint functions uniquely determine one another:

Lemma 11:

$$\begin{array}{ll} & f \text{ and } g \text{ are adjoint functions} \\ \Rightarrow & \\ & f\,b = \sqcap\,\{a \mid b \sqsubseteq g\,a\} \quad \wedge \quad g\,a = \sqcup\,\{b \mid f\,b \sqsubseteq a\}. \end{array}$$

Lemma 9 tells us that adjoints are universally junctive; that a function be universally junctive is in fact the precise condition for existence of an adjoint:

Lemma 12:

$$f \text{ has a right adjoint} \quad \equiv \quad f \text{ is universally disjunctive;}$$
$$g \text{ has a left adjoint} \quad \equiv \quad g \text{ is universally conjunctive.}$$

This thesis is about the use of relational languages in deriving programs. We identify three applications of Galois connections in this context:

> *Remembering properties.* The algebra of relations comprises a great many laws. Many such laws however are just a consequence of operators in the calculus being adjoint to one another. Uncovering and

remembering such adjoint operators proves to be an excellent way to remember many of the properties of such operators. For example, the law $R^{-1} \subseteq S \;\equiv\; R \subseteq S^{-1}$ from relational calculus is precisely the statement that the converse operator is self adjoint. Immediate now from our knowledge of adjoint functions are the following properties:

- $(R^{-1})^{-1} = R$,
- $\bot\!\!\bot^{-1} = \bot\!\!\bot$,
- $\top\!\!\top^{-1} = \top\!\!\top$,
- $(R \cup S)^{-1} \;=\; R^{-1} \cup S^{-1}$,
- $(R \cap S)^{-1} \;=\; R^{-1} \cap S^{-1}$,
- $\neg(R^{-1}) = (\neg R)^{-1}$.

In axiomatic approaches to relational calculus, where manipulation of relational formulae is conducted without ever appealing to the set–theoretic definitions of the operators, Galois connections are clear cases for being chosen as axioms in themselves.

Simplifying calculation. Equivalences are known to be a powerful tool in helping to make calculations clear and concise. Galois equivalences

$$f\,b \sqsubseteq a \;\;\equiv\;\; b \sqsubseteq g\,a$$

are particularly useful, allowing us to switch between isolation of a variable on the left or right side of an inclusion. For example, here is a proof of one part of lemma 9, namely that a function f satisfying such a Galois equivalence must be universally disjunctive:

$$f \sqcup \mathcal{X} \;=\; \sqcup\, f^{*}\mathcal{X}$$

$\equiv$ { adjoints are monotonic (6c) }

$$f \sqcup \mathcal{X} \;\sqsubseteq\; \sqcup\, f^{*}\mathcal{X}$$

$\equiv$ { Galois: f, g }

$$
\begin{array}{ll}
 & \sqcup\mathcal{X} \;\sqsubseteq\; g\,(\sqcup f^{*}\mathcal{X}) \\
\equiv & \qquad \{\ \text{Galois: } \sqcup, \triangle\ \} \\
 & \forall x \in \mathcal{X}.\ (x \;\sqsubseteq\; g\,(\sqcup f^{*}\mathcal{X})) \\
\equiv & \qquad \{\ \text{Galois: } f, g\ \} \\
 & \forall x \in \mathcal{X}.\ (fx \;\sqsubseteq\; \sqcup f^{*}\mathcal{X}) \\
\equiv & \qquad \{\ \text{Galois: } \sqcup, \triangle\ \} \\
 & \sqcup f^{*}\mathcal{X} \;\sqsubseteq\; \sqcup f^{*}\mathcal{X} \\
\equiv & \qquad \{\ \text{reflexivity}\ \} \\
 & \mathit{true}
\end{array}
$$

Backhouse's short note [2] gives another good example. (The hint Galois: $\sqcup, \triangle$ above refers to the fact that $\sqcup$ arises as a left adjoint. For example, the equivalence $a \sqcup b \;\sqsubseteq\; c \;\equiv\; a \sqsubseteq c \;\wedge\; b \sqsubseteq c$ expresses that the binary lub operator $\sqcup$ is left adjoint to the doubling operator $\triangle$ defined by $\triangle x = (x, x)$. Dually, $\sqcap$ is right adjoint to $\triangle$.)

Suggesting operators. A function has an adjoint precisely when it is universally junctive. It may be the case that the universal junctivity of some function is well–known, but its adjoint function is not. An important example is relational composition, an operation that is universally disjunctive in both arguments. The two corresponding right adjoint functions are termed 'factoring' operators by Backhouse [4]. These operators have many properties that can be useful in calculation; section 3.3.1 gives more details. Moreover, factors have computational meaning, corresponding to the notion of weakest pre– and post–specifications [31]. Related to factors are 'monotype factors' [3], used in simplifying proofs involving demonic composition of relations; again these unfamiliar operators arose as adjoints to universally junctive functions.

3.3 Relational algebra

In this section we review the basic properties of the operators of relational calculus, by defining the notion of a relational algebra. The definition we present is due to Backhouse [4], but is equivalent to Tarski's original formulation [62].

Definition 13: A *relational algebra* is a 5–tuple

$$\langle \mathcal{A}, \subseteq, \,;\,, id, {}^{-1} \rangle$$

for which all the following hold:

(a) $\langle \mathcal{A}, \subseteq \rangle$ is a (non empty) power–set lattice.

(b) $\langle \mathcal{A}, \,;\,, id \rangle$ is a monoid:

$$R \;;\; (S \;;\; T) \;=\; (R \;;\; S) \;;\; T,$$

$$id \;;\; R \;=\; R \;=\; R \;;\; id.$$

(c) $(R\,;)$ and $(\,;R)$ are universally disjunctive: $\forall \mathcal{X} \subseteq \mathcal{A}$,

$$R \;;\; \bigcup \mathcal{X} \;=\; \bigcup \{R \;;\; S \mid S \in \mathcal{X}\},$$

$$\bigcup \mathcal{X} \;;\; R \;=\; \bigcup \{S \;;\; R \mid S \in \mathcal{X}\}.$$

(d) ${}^{-1}$ is self adjoint:

$$R^{-1} \subseteq S \;\equiv\; R \subseteq S^{-1}.$$

(e) ${}^{-1}$ is a contravariant homomorphism on the monoid $\langle \mathcal{A}, \,;\,, id \rangle$:

$$(R \;;\; S)^{-1} \;=\; S^{-1} \;;\; R^{-1},$$

$$id^{-1} \;=\; id.$$

(f) The *middle–exchange* rule:

$$X \;;\; R \;;\; Y \;\subseteq\; S \;\equiv\; X^{-1} \;;\; \neg S \;;\; Y^{-1} \;\subseteq\; \neg R.$$

(g) *Tarksi's* rule:

$$\top\!\!\top \;;\; R \;;\; \top\!\!\top = \top\!\!\top \;\equiv\; R \neq \perp\!\!\perp.$$

Of course, the space $P(\mathcal{U}\times\mathcal{U})$ of relations over a non–empty universe $\mathcal{U}$ (together with the standard set–theoretic definitions for the four operators) is an example of a relational algebra. The axiom system is not however complete for these models. Complete axiomatisation of relational calculus with a finite number of axioms is not possible; see section 2.158 of [28]. Note also that axioms 13a–13c are the properties required of a quantale [55]; see section 2.0.2. There is also a close correspondence between a relational algebra and the categorical notion of an *allegory* [28].

A relational algebra comprises three parts: the plat structure, the monoid structure, and the converse structure. Each part is linked to each of the others by a single axiom: 13c links the plat and monoid structures, 13d links the plat and converse structures, and 13e links the monoid and converse structures. It is interesting to note that, as shown below, $id^{-1} = id$ is in fact derivable from other axioms:

$$
\begin{array}{ll}
 & \underline{id^{-1}} \\
= & \qquad \{ \text{ identity } \} \\
 & \underline{id^{-1} \ ; \ id} \\
= & \qquad \{ \text{ converse } \} \\
 & \underline{(id^{-1} \ ; \ id)^{-1-1}} \\
= & \qquad \{ \text{ distribution } \} \\
 & (id^{-1} \ ; \ \underline{id^{-1-1}})^{-1} \\
= & \qquad \{ \text{ converse } \} \\
 & \underline{(id^{-1} \ ; \ id)^{-1}} \\
= & \qquad \{ \text{ identity } \} \\
 & \underline{id^{-1-1}} \\
= & \qquad \{ \text{ converse } \} \\
 & id
\end{array}
$$

Axiom 13f (van der Woude's middle–exchange rule [4]) links all three layers at once; in the presence of the other axioms it is equivalent to the standard Schröder equivalences, but has proved better suited to the demands of calculation. Axioms 13e and 13c are in fact derivable from the the middle–exchange rule [25]. Axiom 13g (Tarski's rule) precludes one–point models of a relational algebra (that is, ensures $\top\top \neq \bot\bot$), and the trivial assignment $\langle {}^{-1}, ;, id\rangle := \langle \lambda R.R, \cap, \top\top\rangle$. In this thesis we never have need to apply the middle–exchange rule or Tarski's rule.

A large number of derived properties of relational algebras are useful when manipulating relational formulae. For example, axiom 13c gives the following:

Lemma 14:

- $\bot\bot$ is a zero for composition: $\bot\bot \,;\, R = R \,;\, \bot\bot = \bot\bot$,
- composition is monotonic in both arguments,
- $\top\top \,;\, \top\top = \top\top$.

3.3.1 Factors

The composition operator for relations is universally disjunctive in both arguments. Universal disjunctivity is precisely the condition for a function having a right adjoint; see lemma 12. Backhouse [4] writes $(R/–)$ for the right adjoint to $(R \,;\, –)$, and $(–\backslash R)$ for the right adjoint to $(– \,;\, R)$, and calls terms of the form R/S *left factors* and terms of the form $S\backslash R$ *right factors*. Here are the defining Galois connections:

Definition 15:

$$\begin{array}{rcl} R \,;\, S \subseteq T & \equiv & S \subseteq R/T, \\ S \,;\, R \subseteq T & \equiv & S \subseteq T\backslash R. \end{array}$$

That adjoint functions are approximate inverses (lemma 8) is expressed in the case of factors by the following *cancellation* properties; it is these rules that motivate the use of the division–like symbols / and \:

Lemma 16:

$$R \,;\, (R/S) \;\subseteq\; S \;\subseteq\; R/(R \,;\, S),$$
$$(S\backslash R) \,;\, R \;\subseteq\; S \;\subseteq\; (S \,;\, R)\backslash R.$$

Factors have proved very useful in simplifying the manipulation of relational formulae [4]. (Backhouse has observed that many of the proofs in Schmidt and Ströhlein's text on relational algebra [56] can be considerably simplified by using factors.) As an example of how factors are used, let F be a monotonic function from relations to relations that distributes over composition. Monotonicity guarantees existence of a least fixpoint μF for F. Let us now prove that μF is transitive:

$$
\begin{array}{ll}
 & \mathit{transitive}.\mu F \\
\equiv & \quad \{ \text{ lemma 50 } \} \\
 & \mu F \,;\, \mu F \;\subseteq\; \mu F \\
\equiv & \quad \{ \text{ lemma 15 } \} \\
 & \mu F \;\subseteq\; \mu F/\mu F \\
\Leftarrow & \quad \{ \textit{ fixpoint induction } \} \\
 & F(\mu F/\mu F) \;\subseteq\; \mu F/\mu F \\
\Leftarrow & \quad \{ \text{ factors } \} \\
 & F\mu F/F\mu F \;\subseteq\; \mu F/\mu F \\
\equiv & \quad \{ \; F\mu F = \mu F \; \} \\
 & \mathit{true}
\end{array}
$$

(Tarski [63] gives more details about fixpoints of functions between lattices, and presents the 'fixpoint induction' rule used above: $\mu F \subseteq R \;\Leftarrow\; FR \subseteq R$.) The hint 'factors' above refers here to the law $F(R/S) \;\subseteq\; FR/FS$, proved as follows:

$$F(R/S) \;\subseteq\; FR/FS$$

$\equiv$ { lemma 15 }

$FR \,;\, F(R/S) \;\subseteq\; FS$

$\equiv$ { distribution }

$F(R \,;\, R/S) \;\subseteq\; FS$

$\Leftarrow$ { monotonicity }

$R \,;\, R/S \;\subseteq\; S$

$\equiv$ { lemma 16 }

true

Backhouse [3] uses a special variant of factors, so–called ‘monotype factors’, to simplify proofs involving demonic composition of relations. In section 6.1.3 we use factors to define and prove properties of *domain operators* $<$ and $>$.

3.4 Ruby

In this section we introduce Ruby, the relational calculus developed by Jones and Sheeran for describing and designing circuits [58, 37, 41].

Proofs of the algebraic laws that we give in this section can be found in some of the earlier Ruby articles. The difference between Ruby programs that are specifications and those that are implementations is the subject of chapter 4. In chapter 7 we derive a number of circuits using Ruby.

A relation on a universe $\mathcal{U}$ is a set of pairs of elements of $\mathcal{U}$. In Ruby the universe is a fixed set $\mathcal{U}$ containing at least the booleans and integers, and closed under finite tupling. If R is a relation, we write $R : \mathcal{A} \leftrightarrow \mathcal{B}$ to mean that R is a relation between subsets $\mathcal{A}$ and $\mathcal{B}$ of the universe, i.e. that $R \subseteq \mathcal{A} \times \mathcal{B}$.

Given a function $f : \mathcal{A} \rightarrow \mathcal{B}$, the relation $f^r : \mathcal{A} \leftrightarrow \mathcal{B}$ (called the *graph* of f) is defined by $f^r = \{(a, fa) \mid a \in \mathcal{A}\}$. A useful trick is to make a unary function from a binary function by fixing one of its arguments; for example, $(*2)$ is the doubling function, and $(1/)$ is the reciprocating function. Formally, if $\oplus : \mathcal{A} \times \mathcal{B} \rightarrow \mathcal{C}$ then $(a\oplus) : \mathcal{B} \rightarrow \mathcal{C}$ and $(\oplus b) : \mathcal{A} \rightarrow \mathcal{C}$ (for $a \in \mathcal{A}$ and $b \in \mathcal{B}$) are defined by $(a\oplus)\ b = a \oplus b$ and $(\oplus b)\ a = a \oplus b$. Functional relations like $(*2)^r$ and $(+1)^r$ are much used in Ruby. In fact, the $(-)^r$ operator is always left implicit in Ruby programs. That is, no distinction is made between a function and a functional relation.

The basis for Ruby is Tarski's calculus of relations, as described in the first paragraph of chapter 2 and axiomatised in section 3.3, to which a number of extensions are made. The most important extension is the *par* (parallel composition) operator, which forms the product of two relations:

Definition 17: $(a, b)\ [R, S]\ (c, d) \ \ \hat{=}\ \ a\ R\ c \wedge b\ S\ d.$

The $[R, S]$ notation is used rather than $R \times S$ for historical reasons. (Note that $[R, S]$ is different from the standard cartesian product $R \times S$, defined by $(a, b)\ R \times S\ (c, d) \equiv a\ R\ b \wedge c\ S\ d$.) Par generalises in the obvious way to n-arguments $[R_1, R_2, \ldots, R_n]$; for example, $[R, S, T]$ relates triples to triples.

The converse operator distributes through par, and par is functorial:

Lemma 18:

$$[R, S]^{-1} \ = \ [R^{-1}, S^{-1}],$$
$$[R, S]\ ;\ [T, U] \ = \ [R\ ;\ T, S\ ;\ U].$$

Two common uses of par merit special abbreviations:

Definition 19:

$$\text{fst}\ R \ \ \hat{=}\ \ [R, id],$$
$$\text{snd}\ R \ \ \hat{=}\ \ [id, R].$$

The identity relation *id* is the simplest example of a *restructuring* or *wiring* relation in Ruby; the other common wiring relations are defined as follows:

Definition 20:

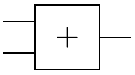
$$\begin{aligned}
&(a,b)\ \mathit{swap}\ (c,d) \ \ \widehat{=}\ \ a=d\ \wedge\ b=c,\\
&(a,b)\ \pi_1\ c \ \ \widehat{=}\ \ a=c,\\
&(a,b)\ \pi_2\ c \ \ \widehat{=}\ \ b=c,\\
&a\ \mathit{fork}\ (b,c) \ \ \widehat{=}\ \ a=b\ \wedge\ a=c,\\
&((a,b),c)\ \mathit{lsh}\ (d,(e,f)) \ \ \widehat{=}\ \ a=d\ \wedge\ b=e\ \wedge\ c=f,\\
&(a,(b,c))\ \mathit{rsh}\ ((d,e),f) \ \ \widehat{=}\ \ a=d\ \wedge\ b=e\ \wedge\ c=f.
\end{aligned}$$

Note that $\mathit{rsh} = \mathit{lsh}^{-1}$. The most commonly used properties of the wiring relations are given below, and are called *shunting* laws. We use the term shunting for any transformation of the form $R\ ;\ S = S\ ;\ T$. (Readers familiar with category theory will recognise that the shunting rules express naturality of the wiring primitives.)

Lemma 21:

$$\begin{aligned}
&[R,S]\ ;\ \mathit{swap}\ =\ \mathit{swap}\ ;\ [S,R],\\
&\mathrm{fst}\ R\ ;\ \pi_1\ =\ \pi_1\ ;\ R,\\
&\mathrm{snd}\ R\ ;\ \pi_2\ =\ \pi_2\ ;\ R,\\
&f \text{ is functional }\ \equiv\ f\ ;\ \mathit{fork}\ =\ \mathit{fork}\ ;\ [f,f],\\
&[[R,S],T]\ ;\ \mathit{lsh}\ =\ \mathit{lsh}\ ;\ [R,[S,T]],\\
&[R,[S,T]]\ ;\ \mathit{rsh}\ =\ \mathit{rsh}\ ;\ [[R,S],T].
\end{aligned}$$

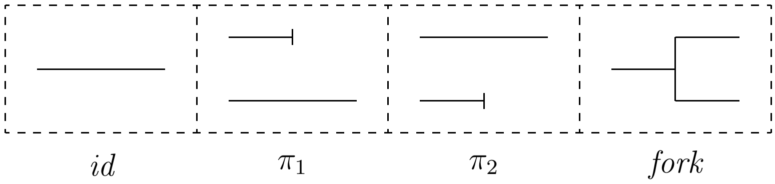
As well as denoting binary relations, Ruby terms have a pictorial interpretation using boxes and wires. Primitive terms are pictured as boxes labelled with the name of the primitive, and the appropriate number of wires on their left and right sides. For example, the addition relation + is pictured as follows:



.

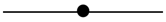
In such pictures, the convention is that domain values flow on the left-hand wires of a primitive, and range values on the right-hand wires. Corresponding to reading tuples of values from left to right, we read busses of wires from bottom to top.

The wiring primitives of Ruby are pictured just as wires. For example,



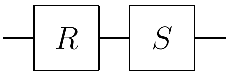
.

Partial identity relations $A \subseteq id$ correspond to wires that are constrained in the values that they may carry, constraints being pictured as blobs:



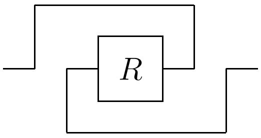
.

Terms built using operators of Ruby are pictured in terms of pictures of their arguments. A term R ; S is pictured by placing a picture of R to the left of a picture of S, and joining the intermediate wires together:



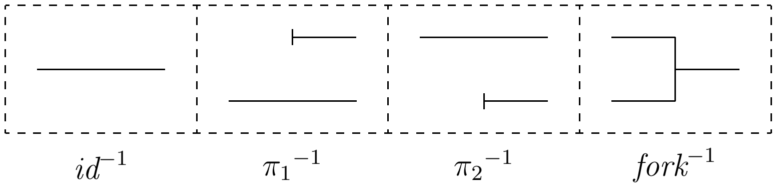
.

A term R^{-1} is pictured by bending wires in a picture of R:



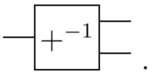
.

When R is a wiring relation, this bending of wires is equivalent to flipping a picture of the primitive about its vertical axis. For example,

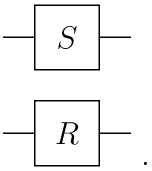


.

Laws such as $(R\ ;\ S)^{-1} = S^{-1}\ ;\ R^{-1}$ and $[R,S]^{-1} = [R^{-1},S^{-1}]$ allow any Ruby term to be rewritten such that the converse operator is only applied to primitive relations. In practice when picturing terms we assume that such rewriting of converses has been done. Moreover, we treat the converse of a primitive relation as itself a primitive relation; for example, $+^{-1}$ would be pictured as follows:

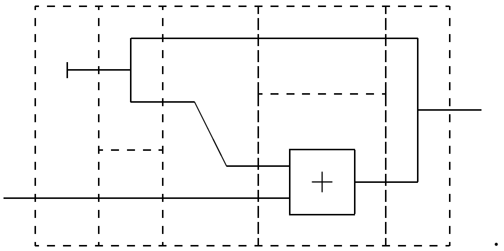


A term $[R,S]$ is pictured by placing a picture of R below a picture of S:

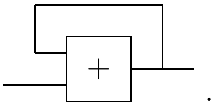


If desired, the difference between $[[R,S],T]$ and $[R,[S,T]]$ and $[R,S,T]$ can be made explicit in pictures by spacing some components further apart; alternatively one might space all three components equally, but 'crimp' wires closer together, or put vertical strokes through wires that are part of the same tuple.

By way of example, consider the Ruby program $R = {\pi_1}^{-1}\ ;\ \mathrm{snd}\, \mathit{fork}\ ;\ \mathit{rsh}\ ;$ $\mathrm{fst}\, +\ ;\ \mathit{fork}^{-1}$. This program is pictured as follows (the dashed boxes are included to help in understanding the translation from program to picture):

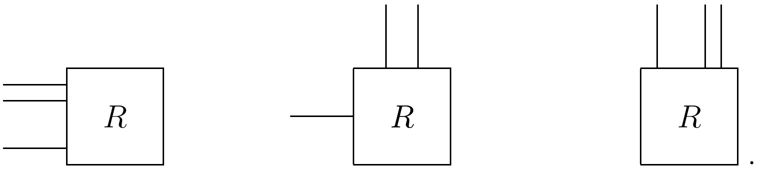


A simpler picture expresses just the connectivity:

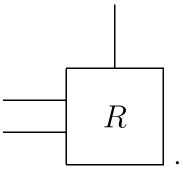


It is clear from this picture that $a\ R\ b \ \equiv\ b = a + b$, which assuming that a and b range over the integers, tells us that $a\ R\ b \ \equiv\ a = 0$.

Suppose that R is a relation defined by $(a, (b, c))\ R\ d \ \equiv\ P(a, b, c, d)$, where P is some predicate. With what we have said about pictures up to now, the relation R would be drawn as a box with three wires coming from the left side, and one wire coming from the right side. In Ruby one is in fact allowed to draw wires on all four sides of a box, with the convention that the left and top sides correspond to the domain of the relation, and the bottom and right sides correspond to the range. For example, here are three ways to draw the domain wires for the relation R:

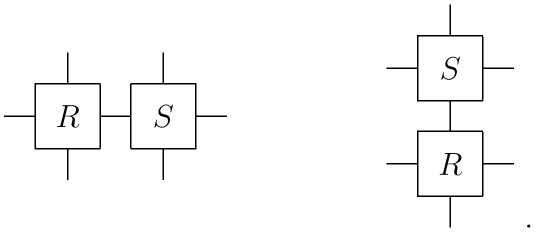


Here is a way that is not acceptable:



This last picture implies that R was defined in the form $((a, b), c)\ R\ d \ \equiv\ P(a, b, c, d)$, which is not the case; the place at which the domain wires are split between the left and top sides is significant in a picture, giving some type information about the corresponding relation. Of course, a similar restriction applies to splitting the range wires between the bottom and right sides of a picture.

Given two relations R and S on pairs of values, they can be composed $R\ ;\ S$ as any other relations; viewing them as 4-sided components, it is natural to think also of placing one beside the other, or one below the other:



The *beside* operator is defined as follows:

Definition 22:

$$(a,(b,c)) \;\; R \leftrightarrow S \;\; ((d,e),f) \quad \hat{=} \quad \exists x.\; (a,b)\; R\; (d,x) \wedge (x,c)\; S\; (e,f).$$

The *below* operator can be defined as the dual to beside:

Definition 23: $R \updownarrow S \quad \hat{=} \quad \left(R^{-1} \leftrightarrow S^{-1}\right)^{-1}.$

(It is interesting to note that $rsh \;=\; [id, id] \leftrightarrow [id, id]$ and $lsh \;=\; [id, id] \updownarrow [id, id]$.) In fact, even the beside operator need not be taken as primitive:

Lemma 24: $R \leftrightarrow S \;=\; rsh \;;\; \mathrm{fst}\; R \;;\; lsh \;;\; \mathrm{snd}\; R \;;\; rsh.$

(A similar result for below can be obtained using definition 23 and properties of converse.) The operators below and beside satisfy an 'abides' law:

Lemma 25: $(R \leftrightarrow S) \updownarrow (T \leftrightarrow U) \quad = \quad (R \updownarrow T) \leftrightarrow (S \updownarrow U).$

Ruby has a number of so–called *generic* combining forms that are defined recursively on an argument that ranges over the natural numbers. The simplest generic construction is the n–fold composition R^n of a relation R:

Definition 26:

$$\begin{array}{lcl} R^0 & \hat{=} & id, \\ R^{n+1} & \hat{=} & R^n \;;\; R. \end{array}$$

For example, $R^4 = R \mathbin{;} R \mathbin{;} R \mathbin{;} R$. One can prove by induction that $R^n \mathbin{;} R^m = R^{n+m}$ and $(R^n)^{-1} = (R^{-1})^n$. No confusion results from abbreviating $(R^{-1})^n$ by R^{-n}.

To make similar recursive definitions for the other generic constructions, we first define some generic wiring primitives. In the following definitions, $\sharp$ gives the arity of a tuple, and $+\!\!+$ is the concatenation operator for tuples. Tuples are indexed by subscripting, with index 0 giving the left–most component.

Definition 27:

$$\begin{aligned}
(x, ys)\ \mathit{apl}_n\ zs &\ \widehat{=}\ \sharp ys = n \ \wedge\ zs = (x) +\!\!+ ys, \\
(xs, y)\ \mathit{apr}_n\ zs &\ \widehat{=}\ \sharp xs = n \ \wedge\ zs = xs +\!\!+ (y), \\
(xs, ys)\ \mathit{zip}_n\ zs &\ \widehat{=}\ \sharp xs = \sharp ys = \sharp zs = n \quad \wedge \quad \forall i < n.\ zs_i = (xs_i, ys_i).
\end{aligned}$$

Here are some examples:

$$\begin{aligned}
&(1,(2,3,4))\ \mathit{apl}_3\ (1,2,3,4), \\
&((1,2,3),4)\ \mathit{apr}_3\ (1,2,3,4), \\
&((1,2,3),(4,5,6))\ \mathit{zip}_3\ ((1,4),(2,5),(3,6)).
\end{aligned}$$

The n–fold product of R is defined as follows:

Definition 28:

$$\begin{aligned}
\mathrm{map}_0\ R &\quad \widehat{=} \quad [], \\
\mathrm{map}_{n+1}\ R &\quad \widehat{=} \quad \mathit{apl}_n^{-1}\ ;\ [R, \mathrm{map}_n\ R]\ ;\ \mathit{apl}_n.
\end{aligned}$$

(Note that the [] in the $\mathrm{map}_0\ R$ definition is the 0–width par.) For example, $\mathrm{map}_4\ R = [R, R, R, R]$. The map_n operator distributes through composition (map_n is functorial), and commutes with the converse operator:

Lemma 29:

$$\begin{aligned}
\mathrm{map}_n\ R\ ;\ \mathrm{map}_n\ S &= \mathrm{map}_n\ (R\ ;\ S), \\
(\mathrm{map}_n\ R)^{-1} &= \mathrm{map}_n\ (R^{-1}).
\end{aligned}$$

Using map one can give shunting laws for the generic wiring primitives:

Lemma 30:

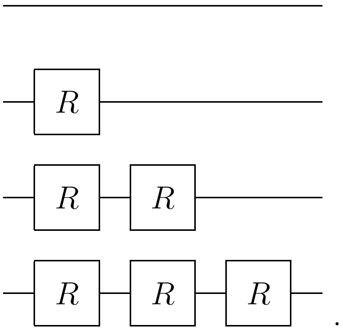
$$[R, \mathrm{map}_n\ R]\ ;\ apl_n\ =\ apl_n\ ;\ \mathrm{map}_{n+1}\ R,$$
$$[\mathrm{map}_n\ R, R]\ ;\ apr_n\ =\ apr_n\ ;\ \mathrm{map}_{n+1}\ R,$$
$$[\mathrm{map}_n\ R, \mathrm{map}_n\ S]\ ;\ zip_n\ =\ zip_n\ ;\ \mathrm{map}_n\ [R, S].$$

Similar to map is the *triangle* construction:

Definition 31:

$$\begin{array}{lll} \mathrm{tri}_0\ R & \widehat{=} & [], \\ \mathrm{tri}_{n+1}\ R & \widehat{=} & {apl_n}^{-1}\ ;\ [R^n, \mathrm{tri}_n\ R]\ ;\ apl_n. \end{array}$$

(Note that what we have defined above as tri is normally written as *irt* in Ruby.) For example, $\mathrm{tri}_4\ R = [R^3, R^2, R^1, R^0]$, or in pictures:



Triangle has similar properties to map:

Lemma 32:

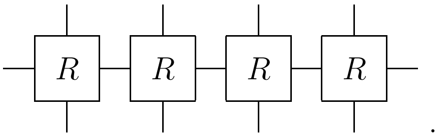
$$R\ ;\ S = S\ ;\ R\ \Rightarrow\ \mathrm{tri}_n\ R\ ;\ \mathrm{tri}_n\ S\ =\ \mathrm{tri}_n\ (R\ ;\ S),$$
$$(\mathrm{tri}_n\ R)^{-1}\ =\ \mathrm{tri}_n\ (R^{-1}).$$

The generic version of ↔ is the *row* construction:

Definition 33:

$$\begin{array}{lcl}
\mathrm{row}_1\ R & \widehat{=} & \mathrm{snd}\ apl_0{}^{-1}\ ;\ R\ ;\ \mathrm{fst}\ apl_0, \\
\mathrm{row}_{n+2}\ R & \widehat{=} & \mathrm{snd}\ apl_{n+1}{}^{-1}\ ;\ (R \leftrightarrow \mathrm{row}_{n+1}\ R)\ ;\ \mathrm{fst}\ apl_{n+1}.
\end{array}$$

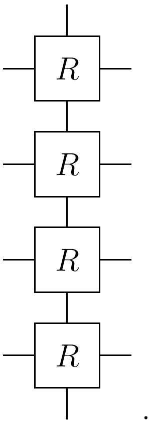
(Choosing $n = 1$ as the base–case for $\mathrm{row}_n\ R$ avoids some technical problems with types [38].) For example, here is a picture of $\mathrm{row}_4\ R$:



Just as $\updownarrow$ is dual to $\leftrightarrow$, so *col* is dual to row:

Definition 34: $\mathrm{col}_n\ R \ \widehat{=}\ (\mathrm{row}_n\ R^{-1})^{-1}$.

For example, here is a picture of $\mathrm{col}_4\ R$:

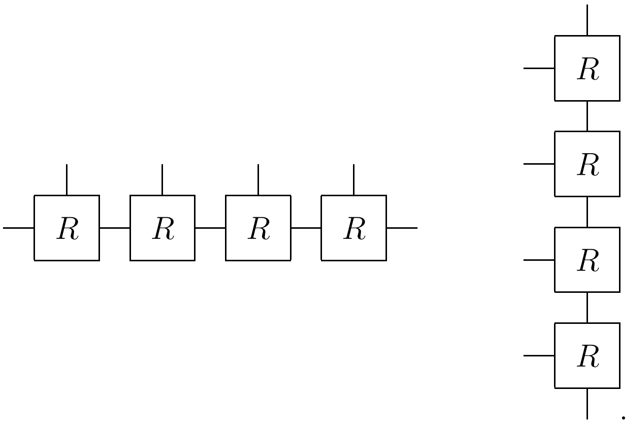


Relational versions of the familiar reduce (or fold) operators from functional programming [10] can be defined in Ruby by using row and col:

Definition 35:

$$\begin{array}{lcl}
\mathrm{rdl}_n\ R & \widehat{=} & \mathrm{row}_n\ (R\ ;\ fork)\ ;\ \pi_2, \\
\mathrm{rdr}_n\ R & \widehat{=} & \mathrm{col}_n\ (R\ ;\ fork)\ ;\ \pi_1.
\end{array}$$

Here are pictures of $\mathrm{rdl}_4\ R$ and $\mathrm{rdr}_4\ R$:



For example, if $\oplus$ is a function from pairs to values, then

$$(a,(b,c,d,e)) \;\; (\mathrm{rdl}_4 \oplus^r) \;\; x \;\; \equiv \;\; x \; = \; (((a \oplus b) \oplus c) \oplus d) \oplus e,$$

$$((a,b,c,d),e) \;\; (\mathrm{rdr}_4 \oplus^r) \;\; x \;\; \equiv \;\; x \; = \; a \oplus (b \oplus (c \oplus (d \oplus e))).$$

Laws for row, col, rdl and rdr are given in section 7.1.

Many circuits operate with streams of values rather than single values. Ruby can be used to design such circuits [59, 40]. A stream is modelled in Ruby as a function from integers to values. Given a relation R, the corresponding relation $\hat{R}$ on streams is defined by $a \; \hat{R} \; b \;\; \equiv \;\; \forall t \in \mathcal{Z}.\; a(t) \; R \; b(t)$. In practice the $\hat{(-)}$ operator is always left implicit in Ruby programs. Note that new definitions for the operators of Ruby are not needed when working with streams; the standard definitions suffice. A single new primitive is introduced, a unit–delay primitive $\mathcal{D}$:

Definition 36: $a \; \mathcal{D} \; b \;\; \hat{=} \;\; \forall t \in \mathcal{Z}.\; a(t-1) = b(t).$

$\mathcal{D}$ has a number of useful properties [59], but in this thesis we don't manipulate programs involving delays. In chapter 5 we present an interpreter for Ruby programs; within the interpreter time is infinite only in the positive direction, and we use a delay primitive $\mathcal{D}_s$ that is parameterised with a starting value s:

Definition 37: $a \; \mathcal{D}_s \; b \;\; \hat{=} \;\; b(0) = s \;\; \wedge \;\; \forall t \in \mathcal{N}.\; b(t+1) = a(t).$

Chapter 4

Causal Relations and Networks

The Ruby approach to circuit design is to derive implementations from specifications. Ruby has been used to derive many different kinds of circuits, but the notion of a Ruby program being an implementation has never been made precise, although the basic ideas can be found in [39]. In this chapter we define what it means for a program to be an implementation, using the notion of a *causal relation* and that of the *network* of relations denoted by a Ruby program.

Section 4.1 introduces the idea of a causal relation. Section 4.2 gives a categorical definition of causality, and some closure properties of causal relations. Section 4.3 introduces networks. And finally, section 4.4 defines 'implementation'.

4.1 Causal relations

Given a function $f : \mathcal{A} \rightarrow \mathcal{B}$, the corresponding functional relation $f^r : \mathcal{A} \leftrightarrow \mathcal{B}$ is defined by $a \; f^r \; b \;\equiv\; b = f\,a$. (In Ruby one always uses the same symbol for a function and its interpretation as a relation, but for our purposes here we prefer to make the difference explicit.) For example, $(x, y) \;\; +^r \;\; z \;\;\equiv\;\; z = x + y$. In this example, one can think of the x and y components as being *inputs* to the relation, and the z component being the *output*. This is not the only way to assign inputs and outputs to the $+^r$ relation however; either of x and y can also be viewed as the output component, since we have $(x, y) \;\; +^r \;\; z \;\;\equiv\;\; x = z - y$ and

$(x, y) \ +^r \ z \ \equiv \ y = z - x$. In this sense, the $+^r$ relation is functional in three different ways; $+^r$ is an example of what we term a *causal* relation.

> **Definition 38:** A relation is *causal* [33] if one can partition the components of each tuple in the relation into two classes, such that the first class (the *output* components) are functionally determined by the second class (the *input* components). Moreover, we require that the partition and function be the same for all tuples in the relation.

Causal relations are more general than functional relations in that the input components are not restricted to the domain of a causal relation, nor output components to the range. An example of a relation that is causal but not functional is $(+^r)^{-1}$. An example of a relation that is not causal is $\{(F, F), (T, F), (T, T)\}$.

Since there may be more than one way to pick input and output components for a causal relation, it is useful then to think of the set of *directions* for a causal relation. Whereas types tell us what kind of data a program expects, directions tell us which components of a causal relation can be inputs and which can be outputs. We write R^d for the set of directions for a relation R; relations that are not causal have $R^d = \emptyset$. For example, the $+^r$ relation has three directions:

$$(+^r)^d \ = \ \{((\mathit{in}, \mathit{in}), \mathit{out}), ((\mathit{in}, \mathit{out}), \mathit{in}), ((\mathit{out}, \mathit{in}), \mathit{in})\}.$$

We give below some closure properties for causal relations:

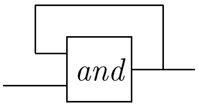
- If R is functional then R is causal.
- If $\mathcal{A}$ and $\mathcal{B}$ are sets, then the product $\mathcal{A} \times \mathcal{B}$ is causal.
- R is causal iff R^{-1} is causal. In fact, $(R^{-1})^d = (R^d)^{-1}$.
- R is causal and S is causal iff $[R, S]$ is causal. In fact, $[R, S]^d = [R^d, S^d]$.

Some of these properties are proved in section 4.2.3. Unfortunately, causal relations are not closed under composition. For example, both or^{-1} and and are causal relations, but their composition $\mathit{or}^{-1} \ ; \ \mathit{and} = \{(F, F), (T, F), (T, T)\}$ is not.

If a composition $R\ ;\ S$ is in fact causal, one might imagine that the ways in which it is causal are related to the ways in which the components R and S are causal, but this is not in general the case. Consider the following definition,

$$(R\ ;\ S)^{wd} \ = \ \{(a,c) \mid \exists b.\ a\ R^d\ b\ \wedge\ \neg b\ S^d\ c\},$$

where the $\neg$ operator gives the complement of a direction, replacing *in*s by *out*s and vice–versa. This definition demands that outputs from R coincide with inputs to S, and conversely, that inputs to R coincide with outputs from S. We say that $R\ ;\ S$ is *well–directed* if $(R\ ;\ S)^{wd} \neq \emptyset$. To see that $(R\ ;\ S)^d \neq (R\ ;\ S)^{wd}$, take $T = and^{-1}\ ;\ and$. This program is equivalent to the identity relation $\{(F,F),(T,T)\}$ on booleans, and hence $T^d = \{(in, out), (out, in)\}$, but $T^{wd} = \emptyset$, since there is a 'clash of inputs' between the two primitives. To see that not even the weaker $(R\ ;\ S)^{wd} \subseteq (R\ ;\ S)^d$ holds, take $T \ = \ {\pi_1}^{-1}\ ;\ \mathrm{snd}\,fork\ ;\ rsh\ ;\ \mathrm{fst}\ and\ ;\ fork^{-1}$. This program denotes the non–causal relation $\{(F,F),(T,F),(T,T)\}$, and hence $T^d = \emptyset$, but from $and^d = \{((in, in), out)\}$ and the following connectivity diagram for T,



,

we conclude that $T^{wd} = \{(in, out)\}$. (At which composition one breaks T to apply the wd definition is irrelevant, because $R = S \ \Rightarrow\ R^d = S^d$; directions are a property of the denotation of a program, rather than of the program itself.)

4.2 Causality categorically

In section 4.2.3 we give a categorical definition for the notion of a relation being causal, and prove some of the closure properties from section 4.1. We choose to work not within the category REL of relations, but within a category $\mathsf{rel(C)}$ whose arrows are subobjects of a binary product of objects in a suitable category C; this is a standard categorical approach to relations [16, 6, 20]. Section 4.2.1 reviews the notion of a subobject. Section 4.2.2 defines the category $\mathsf{rel(C)}$.

4.2.1 Subobjects

An arrow $f : A \to B$ is said to *factor through* an arrow $g : C \to B$ if there exists an $h : A \to C$ for which the following diagram commutes, that is, $f = g \circ h$.

$$\begin{tikzcd}
A \arrow[dd, "f"'] \arrow[dr, "h"] & \\
& C \arrow[dl, "g"] \\
B &
\end{tikzcd}$$

We write $f \approx g$ if each of f and g factors through the other. One can verify that $\approx$ is an equivalence relation on arrows with a common target; we write $[f]$ for the equivalence class of f under $\approx$. If f and g are monic arrows (recall that f is monic if $f \circ g = f \circ h \;\Rightarrow\; g = h$) then the factorising arrows implied by $f \approx g$ are unique, and are inverse isomorphisms. A *subobject* of an object A is defined to be an equivalence class under $\approx$ of monic arrows with target A.

How does this definition link with our intuition about sub–structures? Let us apply the definition in SET, the category of sets and functions. In SET the monic arrows are precisely the injective functions, so a subobject is an equivalence class of injections. Every subobject of A contains precisely one inclusion of a subset A into A itself, and that subset is the image of every element of that subobject; moreover, the subsets of A form a complete set of class representatives for the subobjects of A: each subobject in SET determines and is determined by a unique subset.

4.2.2 The category $\mathsf{rel(C)}$

Let C be a category with binary products and pullbacks. The category $\mathsf{rel(C)}$ of *relations over* C is defined as follows [16]:

- The objects of rel(C) are those of C.

- If $\langle f, g\rangle : A \to B \times C$ is a monic arrow in C, then the subobject $[\langle f, g\rangle]$ is an arrow $B \to C$ in rel(C).

- If $[\langle f, g\rangle] : A \to B$ and $[\langle h, i\rangle] : B \to C$ are arrows in rel(C), then the composition $[\langle f, g\rangle] \mathbin{;} [\langle h, i\rangle] : A \to C$ is the equivalence class of the monic part of an epi–monic factorisation of $\langle f \circ j, i \circ k\rangle$, where (j, k) is a pullback of (g, h). The following diagram is helpful when reading this definition.

$$\begin{tikzcd}
& & \cdot \arrow[dl, "j"'] \arrow[dr, "k"] & & \\
& \cdot \arrow[dl, "f"'] \arrow[dr, "g"] & & \cdot \arrow[dl, "h"'] \arrow[dr, "i"] & \\
A & & B & & C
\end{tikzcd}$$

- The identity on A in rel(C) is the subobject $[\langle id_A, id_A\rangle]$.

(The—rather technical—definition for composition is included above only for completeness; we don't have need to apply it in our proofs.) One can verify that rel(C) so defined does indeed form a category, and moreover, that rel(SET) is isomorphic to REL. (A similar construction is used by de Moor [20] to present and prove properties of a relational programming language derived from a functional programming language.) We make a few other definitions in rel(C):

- If $[\langle f, g\rangle] : A \to B$, then the *converse relation* $[\langle f, g\rangle]^{-1} : B \to A$ is $[\langle g, f\rangle]$.

- If $[\langle f, g\rangle] : A \to B$ and $[\langle h, i\rangle] : C \to D$, then the *product relation* $[\langle f, g\rangle] \times [\langle h, i\rangle] : A \times C \to B \times D$ is given by $[\langle f \times h, g \times i\rangle]$.

- The arrow $[\langle \pi_A, \pi_B\rangle] : A \to B$ is the *complete relation* between A and B.

4.2.3 Causality

Given an arrow $f : A \to B$ in C, the functional relation $f^r : A \to B$ in $\mathsf{rel}(\mathsf{C})$ is given by $f^r = [\langle id_A, f\rangle]$. This motivates the following definition: a relation $[\langle f, g\rangle] : A \to B$ in $\mathsf{rel}(\mathsf{C})$ is *functional* if there exists an arrow $h : A \to B$ in C for which $\langle id_A, h\rangle \in [\langle f, g\rangle]$, or equivalently, $\langle id_A, h\rangle \approx \langle f, g\rangle$. Note that $\langle id_A, h\rangle$ is monic for all h, so comparison with $\langle f, g\rangle$ under $\approx$ is acceptable.

Causal relations generalise functional relations in that inputs are not restricted to the domain of the relation, nor outputs to the range:

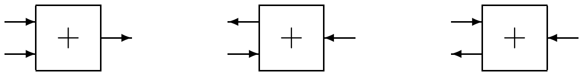
> **Definition 39:** A relation $[\langle f, g\rangle] : A \to B$ in $\mathsf{rel}(\mathsf{C})$ is *causal* if there exists an arrow $h : I \to O$ in C and an isomorphism $\alpha : I \times O \to A \times B$ in C, for which $\alpha \circ \langle id_I, h\rangle \in [\langle f, g\rangle]$, or equivalently $\alpha \circ \langle id_I, h\rangle \approx \langle f, g\rangle$. The pair (h, α) is called a *functional interpretation* of $[\langle f, g\rangle]$.

Note that comparison with $\langle f, g\rangle$ under $\approx$ is acceptable in this case because monics are closed under composition. One might like to add extra conditions to the causality definition to ensure that the isomorphism α be in some sense 'purely structural' (for example, requiring that α be a natural isomorphism or be constructed in a certain way) but such conditions are not needed to prove results 40 to 43 below.

In definition 39 one can think of the isomorphism $\alpha^{-1} : A \times B \to I \times O$ as being used to rearrange the components of the relation such that all the input components are in the domain and all the output components are in the range. An (informal) example is helpful; here are three functional interpretations for $+^r$:

$$\alpha^{-1}((x, y), z) = ((x, y), z),\ h(x, y) = x + y;$$
$$\alpha^{-1}((x, y), z) = ((z, x), y),\ h(x, z) = z - x;$$
$$\alpha^{-1}((x, y), z) = ((z, y), x),\ h(y, z) = z - y.$$

Here are some pictures which show the input and output components above:



.

In the remainder of this section we prove some of the closure properties given in section 4.1 We abbreviate "is a functional interpretation of" by $\in$.

Lemma 40: Functional relations are causal.

Proof: For a functional relation $[\langle id_A, f\rangle] : A \to B$, take $\alpha = id_{A\times B}$ and $h = f$.

$$(id_{A\times B}, h) \;\in\; [\langle id_A, h\rangle]$$

$$\equiv \qquad \{ \text{ def } \in \}$$

$$\underline{id_{A\times B} \circ \langle id_A, h\rangle} \;\approx\; \langle id_A, h\rangle$$

$$\equiv \qquad \{ \text{ products } \}$$

$$\langle id_A, h\rangle \;\approx\; \langle id_A, h\rangle$$

$$\equiv \qquad \{ \approx \text{ is reflexive } \}$$

$$\mathit{true}$$

Lemma 41: Complete relations are causal.

Proof: For a complete relation $[\langle \pi_A, \pi_B\rangle] : A \to B$, take $\alpha = \pi_1 : (A \times B) \times 1 \to A \times B$ and $h = !(A \times B) : A \times B \to 1$, where 1 is any terminal object in C, and $!X$ denotes the unique arrow $X \to 1$ in C. (Complete relations can be viewed as causal relations for which both the domain and range components are inputs.)

$$(\pi_1, !(A \times B)) \;\in\; [\langle \pi_A, \pi_B\rangle]$$

$$\equiv \qquad \{ \text{ def } \in \}$$

$$\underline{\pi_1 \circ \langle id_{A\times B}, !(A \times B)\rangle} \;\approx\; \langle \pi_A, \pi_B\rangle$$

$$\equiv \qquad \{ \text{ projections } \}$$

$$id_{A\times B} \approx \langle \pi_A, \pi_B\rangle$$

$$\equiv \qquad \{ \text{ products } \}$$

$$\mathit{true}$$

Lemma 42: Causal relations are closed under converse.

Proof: Let $(\alpha, h) \in [\langle f, g\rangle]$. Then $(\langle \pi_2, \pi_1\rangle \circ \alpha, h) \in [\langle f, g\rangle]^{-1}$.

$$
\begin{array}{ll}
 & (\langle \pi_2, \pi_1\rangle \circ \alpha, h) \;\in\; \underline{[\langle f, g\rangle]^{-1}} \\
\equiv & \qquad \{ \text{ def } \} \\
 & (\langle \pi_2, \pi_1\rangle \circ \alpha, h) \;\in\; [\langle g, f\rangle] \\
\equiv & \qquad \{ \text{ def} \in \} \\
 & \langle \pi_2, \pi_1\rangle \circ \alpha \circ \langle id, h\rangle \;\approx\; \underline{\langle g, f\rangle} \\
\equiv & \qquad \{ \text{ shunting } \} \\
 & \langle \pi_2, \pi_1\rangle \circ \underline{\alpha \circ \langle id, h\rangle} \;\approx\; \langle \pi_2, \pi_1\rangle \circ \underline{\langle f, g\rangle} \\
\Leftarrow & \qquad \{ \text{ Liebniz } \} \\
 & \alpha \circ \langle id, h\rangle \;\approx\; \langle f, g\rangle \\
\equiv & \qquad \{ \text{ def} \in \} \\
 & (\alpha, h) \;\in\; [\langle f, g\rangle]
\end{array}
$$

Lemma 43: Causal relations are closed under products.

Proof: Let $(\alpha, h) \in [\langle f, g\rangle]$ and $(\beta, k) \in [\langle i, j\rangle]$. Then $(\gamma \circ \alpha \times \beta \circ \gamma, h \times k) \in [\langle f, g\rangle] \times [\langle i, j\rangle]$, where $\gamma = \langle \pi_1 \times \pi_1, \pi_2 \times \pi_2\rangle$.

$$
\begin{array}{ll}
 & (\gamma \circ \alpha \times \beta \circ \gamma, h \times k) \;\in\; \underline{[\langle f, g\rangle] \times [\langle i, j\rangle]} \\
\equiv & \qquad \{ \text{ def } \} \\
 & (\gamma \circ \alpha \times \beta \circ \gamma, h \times k) \;\in\; [\langle f \times i, g \times j\rangle] \\
\equiv & \qquad \{ \text{ def} \in \} \\
 & \gamma \circ \alpha \times \beta \circ \gamma \circ \langle id, h \times k\rangle \;\approx\; \underline{\langle f \times i, g \times j\rangle} \\
\equiv & \qquad \{ \text{ shunting } \}
\end{array}
$$

$$
\begin{array}{ll}
 & \gamma \circ \underline{\alpha{\times}\beta \circ \gamma \circ \langle id, h{\times}k\rangle} \;\approx\; \gamma \circ \underline{\langle f,g\rangle{\times}\langle i,j\rangle} \\
\Leftarrow & \qquad \{ \text{ Liebniz } \} \\
 & \alpha{\times}\beta \circ \underline{\gamma \circ \langle id, h{\times}k\rangle} \;\approx\; \langle f,g\rangle \times \langle i,j\rangle \\
\equiv & \qquad \{ \text{ shunting } \} \\
 & \underline{\alpha{\times}\beta \circ \langle id,h\rangle{\times}\langle id,k\rangle} \;\approx\; \langle f,g\rangle \times \langle i,j\rangle \\
\equiv & \qquad \{ \text{ bifunctors } \} \\
 & (\alpha \circ \langle id,h\rangle) \times (\beta \circ \langle id,k\rangle) \;\approx\; \langle f,g\rangle \times \langle i,j\rangle \\
\Leftarrow & \qquad \{ \text{ products } \} \\
 & \alpha \circ \langle id,h\rangle \;\approx\; \langle f,g\rangle \;\;\wedge\;\; \beta \circ \langle id,k\rangle \;\approx\; \langle i,j\rangle \\
\equiv & \qquad \{ \text{ def } \in \} \\
 & (\alpha,h) \in [\langle f,g\rangle] \;\;\wedge\;\; (\beta,k) \in [\langle i,j\rangle]
\end{array}
$$

4.3 Networks

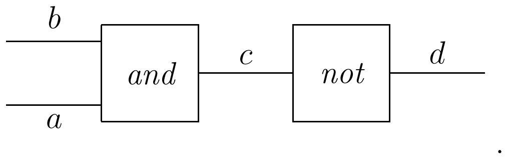
Ruby programs denote binary relations. Ruby programs also have a pictorial interpretation, although the process by which one makes a picture is mostly informal. In this section we introduce the related but formal notion of the *network* of relations denoted by a Ruby program. The difference between pictures and networks is that the layout of the nodes and wires has some significance in a picture, whereas in a network only the connectivity between nodes is important.

> **Definition 44:** Let V be a set of *wire names*. A *wire* is either an element of V, or a finite tuple of wires. A *node* is a triple $\langle D, P, R\rangle$ where D, R are wires (the domain and range wires for the node) and P is a binary relation. A *network* is a triple $\langle N, D, R\rangle$ where N is a set of nodes and D, R are wires (the domain and range wires for the network).

For example, the Ruby program ($and\ ;\ not$) denotes the network

$$\langle \{\langle\langle a,b\rangle, and, c\rangle, \langle c, not, d\rangle\}, \langle a,b\rangle, d\rangle.$$

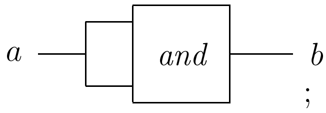
This network corresponds to the following picture:



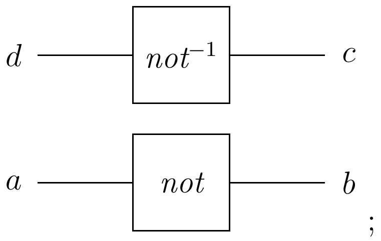
That the *and* and *not* relations are connected to one another is made explicit in the network by the repeated appearance of the wire c.

Here are some other examples of programs and their networks:

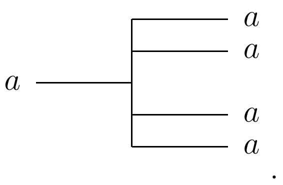
$$\mathit{fork} \;;\; \mathit{and} \;\mapsto\; \langle\{\langle\langle a,a\rangle, \mathit{and}, b\rangle\}, a, b\rangle$$



$$[\mathit{not}, \mathit{not}^{-1}] \;\mapsto\; \langle\{\langle a, \mathit{not}, b\rangle, \langle c, \mathit{not}, d\rangle\}, \langle a, d\rangle, \langle b, c\rangle\rangle$$



$$\mathit{fork} \;;\; \mathit{fork} \;\mapsto\; \langle\emptyset, a, \langle\langle a,a\rangle, \langle a,a\rangle\rangle\rangle$$



Ignoring naming of wires, a Ruby program denotes a unique network. Moreover, such a network contains sufficient information to calculate the relation denoted by the original Ruby program. In the remainder of this section we present an LML program which translates a Ruby program to a network; it is this program that forms the basis of the Ruby interpreter in chapter 5.

We begin by defining types for Ruby programs and networks:

```
type prog = PRIM prim + SWAP + FORK +
            SEQ prog prog + PAR prog prog + INV prog

type prim = AND + NOT

type network == (List node # wire # wire)

type node == (wire # prim # wire)

type wire = NAME Int + TUPLE (List wire)
```

(In reality of course, the `prog` and `prim` types have many more variants, but the few given above suffice here.) Our task now is to define a function

```
translate : (prog # Int) -> (network # Int)
```

that converts a program to a network. The integer argument is assumed to be a fresh wire name, as are all integers larger than this argument; an integer is returned by `translate` because some wire names may be consumed during translation. We ignore here the possibility that the translation process might fail.

We begin by showing how to translate the program `PRIM AND`:

```
translate (PRIM AND,x) =
   let dom = TUPLE [NAME x; NAME (x+1)]
   and rng = NAME (x+2)
   in (([(dom,AND,rng)],dom,rng),x+3)
```

Wiring primitives translate to networks that don't have any nodes. For example, `SWAP` is translated as follows:

```
translate (SWAP,x) =
   (([],TUPLE [NAME x;NAME (x+1)],
        TUPLE [NAME (x+1);NAME x]),x+2)
```

A program `INV p` is translated by first translating program `p`, then exchanging the resulting domain and range wires:

```
translate (INV p,x) =
   let ((ns,d,r),y) = translate (p,x) in ((ns,r,d),y)
```

A program `PAR p q` is translated by translating `p` and `q` independently, combining the resulting sets of nodes, and then tupling the domain and range wires:

```
translate (PAR p q,x) =
   let rec ((ns1,d1,r1),y) = translate (p,x)
   and ((ns2,d2,r2),z) = translate (q,y)
   in ((ns1@ns2,TUPLE[d1;d2],TUPLE[r1;r2]),z)
```

Translating `SEQ p q` is more complicated, since the range wire from the translation of `p` must be *joined* to the domain wire of the translation of `q`. This joining is handled by unifying the wires, that is, finding a most general substitution for the component wires names under which both wires are identical. (The algorithm that performs unification is well known [44], being used to implement polymorphic type inference in languages such as Miranda and Lazy ML.) Since these wire names might occur at other places in the network, the resulting substitution must be applied to the domain and range wires of each node in the translation of `SEQ p q`:

```
translate (SEQ p q,x) =
   let rec ((ns1,d1,r1),y) = translate (p,x)
   and ((ns2,d2,r2),z) = translate (q,y)
   and s = unify (r1,d2)
   and f (d,prm,r) = (apply s d, prm, apply s r)
   in ((map f (ns1 @ ns2), apply s d1, apply s r2),z)
```

Here are some example translations:

```
translate (SEQ (PRIM AND) (PRIM NOT),1) =
   (([(TUPLE [NAME 1;NAME 2],AND,NAME 4);
      (NAME 4,NOT,NAME 5)],
     TUPLE [NAME 1;NAME 2],NAME 5),6)

translate (SEQ FORK (PRIM AND),1) =
   (([(TUPLE [NAME 3;NAME 3],AND,NAME 4)],NAME 3,NAME 4),5)

translate (PAR (PRIM NOT) (INV (PRIM NOT)),1) =
   (([(NAME 1,NOT,NAME 2);(NAME 3,NOT,NAME 4)],
    TUPLE [NAME 1;NAME 4],TUPLE [NAME 2;NAME 3]),5)

translate (SEQ FORK FORK,1) =
   (([],NAME 1,TUPLE [TUPLE [NAME 1;NAME 1];
                      TUPLE [NAME 1;NAME 1]]),3)
```

4.4 Implementations

In this section we define what it means for a Ruby program to be an implementation, by defining what it means for a network to be executable.

Informally, a network is executable if, given a value for some of the external wires (the inputs), the value of each remaining wire in the network (in particular, the external outputs) can be computed as functions of the values of wires. Such executable networks are like functional data-flow networks [45] in that the values of wires are computed as functions of the values of other wires, but more general in that we don't restrict input wires to the left-side of our nodes, nor output wires to the right-side, and thus for some networks there may be several possible choices of input and output wires. The extra generality is motivated by the intention that executable networks represent circuits. A typical circuit has data flowing in many

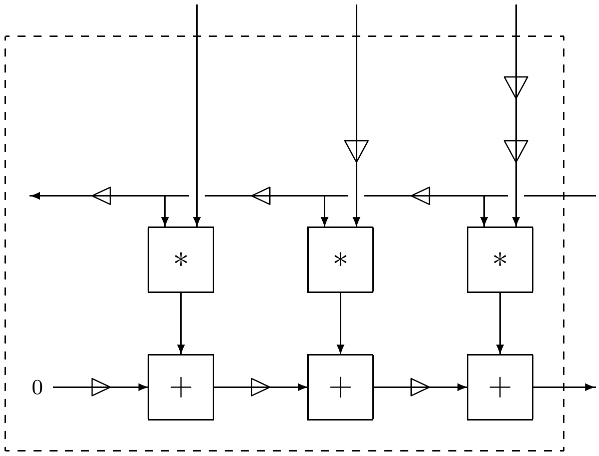


Figure 4.1: A 2-slow systolic correlator

different directions: data-flow is functional, but data is not restricted to flowing from left to right across the circuit board. An example is a '2-slow systolic correlator' [59]. Data in this circuit flows both from left-to-right and from right-to-left, as shown in figure 4.4. (The triangles in this figure represent delay elements.)

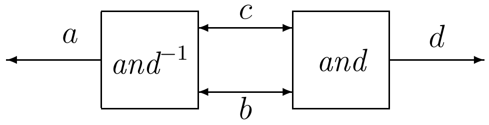
> **Definition 45:** A Ruby program is an *implementation* if it denotes a network that is executable. A network is *executable* if every occurrence of a wire name in the network can be labelled *in* or *out* such that the following three constraints are satisfied:
>
> - The labelling of the domain and range wires for each primitive relation R in the network corresponds to a direction from R^d.
> - Each wire name in the network is labelled *out* precisely once.
> - There are no cyclic dependencies in the network. That is, the value of an input wire to a primitive must not depend upon the value of an output wire from the same primitive.

(For the domain and range wires of each node in a network, a labelling x : *in* means that the node uses the value of wire x, and x : *out* means that the node sets the value of wire x. For the domain and range wires of the network as a whole, $x : \mathit{out}$ means that the outside world must set the value of wire x, and $x : \mathit{in}$ means that it can read the value of wire x.) Executable networks are like functional data-flow networks, except that the nodes are causal relations rather than functions. If a program is an implementation then it necessarily denotes a causal relation. Since a causal relation can be functional in more than one way, there may be more than one way in which the network for an implementation is executable.

> **Definition 46:** Let R and S be Ruby programs. We say that R is *implementable* (able to be implemented) if it denotes a causal relation. We say that R *implements* S if the programs R and S denote the same relation, and R is an implementation.

We illustrate definitions 45 and 46 with some examples. We begin with three programs $R1$, $R2$, and $R3$ that denote the identity relation $\mathit{id}_B = \{(F, F), (T, T)\}$ on the booleans. The relation id_B is causal, with $(\mathit{id}_B)^d = \{(\mathit{in}, \mathit{out}), (\mathit{out}, \mathit{in})\}$, and is hence implementable. Program $R1 = \mathit{and}^{-1}\ ;\ \mathit{and}$ can be viewed as a specification for the relation id_B. Given that $\mathit{and}^d = \{((\mathit{in}, \mathit{in}), \mathit{out})\}$ there is only one candidate for a properly labelled network for $R1$,

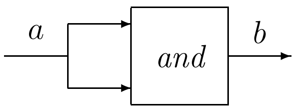
$$\langle\{\langle\langle b : \mathit{in}, c : \mathit{in}\rangle, \mathit{and}, a : \mathit{out}\rangle, \langle\langle b : \mathit{in}, c : \mathit{in}\rangle, \mathit{and}, d : \mathit{out}\rangle\}, a : \mathit{in}, d : \mathit{in}\rangle$$



,

but it is not acceptable, because wires b and c are never labelled *out*. The program $R2 = \mathit{fork}\ ;\ \mathit{and}$ is an implementation of $R1$, because in the labelled network

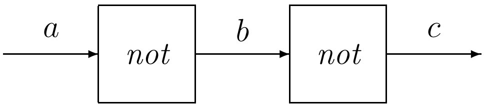
$$\langle\{\langle\langle a : \mathit{in}, a : \mathit{in}\rangle, \mathit{and}, b : \mathit{out}\rangle\}, a : \mathit{out}, b : \mathit{in}\rangle$$



each wire is labelled *out* precisely once. Note however that the network for $R2$ is only executable from domain to range, while the relation id_B denoted by $R2$ is functional also from range to domain.

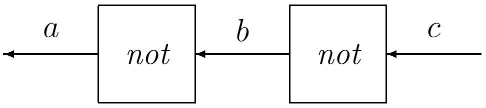
The program $R3 = \mathit{not}\ ;\ \mathit{not}$ is also an implementation of $R1$. Given that $\mathit{not}^d = \{(\mathit{in}, \mathit{out}), (\mathit{out}, \mathit{in})\}$, there are four possible labelled networks for $R3$:

$$\langle \{ \langle a : \mathit{in}, \mathit{not}, b : \mathit{out} \rangle, \langle b : \mathit{in}, \mathit{not}, c : \mathit{out} \rangle \}, a : \mathit{out}, c : \mathit{in} \rangle$$



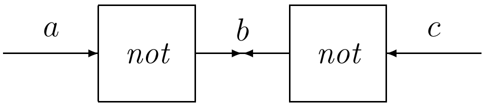
;

$$\langle \{ \langle a : \mathit{out}, \mathit{not}, b : \mathit{in} \rangle, \langle b : \mathit{out}, \mathit{not}, c : \mathit{in} \rangle \}, a : \mathit{in}, c : \mathit{out} \rangle$$



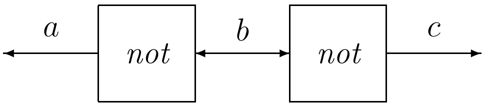
;

$$\langle \{ \langle a : \mathit{in}, \mathit{not}, b : \mathit{out} \rangle, \langle b : \mathit{out}, \mathit{not}, c : \mathit{in} \rangle \}, a : \mathit{out}, c : \mathit{out} \rangle$$



;

$$\langle \{ \langle a : \mathit{out}, \mathit{not}, b : \mathit{in} \rangle, \langle b : \mathit{in}, \mathit{not}, c : \mathit{out} \rangle \}, a : \mathit{in}, c : \mathit{in} \rangle$$



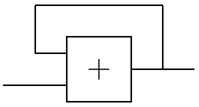
.

Only the first and second of these networks are acceptable; in the third network wire b is labelled *out* more than once, and in the fourth network wire b is never labelled *out*. Since $R3$ is executable in both ways that the specification $R1$ is functional, we say that $R3$ is a *full* implementation of $R1$.

We turn now to an example of a program that denotes a cyclic network:

$$R \;=\; {\pi_1}^{-1}\ ;\ \mathrm{snd}\,\mathit{fork}\ ;\ \mathit{rsh}\ ;\ \mathrm{fst}\,+\ ;\ \mathit{fork}^{-1}.$$

We showed in section 3.4 that R has the following connectivity:



.

There are two possible labelled networks for R:

$$\langle \{ \langle \langle a : in, b : in \rangle, +, b : out \rangle \}, a : out, b : in \rangle,$$

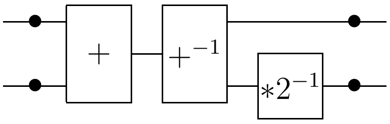
$$\langle \{ \langle \langle a : out, b : in \rangle, +, b : in \rangle \}, a : in, b : out \rangle.$$

The second is acceptable, and tells us that program R is executable from range to domain. The first satisfies the requirement that each wire be labelled *out* precisely once, but not the requirement that there be no cyclic dependencies: the output of the $+$ primitive is connected to one of its own inputs. If we are working with a version of Ruby in which relations are between streams of values rather than just single values, such a cyclic dependency can be removed by placing a delay $\mathcal{D}_s$ somewhere in the feedback path. For example, the program

$$R' \; = \; {\pi_1}^{-1} \; ; \; \mathrm{snd}\, fork \; ; \; rsh \; ; \; [+, {\mathcal{D}_s}^{-1}] \; ; \; fork^{-1}$$

is executable from domain to range, because the input to $+$ at time t depends upon its output at time $t-1$ rather than t. Note that R' is not executable from range to domain, because delays are only functional from domain to range.

In section 7.3 we introduce the term *representation changer* for a Ruby program R specified by the requirement that $R = f \; ; \; g^{-1}$ for functional relations f and g. In practice, many such specifications will already be implementations according to definition 45, but not executable in the direction that they are intended to be used. For example, let $f = [b, b] \; ; \; +$ and $g = [b \; ; \; *2, b] \; ; \; +$, where $b = \{(0,0), (1,1)\}$ is the identity relation on $\{0,1\}$. Both f and g are functional relations; f gives the sum of two bits, and g converts a 2-bit binary number to the corresponding natural number. Consider now the program $HA = f \; ; \; g^{-1}$:



.

The program HA is a specification for a 2-bit adder (a *half-adder*) that gives the binary carry and sum of a pair of bits. In fact HA is already an implementation, with any three of the external wires in the network for HA being inputs, and the remaining wire being the output. The program HA is not however an implementation in the direction in which we want to use it, namely from domain to range. It is in this sense that HA is a specification of a half-adder.

We conclude this section with two final examples:

- ${\pi_1}^{-1}$; $+$ is a full implementation of the cartesian product $\mathcal{Z} \times \mathcal{Z}$ on the integers. The direction set for both programs is $\{(\mathit{in}, \mathit{in})\}$.
- ${\pi_1}^{-1}$; snd $\{(a, a)\}$; $+$ and $(+a)$ are full implementations of one another. The direction set for both programs is $\{(\mathit{in}, \mathit{out}), (\mathit{out}, \mathit{in})\}$.

Chapter 5

The Ruby Interpreter

In chapter 4 we defined what it means for a Ruby program to be an implementation. The definition is general, but not completely formal. In this chapter we give a formal definition for a natural sub-class of the implementations, and present an interpreter for such Ruby programs. Interpreters for functional versions of Ruby have been made in the past; see for example [57, 46]. Our interpreter is the first that does not require that the Ruby program be in fact a functional program.

Section 5.1 defines the class of networks that the interpreter executes. Section 5.2 introduces the interpreter by means of a number of worked examples. Section 5.3 gives some reference material about the interpreter. And finally, section 5.4 suggests some ways in which the interpreter could be extended.

5.1 Executable networks

We say that a wire name x *occurs in* a node $\langle D, P, R\rangle$ if x occurs in either of the wires D and R. A wire name occurs in a network $\langle N, D, R\rangle$ if it occurs in any of the nodes N, or in either of the wires D and R. If $\langle N, D, R\rangle$ is a network and x is a wire name that occurs in this network, then x is called *external* if it occurs in D or R, and *internal* otherwise. Finally, the *dependents* for a node $\langle D, P, R\rangle$ in a network is the set of wire names given by the union of the names in D and the dependents for all nodes $\langle D', P', R'\rangle$ in the network for which the wire names in R'

and those in D are not disjoint. There is one special case: the dependents of a node $\langle D, P, R \rangle$ for which P is a delay primitive is the empty-set $\emptyset$. We can now define the class of networks that the Ruby interpreter is able to execute:

Definition 47: A network is executable if:

- For every node $\langle D, P, R \rangle$ the relation P is functional;
- For each internal (external) wire name x there is precisely one (at most one) node $\langle D, P, R \rangle$ for which x occurs in the range wire R, and moreover, x must occur only once in this R;
- For each node $\langle D, P, R \rangle$ the dependents for this node and the wire names in R are disjoint sets.

That is, an executable network is a network of functions for which each wire name is an output wire name for precisely one function (or at most one function if the wire name is external), and for which there are no cyclic dependencies between the input and output wire names to any function in the network.

Figures 5.1 and 5.2 give examples of executable and non-executable networks.

5.2 Worked examples

In this section we illustrate the features of the Ruby interpreter by working through a number of example programs. The Ruby interpreter is written in the functional language *Lazy ML* (LML), and is used under the interactive LML system. Both the Ruby interpreter and the LML compiler are available by anonymous ftp from Chalmers University (internet address `ftp.cs.chalmers.se` or `129.16.225.66`) in directories `pub/misc/ruby` and `pub/haskell/chalmers` respectively.

To begin with, we load the interactive LML system:

```
graham% lmli
```

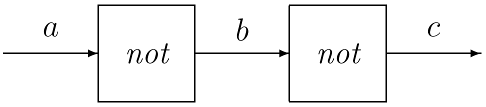
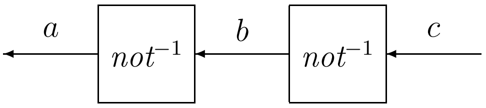
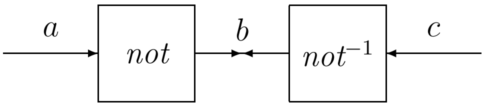
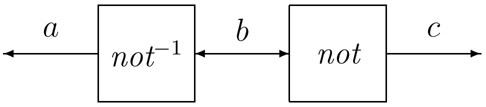
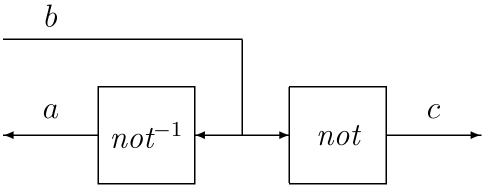
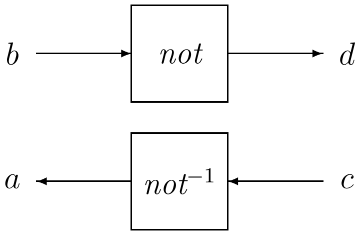
Term	Network	Executable
$\mathit{not}\ ;\ \mathit{not}$	 $\langle\{\langle a, \mathit{not}, b\rangle, \langle b, \mathit{not}, c\rangle\}, a, c\rangle$	yes
$\mathit{not}^{-1}\ ;\ \mathit{not}^{-1}$	 $\langle\{\langle b, \mathit{not}, a\rangle, \langle c, \mathit{not}, b\rangle\}, a, c\rangle$	yes
$\mathit{not}\ ;\ \mathit{not}^{-1}$	 $\langle\{\langle a, \mathit{not}, b\rangle, \langle c, \mathit{not}, b\rangle\}, a, c\rangle$	no (b is driven twice)
$\mathit{not}^{-1}\ ;\ \mathit{not}$	 $\langle\{\langle b, \mathit{not}, a\rangle, \langle b, \mathit{not}, c\rangle\}, a, c\rangle$	no (b is undriven)
fst $(\mathit{not}^{-1})\ ;\ \mathit{fork}^{-1}\ ;\ \mathit{not}$	 $\langle\{\langle b, \mathit{not}, a\rangle, \langle b, \mathit{not}, c\rangle\}, \langle a, b\rangle, c\rangle$	yes
$[\mathit{not}^{-1}, \mathit{not}]$	 $\langle\{\langle c, \mathit{not}, a\rangle, b, \mathit{not}, d\}, \langle a, b\rangle, \langle c, d\rangle\rangle$	yes

Figure 5.1: Examples of executable and non-executable Ruby terms

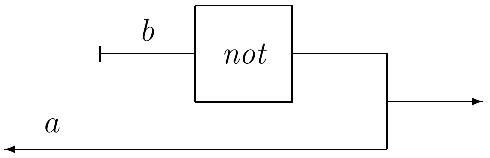
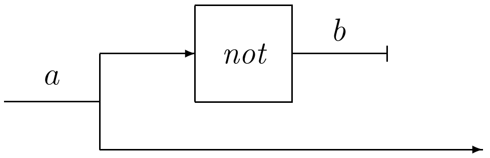
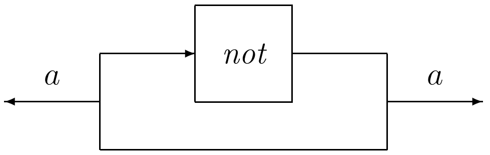
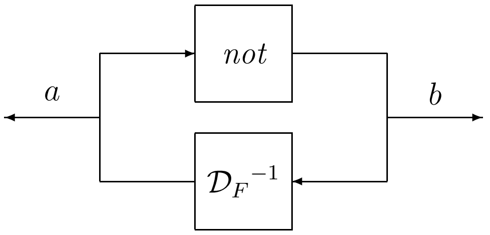
Term	Network	Executable
${\pi_1}^{-1}$; snd *not* ; $fork^{-1}$	 $\langle\{\langle b, not, a\rangle\}, a, a\rangle$	no (b is undriven)
fork ; snd *not* ; π_1	 $\langle\{\langle a, not, b\rangle\}, a, a\rangle$	yes
fork ; snd *not* ; $fork^{-1}$	 $\langle\{\langle a, not, a\rangle\}, a, a\rangle$	no (cyclic dependency)
fork ; $[{\mathcal{D}_F}^{-1}, not]$; $fork^{-1}$	 $\langle\{\langle a, not, b\rangle, \langle b, \mathcal{D}_F, a\rangle\}, a, b\rangle$	yes

Figure 5.2: More examples of executable and non-executable Ruby terms

```
Welcome to interactive LML version 0.999.4 SPARC 1993 Mar 16!
Loading prelude... 348 values, 41 types found.
Type "help;" to get help.
```

Now we load the Ruby interpreter:

```
> source "rubysim";

+------------------------------+
|    ‘The Ruby Interpreter’    |
|                              |
| Copyright 1993 Graham Hutton |
|    graham@cs.chalmers.se     |
+------------------------------+
```

(In the above, ‘`>`’ is the LML prompt, and ‘`;`’ is the command terminator.)

Ruby programs are compiled using the `rc` function, which translates a program to a network. The function `rc` takes an argument of type `prog`, which is the LML type of Ruby programs. What the Ruby programmer need know about the `prog` type is explained in section 5.3. (The main differences from standard Ruby notation is that composition is written as `..`, converse is written as `inv`, and product is written as `!!`.) As a first example, let us compile the program *not* ; *not*:

```
> rc (NOT .. NOT);

   Name      Domain              Range
   ----------------------------------
   NOT       w1                  w2
   ----------------------------------
   NOT       w2                  w3
   ----------------------------------
```

```
Primitives   -  2
Delays       -  0
Longest path -  2
Parallelism  -  0%

Directions -  in ~ out

Wiring -  w1 ~ w3

Inputs -  w1
```

In this example, the network has two nodes: $\langle w1, not, w2\rangle$ and $\langle w2, not, w3\rangle$. Below the network, `Wiring` gives the domain and range wires for the network as a whole, `Inputs` tells which of these wire names are inputs (all other external wire names are outputs), and `Directions` is derived from `Wiring` by replacing each wire name with `in` or `out` as appropriate. `Primitives` and `Delays` are the number of non-delay and delay primitives in the network. `Longest path` is the length of the longest path through the network that does not include a delay primitive. `Parallelism` gives an absolute measure of the available concurrency in the network, being the ratio of the total number of primitives in the network (both delay and non-delay) to the length of the longest path, scaled to a percentage.

The most recently compiled Ruby program is stored in a file `ruby-prog`, and is executed using the `rsim` function. This function takes a string as its argument, containing a value for each of the input wires as named in the `Wiring` part below the network. You can supply more than one set of input values, with successive sets of input values being separated by a semi-colon. For example,

```
> rsim "F;T";

   0 -  F ~ F
   1 -  T ~ T
```

verifies that *not* ; *not* denotes the identity relation on booleans. As shown in this example, for each set of input values supplied, the `rsim` function gives the `Wiring` part for the network, with wire names replaced by the values obtained by executing the network with these input values. So the output `0 -  F ~ F` above says that supplying wire `w1` with value `F` resulted in wire `w2` having value `F`. The number 0 indicates that this is the result for the first set of input values.

Here are some other examples from figures 5.1 and 5.2:

```
> rc (NOT .. inv NOT);

        ERROR: multiple output to single wire

> rc (inv NOT .. NOT);

        ERROR: undriven internal input

> rc (first (inv NOT) .. inv fork .. NOT);

   Name      Domain              Range
   ----------------------------------
   NOT       w1                  w2
   NOT       w1                  w3
   ----------------------------------

   Primitives   -  2
   Delays       -  0
   Longest path -  1
   Parallelism  -  100%

   Directions -  <out,in> ~ out
   Wiring -  <w2,w1> ~ w3

   Inputs -  w1

> rc (fork .. second NOT .. inv fork);

        ERROR: unbroken loop in {NOT}

> rc (fork .. (inv (bdel false) !! NOT) .. inv fork);

   Name      Domain            Range
   ----------------------------------
   NOT       w1                w2
   ----------------------------------
   D_F       w2                w1
   ----------------------------------

   Primitives   -  1
   Delays       -  1
   Longest path -  2
   Parallelism  -  0%

   Directions -  out ~ out

   Wiring -  w1 ~ w2

   Inputs -  none
```

LML can be used as a meta-language to define new primitives and combining forms in terms of those pre-defined by the interpreter. For example, a program sort2 that sorts a pair of numbers can be defined and compiled as follows:

```
> let sort2 = fork .. (MIN !! MAX);

sort2: prog

> rc sort2;

   Name      Domain             Range
   ----------------------------------
   MIN       <w1,w2>            w3
   MAX       <w1,w2>            w4
   ----------------------------------

   Primitives   -  2
   Delays       -  0
   Longest path -  1
   Parallelism  -  100%

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

   Wiring -  <w1,w2> ~ <w3,w4>

   Inputs -  w1 w2
```

For example,

```
> rsim "4 7";

    0 -  (4,7) ~ (4,7)

> rsim "7 4";

    0 -  (7,4) ~ (4,7)
```

As well as supplying numbers as inputs to `sort2`, we can supply *symbolic values*, which are just strings. Using symbolic values allows us to see how the outputs from the program are constructed in terms of the inputs:

```
> rsim "a b";

   0 -  (a,b) ~ (a min b,a max b)
```

Let us aim now to define a program that sorts n numbers, rather than just 2. We begin by using `sort2` to define a generic primitive `minim` that takes an n-tuple ($n > 1$) of numbers, and returns a pair comprising the minimum number and an ($n - 1$)-tuple of the remaining numbers:

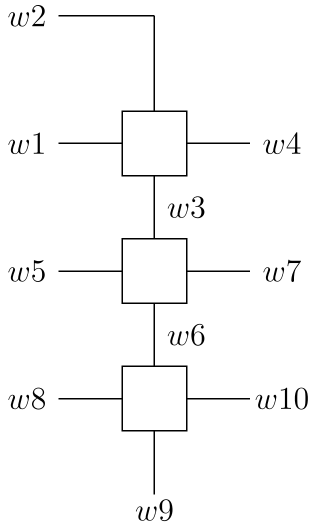
```
> let minim n = inv (apr (n-1)) .. col (n-1) sort2;

minim: Int->prog
```

For example,

```
> rc (minim 4);

   Name      Domain              Range
   ----------------------------------
   MIN       <w1,w2>             w3
   MAX       <w1,w2>             w4
   ----------------------------------
   MIN       <w5,w3>             w6
   MAX       <w5,w3>             w7
   ----------------------------------
   MIN       <w8,w6>             w9
   MAX       <w8,w6>             w10
   ----------------------------------
```

```
Primitives   -  6

Delays       -  0

Longest path -  3

Parallelism  -  20%


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


Wiring -  <w8,w5,w1,w2> ~ <w9,<w10,w7,w4>>


Inputs -  w8 w5 w1 w2
```

Notice that the primitives in the network above are divided into blocks, separated by dashed lines. Each block contains all the primitives whose output depends only upon external inputs, and outputs of primitives in earlier blocks. (Operationally this means that blocks must be executed sequentially, from the first to the last. All the primitives within a block can however be executed in parallel, since they are independent of one another.) Note also that the `Longest path` is just the number of blocks in the network for the program. A picture of the network above is helpful in understanding the definition for `minim` (the boxes represent `sort2`):



Executing the network with symbolic values confirms that the first component of the result is the minimum of the 4 input values:

```
> rsim "a b c d";

         0 -  (a,b,c,d) ~ (a min (b min (c min d)),
                           (a max (b min (c min d)),
                            b max (c min d),
                            c max d))
```

Note that the name `sort2` does not appear in the network for `minim 4`, but rather its definition has been unfolded at each instance. We can prevent such unfolding and treat `sort2` as a new primitive by using the function `NAME` of type `String -> prog -> prog`. For example, if we make the definitions

```
> let sort2 = NAME "sort2" (fork .. (MIN !! MAX));

sort2: prog

> let minim n = inv (apr (n-1)) .. col (n-1) sort2;

minim: Int->prog
```

then the compilation produces the following result: (we have to define `minim` again because the existing version refers to the old binding for `sort2`)

```
> rc (minim 4)

        Name      Domain              Range
        ----------------------------------
        "sort2"   <w1,w2>             <w3,w4>
        ----------------------------------
```

```
"sort2"   <w5,w3>            <w6,w7>
----------------------------------
"sort2"   <w8,w6>            <w9,w10>
----------------------------------


Primitives   -  6
Delays       -  0
Longest path -  3
Parallelism  -  20%

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

Wiring -  <w8,w5,w1,w2> ~ <w9,<w10,w7,w4>>

Inputs -  w8 w5 w1 w2
```

Using the NAME function can reduce compilation time, particularly when named programs are used as arguments to generic combining forms. A named program is compiled once and its network instantiated at each instance, rather than the definition being unfolded at each instance and hence compiled many times.

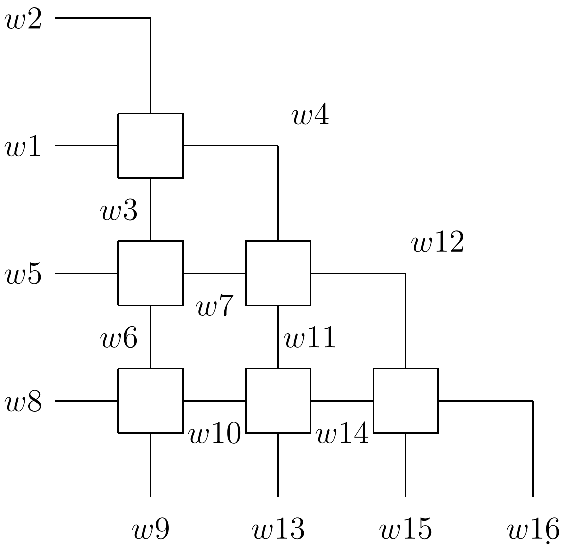
Using minim we can define a sorting program. An n-tuple ($n > 0$) of numbers can be sorted by first selecting the minimum number, and then recursively sorting the remaining $(n-1)$-tuple of numbers. A 1-tuple of numbers requires no sorting, and forms the base-case for the recursive definition:

```
> let rec mysort 1 = par [rid]
  ||      mysort n = minim n .. second (mysort (n-1)) .. apl (n-1);


mysort: Int->prog
```

Let us compile a sorter for 4 numbers:

```
> rc (mysort 4);

   Name      Domain            Range
   ----------------------------------
   "sort2"   <w1,w2>           <w3,w4>
   ----------------------------------
   "sort2"   <w5,w3>           <w6,w7>
   ----------------------------------
   "sort2"   <w8,w6>           <w9,w10>
   "sort2"   <w7,w4>           <w11,w12>
   ----------------------------------
   "sort2"   <w10,w11>         <w13,w14>
   ----------------------------------
   "sort2"   <w14,w12>         <w15,w16>
   ----------------------------------

   Primitives   -  12
   Delays       -  0
   Longest path -  5
   Parallelism  -  12%

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

   Wiring -  <w8,w5,w1,w2> ~ <w9,w13,w15,w16>

   Inputs -  w8 w5 w1 w2
```

Here is a picture of this network:



For example,

```
   > rsim "4 2 3 1";

       0 -  (4,2,3,1) ~ (1,2,3,4)

   > rsim "a 3 1 2";

       0 -  (a,3,1,2) ~ (a min 1,
                          (a max 1) min 2,
                          ((a max 1) max 2) min 3,
                          ((a max 1) max 2) max 3)
```

Note from the last example that symbolic values are not simplified. We can see however that the output 4-tuple in this example is equal to `(a min 1, (a max 1) min 2, (a max 2) min 3, a max 3)`, which makes clear how the symbolic value ‘`a`’ is routed to one of the 4 components in the output tuple.

To finish off, we take a closer look at wiring primitives. Networks produced by the interpreter have two kinds of wires. *Monomorphic* wires start with the letter `w` and are restricted to carrying boolean, integer, and symbolic values; *polymorphic* wires start with p and can also carry tuples of such values. The wiring primitives of Ruby (*id*, *fork*, $\pi_1, \ldots$) have networks with polymorphic wires, and can be used to defined other programs with polymorphic wires. For example,

```
> let swap = fork .. (p2 !! p1);

   swap : prog

> rc swap;

   Wiring -  <p1,p2> ~ <p2,p1>

   Inputs -  p1 p2
```

Note that the polymorphic primitive fork is being used here to duplicate a pair of values. Since the wires in the network produced above are polymorphic, they are not restricted to carrying just basic values. For example, we can swap pairs:

```
> rsim "(a,b) (c,d)";

    0 -  ((a,b),(c,d)) ~ ((c,d),(a,b))
```

New wiring primitives (like swap) need not be defined in terms of the existing wiring primitives, but can also be defined directly using the function wiring of type (expr # expr) -> prog. Values of type expr are built using two functions: wire of type Int -> expr and list of type List expr -> expr. An example shows how these three functions are used to define wiring primitives:

```
> let swap = wiring (list [wire 1; wire 2], list [wire 2; wire 1]);

   swap: prog
```

```
> rc swap;

   Wiring -  <p1,p2> ~ <p2,p1>

   Inputs -  p1 p2
```

5.3 Converting Ruby terms to LML syntax

In this section we explain the LML syntax for Ruby terms as used by the Ruby interpreter. Because of syntactic constraints imposed by LML, some Ruby primitives and combining forms have different names from normal in LML syntax:

Ruby	LML
$r \; ; \; s$	`r .. s`
$[r, s]$	`r !! s`
r^{-1}	`inv r`
r^n	`repeat n r`
id	`rid`
π_1	`p1`
π_2	`p2`
fst r	`first r`
snd r	`second r`
map$_n$ r	`rmap n r`
zip_n	`rzip n`
$r \leftrightarrow s$	`r $beside s`
$r \updownarrow s$	`r $below s`

Two programs are placed in parallel using `!!`. For other than two programs use the function `par`, which takes a list of programs as its argument; for example, $[r, s, t]$ in Ruby becomes `par [r; s; t]` in LML. Note that all infix Ruby combining forms have the same precedence when written in LML notation, and associate to the right. For example, `r !! s .. t` would be interpreted as `r !! (s .. t)`.

Delays are made using one of three functions (`bdel`, `idel`, or `sdel`), depending on whether the starting value is boolean, integer, or symbolic. Constant relations are made using `bcon`, `icon`, or `scon`. For example, the delay $\mathcal{D}_5$ is written as `idel 5` in LML and the constant relation $\{(T, T)\}$ is written as `bcon true`.

A number of logical and arithmetic functions are built-in to the interpreter:

$$\texttt{AND}\ \langle a, b\rangle \ \Leftrightarrow\ a \wedge b$$

$$\texttt{OR}\ \langle a, b\rangle \ \Leftrightarrow\ a \vee b$$

$$\texttt{NOT}\ a \ \Leftrightarrow\ \neg a$$

$$\texttt{LT}\ \langle m, n\rangle \ \Leftrightarrow\ m < n$$

$$\texttt{GT}\ \langle m, n\rangle \ \Leftrightarrow\ m > n$$

$$\texttt{EQ}\ \langle m, n\rangle \ \Leftrightarrow\ m = n$$

$$\texttt{IF}\ \langle b, \langle x, y\rangle\rangle = \begin{cases} x & \text{if } b = true \\ y & \text{if } b = false \end{cases}$$

$$\texttt{BTOI}\ b = \begin{cases} 0 & \text{if } b = false \\ 1 & \text{if } b = true \end{cases}$$

$$\texttt{ITOB}\ n = \begin{cases} false & \text{if } n = 0 \\ true & \text{if } n = 1 \end{cases}$$

$$\texttt{MUX n}\ \langle i, xs\rangle = xs_i \quad \{0 \leq i < n\}$$

$$\texttt{ADD}\ \langle m, n\rangle = m + n$$

$$\texttt{SUB}\ \langle m, n\rangle = m - n$$

$$\texttt{MULT}\ \langle m, n\rangle = m * n$$

$$\texttt{DIV}\ \langle m, n\rangle = max\ \{i \mid n * i \leq m\}$$

$$\texttt{MOD}\ \langle m, n\rangle = m - n * (m\ div\ n)$$

$$\texttt{EXP}\ \langle m, n\rangle = m^n$$

$$\texttt{LOG}\ \langle m, n\rangle = max\ \{i \mid i^n \leq m\}$$

$$\texttt{MAX}\ \langle m, n\rangle = max\ \{m, n\}$$

$$\texttt{MIN}\ \langle m, n\rangle = min\ \{m, n\}$$

$$\texttt{GCD}\ \langle m, n\rangle = max\ \{i \mid m\ mod\ i\ =\ n\ mod\ i\ =\ 0\}$$

$$\texttt{FAC}\ n = 1 * 2 * \ldots * n$$

The wiring primitives of Ruby are not built-in to the interpreter, but are defined as ordinary LML definitions using the `wiring` function:

```
rid = wiring (wire 1,wire 1)

p1 = wiring (list [wire 1;wire 2],wire 1)

p2 = wiring (list [wire 1;wire 2],wire 2)

fork = wiring (wire 1,list [wire 1;wire 1])

rsh = wiring (list [wire 1;list [wire 2;wire 3]],
              list [list [wire 1;wire 2];wire 3])

lsh = wiring (list [list [wire 1;wire 2];wire 3],
              list [wire 1;list [wire 2;wire 3]])

swap = wiring (list [wire 1;wire 2],list [wire 2;wire 1])

rev n = let vs = map wire (1 $to n)
        in wiring (list vs,list (reverse vs))

apl n = let vs = map wire (1 $to (n+1))
        in wiring (list [hd vs; list (tl vs)], list vs)

apr n = let vs = map wire (1 $to (n+1))
        in wiring (list [list (head n vs); last vs], list vs)

distl n = let vs = map wire (1 $to (n+1))
          in wiring (list [hd vs; list (tl vs)],
                     list [list [hd vs;x] ;; x <- tl vs])

distr n = let vs = map wire (1 $to (n+1))
          in wiring (list [list (head n vs); last vs],
                     list [list [x;last vs] ;; x <- head n vs])

flatr n = let f e es = LIST [e;es]
          and vs = map wire (1 $to n)
          in wiring (foldr1 f vs, LIST vs)

pair n = let rec vs = map wire (1 $to (2*n))
         and pairup [] = []
         ||  pairup (x.y.ys) = list [x;y] . pairup ys
         in wiring (list vs, list (pairup vs))

halve n = let vs = map wire (1 $to (2*n))
          in wiring (list vs, list [list (head n vs); list (tail n vs)])
```

```
rzip n = let rec vs = map wire (1 $to (2*n))
         and (v1,v2) = (head n vs, tail n vs)
         and zipped = [list [x;y] ;; (x,y) <- v1 $zip v2]
         in wiring (list [list v1;list v2], list zipped)
```

Of the combining forms of Ruby only composition, converse, and product are built-in to the interpreter. All the other combining forms, including all generic combining forms, are defined just as ordinary LML definitions:

```
first r = r !! rid

second r = rid !! r

repeat n r = foldr (..) rid (rept n r)

rmap n r = par (rept n r)

r $beside s = rsh .. first r .. lsh .. second s .. rsh

r $below s = inv (inv r $beside inv s)

row n r = second (inv (flatr n))
          .. foldr1 ($beside) (rept n r)
          .. first (flatr n)

col n r = inv (row n (inv r))

grid (m,n) r = row m (col n r)

rdl n r = row n (r .. inv p2) .. p2

rdr n r = col n (r ..inv p1) .. p1

tri n r = par [repeat x r ;; x <- 0 $to (n-1)]

irt n r = rev n .. tri n r .. rev n
```

5.4 Extending the interpreter

In this section we suggest some ways to improve the Ruby interpreter.

5.4.1 Simplifying symbolic expressions

Symbolic simulation is useful in seeing how result values are built up inside a program. For example, applying a sorting program built using `max` and `min` (in fact, Batcher's *bitonic sorter* [60]) to `(5,1,3,a)` might give the following result:

```
((1 min (a max 3)) min (5 min (a min 3)),
 (1 min (a max 3)) max (5 min (a min 3)),
 (1 max (a max 3)) min (5 max (a min 3)),
 (1 max (a max 3)) max (5 max (a min 3)))
```

It is only after manually simplifying this result to

```
(1 min a, 1 max (a min 3), (a max 3) min 5, a max 5)
```

that it becomes clear how '`a`' can be routed to any of the four output positions. Having the interpreter itself make such simplifications would be very useful.

5.4.2 Causal primitives

Ruby wiring primitives are treated properly as causal relations within the interpreter, being able to be used in any way in which they are functional. The arithmetic and logical primitives however are restricted to being used from domain to range as functional relations. For example, the `NOT` primitive cannot be used from range to domain even though it is functional in this way. It would be interesting to implement all primitive relations properly as causal relations. Such an extension would make the interpreter more flexible, but gives no extra power: using just the functional primitives one can bend wires around to get components that are functional in other ways than from domain to range. For example, the $((out, in), in)$ instance of $+$ can be expressed by $\mathrm{fst}\ (-)^{-1}\ ;\ lsh\ ;\ \mathrm{snd}\ fork^{-1}\ ;\ \pi_1$.

5.4.3 A relational interpreter

The interpreter at present works only with Ruby programs that are implementations. It would be useful to be able to simulate arbitrary Ruby programs. If external wires are constrained to carrying only a small number of values (as is typical of many Ruby programs) it might be possible that such information could be used to reduce the otherwise (potentially) huge search space during a relational simulation.

5.4.4 Type inference

The type checking done by the interpreter at present is very basic; all that is checked is that wires are connected together in an acceptable way. One could implement an ML-style type inference system for Ruby, the only complication being that Ruby programs can contain *size* variables. For example, consider programs

$$\begin{aligned} \mathit{apl}_n &: (\alpha, \alpha^n) \leftrightarrow \alpha^{n+1}, \\ \mathit{halve}_m &: \beta^{2m} \leftrightarrow (\beta^m, \beta^m). \end{aligned}$$

In the above typings, α and β are type variables, while α^n is the type of an n-tuple of elements of type α. Deducing the type $(\gamma, \gamma^{2t+1}) \leftrightarrow (\gamma^{t+1}, \gamma^{t+1})$ for the composition $\mathit{apl}_n \,;\, \mathit{halve}_m$ means, in addition to the normal work of type inference, having to solve the size equation $n + 1 = 2m$. At first glance one might think that simple rules of arithmetic could be applied, giving solutions $n = 2m - 1$ and $m = (n+1)/2$. There is a problem however: size variables range over the natural numbers. Putting $m = 0$ in the first solution gives $n = -1$, which is not an acceptable size; putting $n = 0$ in the second solution gives $m = 1/2$, which is again not an acceptable size. In fact, the most general solution to $n+1 = 2m$ is given by $n = 2t+1$ and $m = t+1$, where t is a fresh size variable.

Equations whose variables range over the natural numbers are called *Diophantine equations*. It is known to be undecidable whether or not an arbitrary Diophantine equation has a solution. (Note that such problems do not arise with equations whose variables range over the integers rather than the naturals.) Provided that we work only with simple equations however, such as those of the form $ax+b=cy+d$, algorithms to compute the most general solution do exist.

Polymorphic type inference is a delicate algorithm (having in fact worse than exponential time complexity in the worst case.) Making types more informative, for example by adding size information, is very tempting, but one always has to be careful not to end up with a type inference algorithm that is intractably slow. It seems likely however that restricting ourselves to linear size expressions will give a reasonable trade–off between types that are informative and a tractable inference algorithm. Programs that naturally have types with non–linear sizes, for example $+\!\!+_{n,m} \ : \ (\alpha^n, \alpha^m) \leftrightarrow \alpha^{n+m}$ can be given valid but less informative types; for example, replace $n+m$ above by a fresh size variable p. Alternatively, such programs could be restricted to being used at instances in which their sizes are in fact linear, as is the case for $\mathit{fork} \ ; \ +\!\!+_{n,n} \ : \ \alpha^n \leftrightarrow \alpha^{2n}$.

In a Ruby program, some size arguments to generic primitives and combining forms might be redundant, being able to be deduced by examining the context in which the generic appears in the program. For example, the most general way to fill in the missing sizes in the program $\mathit{apl} \ ; \ \mathrm{map}\ R$ is as $\mathit{apl}_n \ ; \ \mathrm{map}_{n+1}\ R$. So–called *unsized* generic combining forms can be defined in terms of normal generic combining forms. For example, $\mathrm{map}\ R$ can be defined by $\bigcup \, \{\mathrm{map}_n\ R \mid n \in \mathcal{N}\}$. Alternatively, one can give a direct recursive definition; for example,

$$\begin{array}{cccll} () & (\mathrm{map}\ R) & () & \equiv & \mathit{true} \\ (x.xs) & (\mathrm{map}\ R) & (y.ys) & \equiv & x\ R\ y \ \wedge \ xs\ (\mathrm{map}\ R)\ ys \end{array}$$

One can admit unsized generics in program by (prior to type inference) replacing each instance by a sized generic with a fresh size variable.

Here are some examples of programs before and after such a type inference:

$$\begin{array}{lcl} [R,S] \ ; \ \mathrm{map}\ T & \mapsto & [R,S] \ ; \ \mathrm{map}_2\ T, \\[1ex] \mathit{fork} \ ; \ [\mathrm{map}_{n+1}\ R, \mathrm{map}\ S] & \mapsto & \mathit{fork} \ ; \ [\mathrm{map}_{n+1}\ R, \mathrm{map}_{n+1}\ S], \\[1ex] \mathit{apl} \ ; \ \mathrm{map}\ R & \mapsto & \mathit{apl}_n \ ; \ \mathrm{map}_{n+1}\ R, \\[1ex] \mathit{apl} \ ; \ \mathit{halve} & \mapsto & \mathit{apl}_{2n+1} \ ; \ \mathit{halve}_{n+1}. \end{array}$$

One might ask why a type of *sized lists* is not provided in any of the standard functional languages? The reason is that sizes of tuples in Ruby programs are fixed at compile–time (Ruby programs denote static networks), whereas sizes of lists in functional programs can depend upon run–time values. For example, the length of the list produced by the functional program *filter p* depends not only upon the length of the argument list, but upon the elements of the argument list.

Chapter 6

Pers and Difunctionals

In this chapter we introduce the idea of *partial equivalence relations* (pers) as types in the relational calculus. Pers are now adopted as types in both the spec calculus [70, 71] and Ruby [43, 38]. In the spec calculus it was the desire to have 'types with laws' that motivated working with pers; in Ruby the choice of pers as types arose naturally from experience in deriving programs. Section 6.1 introduces the theory of pers. Section 6.2 introduces the *difunctional* relations, which generalise pers in a natural way. Our interest in pers and difunctionals is in their application in chapter 7 to deriving programs. For a more theoretical treatment, including proofs of many of the results that we state, see Voermans forthcoming thesis [70].

6.1 Partial equivalence relations

Recall that a *partial equivalence relation* (per) on a set $\mathcal{S}$ is a symmetric and transitive relation on $\mathcal{S}$; such a per that is also reflexive on $\mathcal{S}$ is an *equivalence relation*. Throughout this chapter A, B, C denote pers, and we omit the 'on $\mathcal{S}$' part when talking of pers. The notion of being a per has a simple point–free formulation:

Definition 48: $per.A \quad \hat{=} \quad A = A^{-1} \;\; \wedge \;\; A \,;\, A \subseteq A.$

The first term in this definition expresses symmetry, the second transitivity:

Lemma 49: $symmetric.R \;\; \equiv \;\; R = R^{-1}.$

Lemma 50: $transitive.R \;\equiv\; R \,;\, R \subseteq R.$

For example, the transitivity result is proved as follows:

$$
\begin{array}{ll}
& R \,;\, R \;\subseteq\; R \\
\equiv & \qquad \{ \text{ def} \subseteq \} \\
& x \,(R \,;\, R)\, z \;\Rightarrow\; x \,R\, z \\
\equiv & \qquad \{ \text{ def ; } \} \\
& \exists y.\, (x \,R\, y \;\wedge\; y \,R\, z) \;\Rightarrow\; x \,R\, z \\
\equiv & \qquad \{ \text{ predicate calculus } \} \\
& x \,R\, y \;\wedge\; y \,R\, z \;\;\Rightarrow\;\; x \,R\, z
\end{array}
$$

Two other formulations for *per.A* prove useful:

Lemma 51: $per.A \;\equiv\; A = A^{-1} \;\wedge\; A \,;\, A = A.$

Lemma 52: $per.A \;\equiv\; A = A \,;\, A^{-1}.$

Here is a proof of the non-immediate part of lemma 51:

$$
\begin{array}{ll}
& A \,;\, A \\
\supseteq & \qquad \{ \text{ } transitive.A \text{ } \} \\
& A \,;\, A \,;\, A \\
= & \qquad \{ \text{ } symmetric.A \text{ } \} \\
& A \,;\, A^{-1} \,;\, A \\
\supseteq & \qquad \{ \text{ } triple\ rule\text{; see lemma 84 } \} \\
& A
\end{array}
$$

For some (fixed) per A and an element $a \in dom\ A$, the *equivalence class* $[a]$ of a under A is defined by $[a] = \{b \mid a \ A \ b\}$. We write $\mathcal{S}^2$ for the full relation $\mathcal{S} \times \mathcal{S}$. Using these two definitions, here are three well–known properties of pers: (Proofs of point–free versions of these properties can be found in [35].)

Lemma 53: For any per A,

(a) Equivalence classes are either identical or disjoint:

$$[a] = [b] \quad \vee \quad [a] \cap [b] = \emptyset.$$

(b) A is completely determined by its equivalence classes:

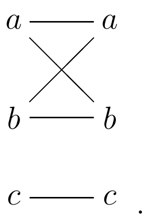
$$A = \bigcup \{[a]^2 \mid a \in \mathit{dom}\, A\}.$$

(c) Decomposition in terms of disjoint full relations is unique:

Let $\mathcal{X}$ be a set of disjoint non–empty sets. Then,

$$A = \bigcup \{\mathcal{S}^2 \mid \mathcal{S} \in \mathcal{X}\} \quad \equiv \quad \mathcal{X} = \{[a] \mid a \in \mathit{dom}\, A\}.$$

In summary, "pers can be written in a unique way as the union of disjoint non–empty full relations, each such relation representing an equivalence class of the per." For example, the per $\{(a, a), (a, b), (b, a), (b, b), (c, c)\}$ can be written in terms of full relations as $\{a, b\}^2 \cup \{c\}^2$; here is a picture:



We observe that pers are closed under converse, intersection, and par, but not under union or composition. To see that pers are not closed under $\cup$, take $A = \{a, b\}^2$ and $B = \{b, c\}^2$; then $A \cup B = \{a, b, c\}^2 - \{(a, c), (c, a)\}$ is not a per, where "$-$" means set difference. For composition, take $A = \{a, b\}^2 \cup \{c\}^2$ and $B = \{a\}^2 \cup \{b, c\}^2$; then $A \mathbin{;} B = \{a, b, c\}^2 - \{(c, a)\}$ is not a per.

6.1.1 Orderings

In this section we introduce three orderings on pers.

We begin by defining operators $\vdash$ and $\dashv$ that are useful in defining other constructions involving pers. If a relation S can be composed on the right of R without affecting R we say that R *right–guarantees* S, written $R \vdash S$:

Definition 54: $R \vdash S \quad \widehat{=} \quad R \mathbin{;} S = R.$

Here are some properties of $\vdash$ [42]:

Lemma 55: $R \vdash S \quad \Rightarrow \quad (T \mathbin{;} R) \vdash S.$

Lemma 56: $R \vdash (S \mathbin{;} T) \quad \Rightarrow \quad (R \mathbin{;} S) \vdash (T \mathbin{;} S).$

Lemma 57: $R \vdash (S \mathbin{;} T) \ \wedge \ (T \mathbin{;} U) \vdash V \quad \Rightarrow \quad (R \mathbin{;} U) \vdash V.$

The *left–guarantees* operator $\dashv$ is dual to $\vdash$:

Definition 58: $S \dashv R \quad \widehat{=} \quad S \mathbin{;} R = R.$

Lemma 59: $S \dashv R \quad \equiv \quad R^{-1} \vdash S^{-1}.$

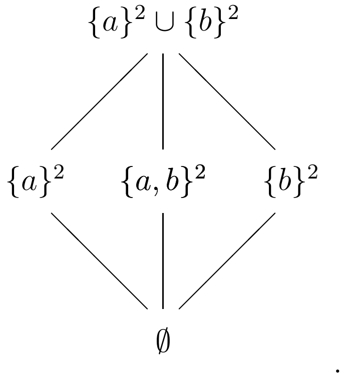
We write $\vdash$ as $\lhd$ when both arguments are pers:

Definition 60: $A \lhd B \quad \widehat{=} \quad A \vdash B.$

Pers are partially ordered by $\lhd$; in fact we have the following result:

Lemma 61: Pers form a complete lattice under $\lhd$.

The least element in this complete lattice of pers is $\perp\!\!\perp$, the greatest element is *id*. See Voermans [69] or van der Woude [65] for details of the $\bigvee$ and $\bigwedge$ operators for $\lhd$. For example, here is the lattice of pers on the universe $\{a, b\}$:



The per lattice is in general neither complemented nor distributive. Recall that a lattice is complemented if every element A has a unique element B satisfying $A \cup B = \top\!\!\top$ and $A \cap B = \bot\!\!\bot$. Complements exist in the per lattice, but are not unique; for example, both $\{a,b\}^2$ and $\{b\}^2$ in the example lattice above satisfy the properties expected of a complement to $\{a\}^2$. The per lattice is distributive if $A \bigvee (B \bigwedge C) = (A \bigvee B) \bigwedge (A \bigvee C)$. Taking $A = \{a\}^2$, $B = \{a,b\}^2$, and $C = \{b\}^2$ in the lattice above shows that this equation does not hold: $\{a\}^2 \neq \{a\}^2 \cup \{b\}^2$.

In [35] it is shown that $A \lhd B$ expresses that equivalence classes in A are the union of equivalence classes in B, i.e. that A is formed by discarding, copying, and combining classes from B. We read $\lhd$ then as "*is a per on*". For example,

$$\{a,b,c\}^2 \cup \{d,e\}^2 \;\; \lhd \;\; \{a,b\}^2 \cup \{c\}^2 \cup \{d,e\}^2 \cup \{f\}^2 \,;$$

that is, the equivalence class $\{a,b,c\}$ in A is formed by combining the equivalence classes $\{a,b\}$ and $\{c\}$ from B, while the equivalence class $\{d,e\}$ is copied unchanged from B to A, and $\{f\}$ in B is discarded when moving from B to A.

Two other orderings on pers are defined using $\lhd$:

Definition 62: $A \leq B \;\; \widehat{=} \;\; A \lhd B \;\wedge\; A \subseteq B$.

Definition 63: $A \mathrel{\lhd\!|} B \;\; \widehat{=} \;\; A \lhd B \;\wedge\; A \supseteq B$.

Pers form a meet semi–lattice under $\leq$, and a join semi–lattice under $\lhd\!|$. Writing $A \leq B$ says that A is formed from B by copying and discarding equivalence classes, but no classes are combined; read $\leq$ as "*is a sub–per of*" [35]. Writing $A \lhd\!| \; B$ says that A is formed from B by copying or combining equivalence classes, but no classes are discarded; read $\lhd\!|$ as "*is an equivalence relation on*" [35]. For example,

$$\{a,b\}^2 \cup \{c\}^2 \;\leq\; \{a,b\}^2 \cup \{c\}^2 \cup \{d,e\}^2,$$

$$\{a,b\}^2 \cup \{c\}^2 \;\lhd\!|\; \{a\} \cup \{b\} \cup \{c\}.$$

6.1.2 Pers as types

In relational calculus it is common to write $R \subseteq \mathcal{A} \times \mathcal{B}$ in the form of a typing judgement $R \in \mathcal{A} \leftrightarrow \mathcal{B}$. Beginning with the observation that $R \in \mathcal{A} \leftrightarrow \mathcal{B}$ is equivalent to $id_{\mathcal{A}} \dashv R \;\wedge\; R \vdash id_{\mathcal{B}}$, where $id_{\mathcal{X}}$ denotes the identity relation on a set $\mathcal{X}$, Backhouse [4] explores the use of identity relations as types in relational calculus. In this section we introduce the idea of using pers as types.

For a relation $R \subseteq \mathcal{A} \times \mathcal{B}$, it may be the case that many elements in $\mathcal{A}$ have the same image under R, or conversely, that many elements in $\mathcal{B}$ have the same inverse image under R. It is the desire that such information be able to be encoded in types that motivates working with partial equivalence relations as types rather than with identity relations as types. We begin with the following definition:

Definition 64: A *left domain* of R is a per A for which $A \dashv R$.

The condition $A \dashv R$ expresses that equivalent elements under A have the same image under R, and that A is big enough in that $dom\ A \supseteq dom\ R$ [35]. A relation can have many left domains; in section 6.1.3 we define an operator $<$ that gives the smallest such domain with respect to the $\lhd$ ordering on pers.

Of course, there is also the dual notion of a right domain:

Definition 65: A *right domain* of R is a left domain of R^{-1}, or equivalently, a right domain of R is a per B for which $R \vdash B$.

We write $R \in A \leftrightarrow B$ when A is a left domain of R and B is a right domain:

Definition 66: $R \in A \leftrightarrow B \;\; \widehat{=} \;\; A \dashv R \;\wedge\; R \vdash B$.

The two parts of this definition can be combined in a single equality:

Lemma 67: $R \in A \leftrightarrow B \;\; \equiv \;\; A \,;\, R \,;\, B = R$.

One can now give type inference rules for Ruby. Here are some examples:

Lemma 68:

$$R \in A \leftrightarrow B \;\wedge\; S \in B \leftrightarrow C \;\;\Rightarrow\;\; R \;;\; S \;\in\; A \leftrightarrow C,$$

$$R \in A \leftrightarrow B \;\;\equiv\;\; R^{-1} \in B \leftrightarrow A,$$

$$R \in A \leftrightarrow B \;\wedge\; S \in C \leftrightarrow D \;\;\Rightarrow\;\; [R, S] \;\in\; [A, C] \leftrightarrow [B, D],$$

$$R \in A \leftrightarrow B \;\;\Rightarrow\;\; \mathrm{map}_n\, R \;\in\; \mathrm{map}_n\, A \;\leftrightarrow\; \mathrm{map}_n\, B,$$

$$R \;\in\; [A, B] \leftrightarrow [C, A] \;\;\Rightarrow\;\; \mathrm{row}_n\, R \;\in\; [A, \mathrm{map}_n\, B] \;\leftrightarrow\; [\mathrm{map}_n\, C, A],$$

$$R \;\in\; [A, B] \leftrightarrow A \;\;\Rightarrow\;\; \mathrm{rdl}_n\, R \;\in\; [A, \mathrm{map}_n\, B] \;\leftrightarrow\; A.$$

These rules are consequences of familiar properties of the combining forms. For example, the proof of the rule for map relies on a distributivity property:

$$
\begin{array}{ll}
 & \mathrm{map}_n\, R \;\in\; \mathrm{map}_n\, A \;\leftrightarrow\; \mathrm{map}_n\, B \\
\equiv & \qquad \{ \text{ lemma 67 } \} \\
 & \mathrm{map}_n\, A \;;\; \mathrm{map}_n\, R \;;\; \mathrm{map}_n\, B \;=\; \mathrm{map}_n\, R \\
\equiv & \qquad \{ \text{ distribution (29) } \} \\
 & \mathrm{map}_n\, (A \;;\; R \;;\; B) \;=\; \mathrm{map}_n\, R \\
\Leftarrow & \qquad \{ \text{ Leibniz } \} \\
 & A \;;\; R \;;\; B \;=\; R \\
\equiv & \qquad \{ \text{ lemma 67 } \} \\
 & R \in A \leftrightarrow B
\end{array}
$$

6.1.3 Domain operators

In this section we introduce operators ${<}$ and ${>}$ that give the *best* left and right types for a relation. The reader is referred to [69, 65, 71] for proofs of the results that we give in this section. We begin with the following definition:

Definition 69: $a\; \acute{R}\; b \;\; \hat{=} \;\; \forall x.\, (a\; R\; x \;\equiv\; b\; R\; x)$.

Elements a and b are related by $\acute{R}$ if they have the same image under R. As usual, for calculation we prefer a point–free formulation. Using that

$$a\;(R/S)\;b \;\;\equiv\;\; \forall x.\, (b\; S\; x \;\Rightarrow\; a\; R\; x),$$

we see that $\acute{R}$ can be expressed using factors:

Lemma 70: $\acute{R} \;=\; (R/R) \cap (R/R)^{-1}$.

One can verify now that $\acute{R}$ is a left domain of R, and moreover, that it is the largest left domain of R under the inclusion ordering $\subseteq$ on relations. Not surprisingly, $\acute{R}$ is also the $\lhd$–smallest equivalence relation (on the implicit universe) that is a left domain of R. (We note in passing that $\acute{R}$ is a special case of Freyd and Scedrov's *symmetric division* operator; see section 2.35 in [28].)

The definition for $\acute{R}$ does not require that image sets are non–empty. Discarding from $\acute{R}$ the equivalence class comprising elements with empty image sets (that is, elements outwith *dom* R) gives a relation that we write as $R{<}$:

Definition 71: $a\; R{<}\; b \;\; \hat{=} \;\; a, b \in \mathit{dom}\, R \;\wedge\; a\; \acute{R}\; b$.

Again we can make a point–free version:

Lemma 72: $R{<} \;=\; (R\,;\, R^{-1}) \cap \acute{R}$.

One can verify now that $R{<}$ is the $\lhd$–smallest left domain of R:

Lemma 73: $R{\scriptstyle<} \ \dashv \ R.$

Lemma 74: $A \dashv R \ \Rightarrow \ R{\scriptstyle<} \lhd A.$

Reflecting its unique status, we refer to $R{\scriptstyle<}$ as *the* left domain of R.

Here are some properties of the ${\scriptstyle<}$ operator:

Lemma 75: $(A \mathbin{;} R){\scriptstyle<} \ \lhd \ A.$

Lemma 76: $(R \mathbin{;} S){\scriptstyle<} \ \lhd \ R{\scriptstyle<}.$

Lemma 77: $(R \mathbin{;} S){\scriptstyle<} \ \lhd \ (R \mathbin{;} S{\scriptstyle<}){\scriptstyle<}.$

Lemma 78: $A \dashv R \ \equiv \ R{\scriptstyle<} \lhd A.$

Lemma 79: $\mathit{per}.R \ \equiv \ R{\scriptstyle<} = R.$

Lemma 80: $[R, S]{\scriptstyle<} \ = \ [R{\scriptstyle<}, S{\scriptstyle<}].$

Of course, there is also a domain operator ${\scriptstyle>}$ that gives the $\lhd$–smallest right domain of a relation. The ${\scriptstyle>}$ operator is defined as the dual to ${\scriptstyle<}$:

Definition 81: $R{\scriptstyle>} \ \widehat{=} \ (R^{-1}){\scriptstyle<}$

In chapter 7 we find that the preconditions to ‘induction’ laws that are applied when deriving programs using Ruby often work out to be assertions of the form $A \dashv R$. In the past, such assertions have been verified by informal arguments or by using predicate calculus, rather than by applying algebraic laws from Ruby. In chapter 7 we verify such assertions without stepping outside Ruby, by first expressing them in the equivalent form $R{\scriptstyle<} \lhd A$ using lemma 78, which can then be verified using the algebraic properties given above for the ${\scriptstyle<}$ operator.

6.2 Difunctional relations

In this section we introduce the *difunctional* relations, a class of relations that generalise pers in a natural way. Difunctionals were first studied by Riguet in the late 1940's [54]. They have also been called 'regular' relations and 'pseudo–invertible' relations [36], and 'abstractions' [61]. Section 6.2.1 shows that the domain operators have a simple formulation for difunctional relations, and gives conditions (involving types) under which difunctionals are closed under composition. Sections 6.2.2 to 6.2.4 present different ways to think about difunctionals.

The notion of a relation being difunctional is defined as follows:

Definition 82: $\mathit{difun}.R \;\;\widehat{=}\;\; a\,R\,b \;\wedge\; c\,R\,b \;\wedge\; c\,R\,d \;\;\Rightarrow\;\; a\,R\,d.$

In calculation we always use a point–free formulation:

Lemma 83: $\mathit{difun}.R \;\;\equiv\;\; R\,;R^{-1}\,;R \;\subseteq\; R.$

Proof

$$
\begin{array}{ll}
 & R\,;R^{-1}\,;R \;\subseteq\; R \\
\equiv & \qquad \{\ \text{def} \subseteq\ \} \\
 & a\,(R\,;R^{-1}\,;R)\,d \;\Rightarrow\; a\,R\,d \\
\equiv & \qquad \{\ \text{def}\ ;\ \} \\
 & \exists b,c.(a\,R\,b \;\wedge\; b\,R^{-1}\,c \;\wedge\; c\,R\,d) \;\;\Rightarrow\;\; a\,R\,d \\
\equiv & \qquad \{\ \text{predicate calculus}\ \} \\
 & a\,R\,b \;\wedge\; b\,R^{-1}\,c \;\wedge\; c\,R\,d \;\;\Rightarrow\;\; a\,R\,d \\
\equiv & \qquad \{\ \text{def}\ ^{-1}\ \} \\
 & a\,R\,b \;\wedge\; c\,R\,b \;\wedge\; c\,R\,d \;\;\Rightarrow\;\; a\,R\,d
\end{array}
$$

Every relation satisfies the reverse inclusion, which we call the *triple rule* [35]:

Lemma 84: $R\,;R^{-1}\,;R \;\supseteq\; R$

Proof

$\underline{R\,;R^{-1}}\,;R$

$\supseteq$ { lemma 72 }

$R{<}\,;R$

$=$ { domains }

R

The two previous results are combined in a single equality:

Lemma 85: $\mathit{difun}.R \quad\equiv\quad R\,;R^{-1}\,;R \;=\; R.$

There are larger but equivalent formulations. For example,

Lemma 86: $\mathit{difun}.R \quad\equiv\quad R\,;R^{-1}\,;R\,;R^{-1}\,;R \;=\; R.$

Proof "⇒"

$R\,;R^{-1}\,;R\,;R^{-1}\,;R$

$=$ { $\mathit{difun}.R$ }

$R\,;R^{-1}\,;R$

$=$ { $\mathit{difun}.R$ }

R

Proof "⇐"

$R\,;R^{-1}\,;R$

$\subseteq$ { triple rule (84) }

$R\,;R^{-1}\,;R\,;R^{-1}\,;R$

$=$ { assumption }

R

In general, for all naturals $n > 0$:

Lemma 87: $\mathit{difun}.R \quad \equiv \quad R \,;\, (R^{-1} \,;\, R)^n \;=\; R.$

For $a \in \mathit{dom}\ R$, the *image* of a under R is defined by $a.R = \{b \mid a \;R\; b\}$. The *inverse image* of $b \in \mathit{rng}\ R$ under R is defined by $R.b = b.(R^{-1})$. We saw in section 6.1 that pers can be written in terms of disjoint full relations $\mathcal{S} \times \mathcal{S}$. Difunctionals generalise pers in a natural way, being able to be expressed in terms of disjoint complete relations $\mathcal{S} \times \mathcal{T}$, where sets $\mathcal{S}$ and $\mathcal{T}$ may differ:

Lemma 88: For any difunctional relation R,

(a) Image sets are either identical or disjoint:

$$a.R = a'.R \quad \vee \quad a.R \,\cap\, a'.R = \emptyset$$

$$R.b = R.b' \quad \vee \quad R.b \,\cap\, R.b' = \emptyset.$$

(b) Image sets are in one–to–one correspondence:

$$\{a.R \mid a \in \mathit{dom}\ R\} \;\cong\; \{R.b \mid b \in \mathit{rng}\ R\}.$$

(c) R is completely determined by its image sets:

$$R = \bigcup \{R.b \times a.R \mid a \;R\; b\}.$$

(d) Decomposition in terms of disjoint products is unique. Let $\mathcal{X}, \mathcal{Y}$ be indexed sets of disjoint non–empty sets, with $|\mathcal{X}| = |\mathcal{Y}| = n$. Then,

$$R = \bigcup \{\mathcal{X}_i \times \mathcal{Y}_i \mid 0 \leq i < n\}$$

$$\Rightarrow$$

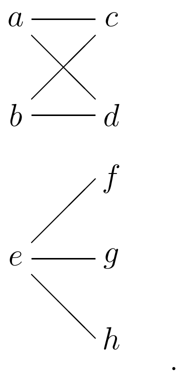
$$\mathcal{X} = \{R.b \mid b \in \mathit{rng}\ R\} \quad \wedge \quad \mathcal{Y} = \{a.R \mid a \in \mathit{dom}\ R\}.$$

For example, the difunctional relation

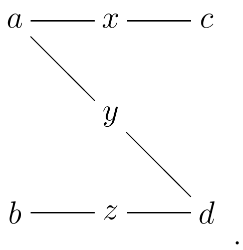
$$\{(a,c),(a,d),(b,c),(b,d),(e,f),(e,g),(e,h)\}$$

can be written in terms of disjoint products as

$$\{a,b\} \times \{c,d\} \;\cup\; \{e\} \times \{f,g,h\}$$



Note that pers and functional relations are difunctional. Moreover, just as for pers, difunctionals are closed under converse, intersection, and par, but not under union or composition [35]. To see that difunctionals are not closed under composition, take $R = \{(a,x),(a,y),(b,z)\}$ and $S = \{(x,c),(y,d),(z,d)\}$; then R and S are both difunctional (R is the converse of a functional relation, S is a functional relation), but their composition $R\,;S \;=\; \{(a,c),(a,d),(b,d)\}$ is not difunctional, being in fact the simplest non–difunctional relation:



In section 6.2.1 we give a necessary and sufficient condition (involving types) for the composition of two difunctionals being difunctional.

6.2.1 Domains

The domain operators have a simple formulation for difunctionals:

Lemma 89: *difun.R* $\;\equiv\;$ $R{<} \;=\; R\,;R^{-1}$.

Lemma 90: *difun.R* $\;\equiv\;$ $R{>} \;=\; R^{-1}\,;R$.

Proof 89

$$R{\scriptstyle <} \;=\; R \,;\, R^{-1}$$

$$\equiv \qquad \{ \text{ lemma 72 } \}$$

$$\underline{R{\scriptstyle <}} \;\supseteq\; R \,;\, R^{-1}$$

$$\equiv \qquad \{ \text{ lemma 72 } \}$$

$$(R \,;\, R^{-1}) \;\cap\; (R/R) \;\cap\; (R/R)^{-1} \;\;\supseteq\;\; R \,;\, R^{-1}$$

$$\equiv \qquad \{ \text{ glbs } \}$$

$$(R \,;\, R^{-1} \;\supseteq\; R \,;\, R^{-1}) \;\wedge\; (R/R \;\supseteq\; R \,;\, R^{-1}) \;\wedge\; ((R/R)^{-1} \;\supseteq\; R \,;\, R^{-1})$$

$$\equiv \qquad \{ \text{ calculus } \}$$

$$R/R \;\supseteq\; R \,;\, R^{-1}$$

$$\equiv \qquad \{ \text{ factors } \}$$

$$R^{-1} \;\supseteq\; R^{-1} \,;\, R \,;\, R^{-1}$$

$$\equiv \qquad \{ \text{ def } \}$$

$$\mathit{difun}.R$$

These results are used to prove that difunctionals are closed under composition precisely when the composition of the smallest intermediate domains is transitive: (Sheeran [61] gives an alternative proof using the operators ⊣ and ⊢.)

Lemma 91: $\mathit{difun}.R \;\wedge\; \mathit{difun}.S \;\;\Rightarrow$

$$\mathit{difun}.(R \,;\, S) \;\;\equiv\;\; \mathit{transitive}.(R{\scriptstyle >} \,;\, S{\scriptstyle <}).$$

Proof "⇒"

$$(\underline{R{\scriptstyle >}} \,;\, \underline{S{\scriptstyle <}}) \,;\, (\underline{R{\scriptstyle >}} \,;\, \underline{S{\scriptstyle <}})$$

$$= \qquad \{ \; \mathit{difun}.R \;\wedge\; \mathit{difun}.S \; \}$$

$$R^{-1} \,;\, R \,;\, S \,;\, \underline{S^{-1} \,;\, R^{-1}} \,;\, R \,;\, S \,;\, S^{-1}$$

$=$ $\quad\{\ \text{converse}\ \}$

$R^{-1} ; \underline{R ; S ; (R ; S)^{-1} ; R ; S} ; S^{-1}$

$\subseteq$ $\quad\{\ \mathit{difun}.(R ; S)\ \}$

$\underline{R^{-1} ; R} ; \underline{S ; S^{-1}}$

$=$ $\quad\{\ \mathit{difun}.R \ \wedge\ \mathit{difun}.S\ \}$

$(R_{>} ; S_{<})$

Proof "$\Leftarrow$"

$\mathit{difun}.(R ; S)$

$\equiv$ $\quad\{\ \text{def}\ \}$

$R ; \underline{S ; S^{-1}} ; \underline{R^{-1} ; R} ; S \quad\subseteq\quad R ; S$

$\equiv$ $\quad\{\ \mathit{difun}.R \ \wedge\ \mathit{difun}.S\ \}$

$\underline{R} ; S_{<} ; R_{>} ; \underline{S} \quad\subseteq\quad \underline{R} ; \underline{S}$

$\equiv$ $\quad\{\ \text{domains}\ \}$

$R ; \underline{R_{>} ; S_{<} ; R_{>} ; S_{<}} ; S \quad\subseteq\quad R ; \underline{R_{>} ; S_{<}} ; S$

$\Leftarrow$ $\quad\{\ \text{monotonicity}\ \}$

$R_{>} ; S_{<} ; R_{>} ; S_{<} \quad\subseteq\quad R_{>} ; S_{<}$

$\equiv$ $\quad\{\ \text{def}\ \}$

$\mathit{transitive}.(R_{>} ; S_{<})$

In practice, one of two sufficient conditions can often be used to show that the composition of two difunctionals is difunctional:

Lemma 92: $\mathit{difun}.R \ \wedge\ \mathit{difun}.S \ \Rightarrow$

$\mathit{difun}.(R ; S) \ \Leftarrow\ (R_{>} \lhd S_{<} \ \vee\ S_{<} \lhd R_{>})$.

Proof

$$\underline{\mathit{difun}.(R \ ; \ S)}$$

$$\equiv \qquad \{ \text{ lemma 91 } \}$$

$$\underline{\mathit{transitive}.(R{\scriptstyle >} \ ; \ S{\scriptstyle <})}$$

$$\equiv \qquad \{ \text{ def } \}$$

$$R{\scriptstyle >} \ ; \ S{\scriptstyle <} \ ; \ R{\scriptstyle >} \ ; \ S{\scriptstyle <} \quad \subseteq \quad R{\scriptstyle >} \ ; \ S{\scriptstyle <}$$

$$\equiv \qquad \{ \ R{\scriptstyle >} \lhd S{\scriptstyle <} \ \vee \ S{\scriptstyle <} \lhd R{\scriptstyle >} \ \}$$

$$\mathit{true}$$

6.2.2 Difunctional ≡ Factorisable

It is well known that every relation R can be expressed in the form $f^{-1} \ ; \ g$, for f and g some functional relations. For example, take $f = id_R \,;\, \pi_1$ and $g = id_R \,;\, \pi_2$, where id_R is the identity relation on the set of pairs comprising R. One can verify now that f and g are functional relations, and that $R = f^{-1} \ ; \ g$.

Turning things around, we find that the composition of a functional relation and the converse of a functional relation is always difunctional:

Lemma 93: $\mathit{fn}.f \ \wedge \ \mathit{fn}.g \quad \Rightarrow \quad \mathit{difun}.(f \,;\, g^{-1})$.

Proof

$$(f \,;\, g^{-1}) \,;\, \underline{(f \,;\, g^{-1})^{-1}} \,;\, (f \,;\, g^{-1})$$

$$= \qquad \{ \text{ converse } \}$$

$$f \,;\, \underline{g^{-1} \,;\, g} \,;\, \underline{f^{-1} \,;\, f} \,;\, g^{-1}$$

$$\subseteq \qquad \{ \ \mathit{fn}.f \ \wedge \ \mathit{fn}.g \ \}$$

$$f \,;\, id \,;\, id \,;\, g^{-1}$$

$$= \qquad \{ \text{ identity } \}$$

$$f \,;\, g^{-1}$$

Moreover, every difunctional can be factorised in this way:

Lemma 94: $\mathit{difun}.R \;\Rightarrow\; \exists f, g.\ (\mathit{fn}.f \;\wedge\; \mathit{fn}.g \;\wedge\; R = f \;;\, g^{-1})$.

See [36] for a proof. Combining the two results above: "Precisely the difunctional relations can be expressed as the composition of a functional relation and the converse of a functional relation." All the programs that we derive in chapter 7 are difunctional programs specified as such a composition.

6.2.3 Difunctional $\equiv$ Invertible

An *inverse* of a function $f : \mathcal{A} \to \mathcal{B}$ is a function $g : \mathcal{B} \to \mathcal{A}$ satisfying $g \circ f = id_{\mathcal{A}}$ and $f \circ g = id_{\mathcal{B}}$. We can make such a definition in relational calculus:

Definition 95: An *inverse* of a relation $R \in A \leftrightarrow B$ is a relation $S \in B \leftrightarrow A$ for which $R \;;\, S = A$ and $S \;;\, R = B$.

Not every relation $R \in A \leftrightarrow B$ is invertible, but those that are have R^{-1} as a unique inverse: (See lemma D22 in [4] for the proof.)

Lemma 96: $S \in B \leftrightarrow A$ is an inverse of $R \in A \leftrightarrow B \quad \Rightarrow \quad S = R^{-1}$.

Which relations can be inverted? Precisely the difunctionals:

Lemma 97: $R \in A \leftrightarrow B$ is invertible $\;\Rightarrow\;$ $\mathit{difun}.R$.

Lemma 98: $\mathit{difun}.R \;\Rightarrow\; R \in R{<} \leftrightarrow R{>}$ is invertible.

Definition 95 assumes that A and B are pers. Is it possible to separate the invertibility result from a specific choice of types? Perhaps one can define an inverse of a relation R as an S satisfying $R \;;\, S \;;\, R = R$ and $S \;;\, R \;;\, S = S$? Using this definition however, R^{-1} is no longer the unique inverse of R. Take $R = \{(a,a),(a,b)\}$ and $S = \{(a,a)\}$, then $R \;;\, S \;;\, R = R$ and $S \;;\, R \;;\, S = S$, but $S \neq R^{-1}$.

6.2.4 Difunctional $\equiv$ Per–functional

The notion of a functional relation depends upon our choice of types. In this section we give a brief argument that when pers are types, difunctionals relations are the functional relations. A more formal argument is given in [35].

In the standard approach to relational calculus, equivalent to working with partial identity relations (pirs) $A \subseteq id$ as types, a relation R is functional if $R^{-1}\,;\,R \subseteq id$. Working with pirs, the right domain of a relation R is defined by $R{>} = (R^{-1}\,;\,R) \cap id$ [4]. We observe now that functionality can be expressed in terms of the domain operator: R is functional $\equiv$ $R^{-1}\,;\,R = R{>}$.

Working with pers, the right domain of R is given by $R{>} = (R^{-1}\,;\,R) \cap \acute{R}^{-1}$. Taking the lead from the observation for pirs, we make the following definition:

Definition 99: R is *per–functional* $\;\widehat{=}\;$ $R^{-1}\,;\,R = R{>}$.

Applying lemma 90, we conclude that the per–functional relations are precisely the difunctional relations: (It is interesting to note that, because *difun*.R $\equiv$ *difun*.R^{-1}, the converse of a per–functional relation is also per–functional!)

Lemma 100: R is per–functional $\;\equiv\;$ R is difunctional.

6.3 Counting pers and difunctionals

We end this chapter with a bit of fun. The space of relations on a universe $\mathcal{A}$ is given by the powerset $P(\mathcal{A} \times \mathcal{A})$. If $\mathcal{A}$ has n elements, then the space of relations has $2^{n \times n}$ elements. In this section we define Miranda functions that compute how many of these relations are equivalence relations, pers, and difunctional relations. Writing such programs helps in becoming familiar with the three notions.

We begin by defining a function `eqvs`: (The function `rep` used below takes a number n and a value x, and gives a list of n copies of x.)

```
eqvs   0    = [0]
eqvs (n+1) = concat [i+1 : rep i i | i <- eqvs n]
```

The length of the list `eqvs n` is the number of equivalence relations on a set with `n` elements. Each element of `eqvs n` is the number of classes in one such equivalence relation; for example, `eqvs 3 = [3,2,2,2,1]` tells us that there are 5 equivalence relations on a set with 3 elements, one having 3 classes, three having 2 classes, and one having 1 class. Let us explain how `eqvs` works: if an equivalence relation has i classes then adding a new point to the universe gives i equivalence relations with i classes (one for each class the new point can be placed in) and one equivalence relation with $i+1$ classes (in which the new point forms a completely new class).

The `pers` function is defined just as for `eqvs`, except that we have an extra relation with i classes, in which the new point is not added to any class:

```
pers   0    = [0]
pers (n+1) = concat [i+1 : rep (i+1) i | i <- pers n]
```

Now for `difuns`. If a difunctional comprises i products, then adding a new point to the universe gives $3i+1$ difunctionals with i products: one for adding the new point to none of the products, and one each for adding the new point to the left side, right side, and both sides of each product. In addition, we get one difunctional with $i+1$ products, in which the new point gives rise to a completely new product.

```
difuns   0    = [0]
difuns (n+1) = concat [i+1 : rep ((3*i)+1) i | i <- difuns n]
```

Here are the number of eqvs, pers and difuns on sets with $0 \ldots 9$ points:

```
[#(eqvs n) | n <- [0..9]]
   = [1,1,2,5,15,52,203,877,4140,21147]


[#(pers n) | n <- [0..9]]
```

```
   = [1,2,5,15,52,203,877,4140,21147,115975]


[#(difuns n) | n <- [0..9]]
   = [1,2,7,35,214,1523,12349,112052,1120849,12219767]
```

Note that the pers list is the eqvs list with the first element removed: `pers n = eqvs (n+1)`. This is explained by noting that a per A can be seen as an equivalence relation with an extra equivalence class comprising points outside *dom* A.

Chapter 7

Calculating Programs

In this chapter we put everything we have developed into practice. We use Ruby to derive some programs specified as the composition of a functional relation and the converse of a functional relation, and run the resulting implementations on the Ruby interpreter. Preconditions of the 'induction' laws used within the derivations are verified using algebraic properties of the domain operators $<$ and $>$.

7.1 More Ruby laws

Laws for some primitive relations and combining forms were given in section 3.4. In this section we give some laws about arithmetic primitives and generic combining forms. Proofs of these laws can be found in the earlier Ruby articles. Thinking in terms of pictures is very helpful in understanding the laws for the generic combining forms; see for example the pictures that follow lemma 113.

The first four laws are about the primitives $+$ and $*$. The first two are dummy–free versions of the standard associativity and distributivity rules; the second two are relational properties that are much used when deriving programs:

Lemma 101: (associativity)

$$\mathrm{fst}\ + \ ;\ + \ =\ \mathit{lsh}\ ;\ \mathrm{snd}\ + \ ;\ +.$$

Lemma 102: (distribution)

$$+ \ ; \ {*n} \ = \ [{*n}, {*n}] \ ; \ +.$$

Lemma 103: $\mathrm{fst}\ {*n} \ ; \ + \ ; \ {*n}^{-1} \ = \ \mathrm{snd}\ {*n}^{-1} \ ; \ +.$

Lemma 104: $+ \ ; \ +^{-1} \ = \ \mathrm{snd}\ +^{-1} \ ; \ \mathit{rsh} \ ; \ \mathrm{fst}\ +.$

The next two laws express that the argument to map can be pushed inside left reductions and columns. (Of course, similar laws hold for right reductions and rows, but we don't have need for them in this thesis.) These laws are used in the left–to–right direction, to bring together the argument of a map and the argument of a left reduction (or column) so that they can be manipulated together:

Lemma 105: (map through left–reduce)

$$\mathrm{snd}\ (\mathrm{map}_n\ R) \ ; \ \mathrm{rdl}_n\ S \ \ = \ \ \mathrm{rdl}_n\ (\mathrm{snd}\ R \ ; \ S).$$

Lemma 106: (map through column)

$$\mathrm{fst}\ (\mathrm{map}_n\ R) ; \mathrm{col}_n\ S ; \mathrm{snd}\ (\mathrm{map}_n\ T) \ \ = \ \ \mathrm{col}_n\ (\mathrm{fst}\ R \ ; \ S \ ; \ \mathrm{snd}\ T).$$

The next two laws express that a transformation of a certain form (essentially a 'shunting transformation') can be rippled through a left reduction or a column of components: (Again, similar laws hold for right reductions and rows.)

Lemma 107: (left–reduce induction)

$$\begin{array}{ll} & \mathrm{fst}\ R \ ; \ S \ \ = \ \ T \ ; \ R \\ \Rightarrow & \\ & \mathrm{fst}\ R \ ; \ \mathrm{rdl}_n\ S \ \ = \ \ \mathrm{rdl}_n\ T \ ; \ R. \end{array}$$

Lemma 108: (column induction)

$$\begin{array}{ll} & \text{snd } R \; ; \; S \;\; = \;\; T \; ; \; \text{fst } R \\ \Rightarrow & \\ & \text{snd } R \; ; \; \text{col}_n \; S \;\; = \;\; \text{col}_n \; T \; ; \; \text{fst } R. \end{array}$$

Column induction is commonly used to push a type constraint on the domain of a column through to the argument program. Column induction gives the 'rippling' law below by taking $R = A$ (where A is a per) and $T \; = \; \text{snd } A \; ; \; S \; ; \; \text{fst } A$: (Note that we have used $\text{col}_n \; (R \; ; \; \text{fst } A) \; \vdash \; \text{fst } A$, which follows from the type inference rule for column, to eliminate the trailing fst A from the column rippling rule.)

Lemma 109: (column rippling)

$$\begin{array}{ll} & \text{snd } A \; ; \; S \; \vdash \; \text{fst } A \\ \Rightarrow & \\ & \text{snd } A \; ; \; \text{col}_n \; S \;\; = \;\; \text{col}_n \; (\text{snd } A \; ; \; S \; ; \; \text{fst } A). \end{array}$$

The precondition above asserts that fst A is a right domain of snd A ; S. For simple programs, one can verify such a precondition by applying shunting–like laws. See for example the use of column rippling in the derivation of the binary addition program in section 7.4. When deriving more complicated programs however, such as the base–conversion program in section 7.6, we take a different approach: the precondition is first expressed in the form $(\text{snd } A \; ; \; S)^{>} \; \lhd \; \text{fst } A$ using lemma 78, which is then verified using the algebraic properties of the $^{>}$ operator.

The term *Horner's rule* usually refers to

$$a_n.x^n + \ldots a_2.x^2 + a_1.x^1 + a_0.x^0 \;\; = \;\; (((a_n.x + \ldots)x + a_2)x + a_1)x + a_0,$$

which shows how to evaluate polynomials more efficiently. Bird and Meertens have used a variant of Horner's rule to great effect in deriving functional programs [8, 49]. The two laws below are variants of Horner's rule that are used in Ruby:

Lemma 110: (Horner's rule for left–reduce)

$$[R, R] \ ; \ S \ = \ S \ ; \ R$$

$\Rightarrow$

$$\mathrm{tri}_{n+1} \ R \ ; \ {apl_n}^{-1} \ ; \ \mathrm{rdl}_n \ S \ = \ {apl_n}^{-1} \ ; \ \mathrm{rdl}_n \ (\mathrm{fst} \ R \ ; \ S).$$

Lemma 111: (Horner's rule for column)

$$\mathrm{fst} \ R \ ; \ S \ ; \ [T, T] \ = \ \mathrm{snd} \ T \ ; \ S$$

$\Rightarrow$

$$\mathrm{fst} \ (\mathrm{tri}_n \ R) \ ; \ \mathrm{col}_n \ S \ ; \ [T^n, \mathrm{tri}_n \ T] \ = \ \mathrm{col}_n \ (S \ ; \ \mathrm{fst} \ T).$$

Our final two laws combine a reduction and the converse of a reduction to give a column; the application of one of these rules is a key step in all our derivations:

Lemma 112: (sliding reductions)

$$R \ ; \ S^{-1} \ = \ \mathrm{snd} \ S^{-1} \ ; \ rsh \ ; \ \mathrm{fst} \ R$$

$\Rightarrow$

$$\mathrm{rdr}_n \ R \ ; \ (\mathrm{rdl}_n \ S)^{-1} \ = \ \mathrm{col}_n \ (R \ ; \ S^{-1}).$$

Lemma 113: (zipping reductions)

$$\mathrm{fst} \ R \ ; \ S \ ; \ T^{-1} \ = \ lsh \ ; \ \mathrm{snd} \ W \ ; \ rsh \ ; \ \mathrm{fst} \ S$$

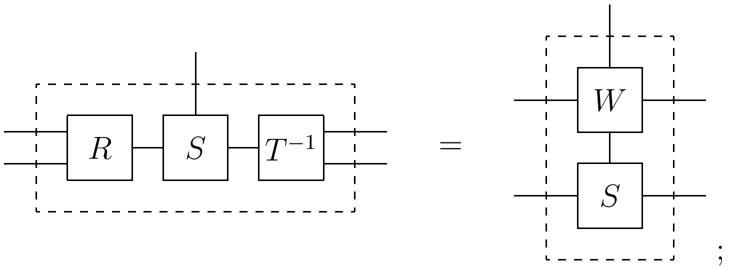
$\Rightarrow$

$$\mathrm{fst} \ (\mathrm{rdl}_n \ R) \ ; \ S \ ; \ (\mathrm{rdl}_n \ T)^{-1}$$

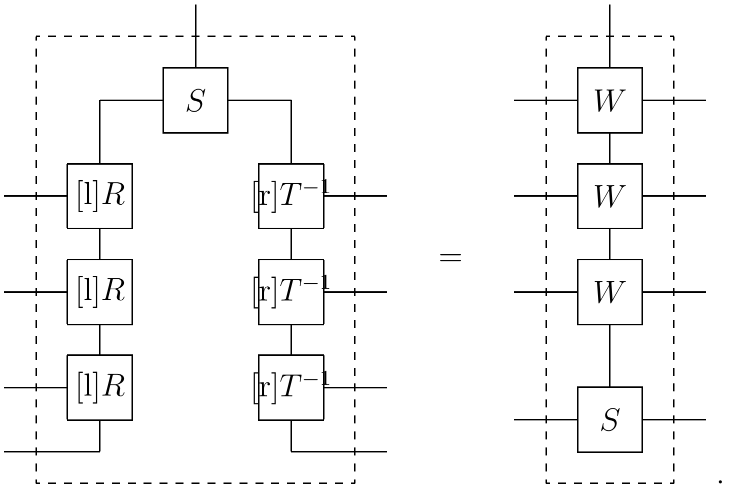
$=$

$$lsh \ ; \ \mathrm{snd} \ (\mathrm{col}_n \ W) \ ; \ rsh \ ; \ \mathrm{fst} \ S.$$

Here is a picture of the precondition to lemma 113:



here is a picture of an instance of the consequent:



7.2 Representing numbers

All the programs that we derive in this chapter work with numbers. In this section we give some basic definitions involving numbers, and some laws.

Definition 114: $sum_{n+1} \;\; \widehat{=} \;\; {apr_n}^{-1} \; ; \; \mathrm{rdr}_n \, +$.

Definition 115: $\mathrm{lad}_n \, x \;\; \widehat{=} \;\; \mathrm{tri}_n \, (*x) \; ; \; sum_n$.

The program sum_n relates an n–tuple ($n > 0$) of numbers to their sum; $\mathrm{lad}_n \, x$ relates an n–tuple of numbers to their weighted sum, where the ith component of the tuple (reading from right to left and starting from 0) is given weight x^i. For example, $(1,2,3,4) \; sum_4 \; 10$ and $(a,b,c,d) \;\; \mathrm{lad}_4 \, 2 \;\; 8a+4b+2c+d$.

Here are some useful properties: (the first three are consequences of the associativity of addition, the last that $*$ distributes over $+$.)

Lemma 116: $sum_{n+1} \; = \; {apl_n}^{-1} \; ; \; \mathrm{rdl}_n \, +$.

Lemma 117: $\mathrm{fst} \; sum_n \; ; \; + \; = \; \mathrm{rdr}_n \, +$.

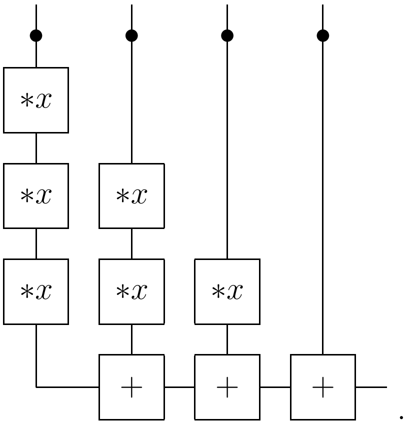
Lemma 118: snd sum_n ; $+$ $=$ $\mathrm{rdl}_n +$.

Lemma 119: $[\mathrm{lad}_n\, x, \mathrm{lad}_n\, x]$; $+$ $=$ zip_n ; $\mathrm{map}_n +$; $\mathrm{lad}_n\, x$.

An n–digit number in base x will be represented as an n–tuple of integers in the range 0 to $x-1$. The leftmost value in the tuple is assumed to be most significant. For example, the tuple $(2,0,2,1,0)$ is the representation of the decimal number 183 as a 5–digit number in base 3; that is, $2.3^4 + 0.3^3 + 2.3^2 + 1.3^1 + 0.3^0 = 183$. The same number represented as an 8–digit number in base 2 is $(1,0,1,1,0,1,1,1)$. The program $eval_n\, x$ converts an n–digit number in base x to a natural number: (For $n > 0$, $|\,n\,|$ abbreviates the identity relation $[\![0, \ldots, n-1]\!]$.)

Definition 120: $eval_n\, x \;\widehat{=}\; \mathrm{map}_n\, |\,x\,|$; $\mathrm{lad}_n\, x$.

The following picture is helpful in reading this definition: (● represents $|\,x\,|$)



Here are some properties of $eval_n\, x$:

Lemma 121: $eval_n\, x$ is a bijection.

Lemma 122: $(eval_n\, x)\!>\; = \;|\,x^n\,|$.

Lemma 123: $(eval_n\, x)\!<\; = \;\mathrm{map}_n\, |\,x\,|$.

Since multiplication distributes over addition (lemma 102), Horner's rule for left–reduce (lemma 110) can be applied to eliminate many of the multiplications:

Lemma 124: $\mathit{eval}_n\, x \;\; = \;\; \mathrm{map}_n\, \lfloor x \rfloor \;\; ; \;\; \mathit{apl}_{n-1}{}^{-1} \;\; ; \;\; \mathrm{rdl}_{n-1}\, (\mathrm{fst} * x \;\; ; \;\; +).$

An alternative formulation with a hidden 0 is sometimes useful:

Lemma 125: $\mathit{eval}_n\, x \;\; = \;\; \mathrm{map}_n\, \lfloor x \rfloor \;\; ; \;\; \pi_2{}^{-1} \;\; ; \;\; \mathrm{fst}\; [\![0]\!] \;\; ; \;\; \mathrm{rdl}_n\, (\mathrm{fst} * x \;\; ; \;\; +).$

We write b for the type of binary digits:

Definition 126: $b \;\; \widehat{=} \;\; [\![0, 1]\!].$

The program bin_n converts an n–bit binary number to a natural number:

Definition 127: $\mathit{bin}_n \;\; \widehat{=} \;\; \mathit{eval}_n\, 2.$

A *carry–save* number is like a binary number in that digits have weight 2^i, but different in that acceptable digits are $\{0, 1, 2\}$, and that a digit is represented as a pair of bits whose sum is that digit. For example, $((1, 0), (1, 1), (0, 1))$ is a carry–save representation of 9, because $(1 + 0).2^2 + (1 + 1).2^1 + (0 + 1).2^0 = 9$. A number can have many carry–save representations; e.g. $((1, 1), (0, 0), (1, 0))$ also represents 9. The program csv_n converts an n–wide carry–save number to a natural number:

Definition 128: $\mathit{csv}_n \;\; \widehat{=} \;\; \mathrm{map}_n\, ([b, b] \;\; ; \;\; +) \;\; ; \;\; \mathrm{lad}_n\, 2.$

Here are some properties of csv_n:

Lemma 129: csv_n is functional.

Lemma 130: $\mathit{csv}_n{}^{>} \;\; = \;\; \lfloor 2^{n+1} - 1 \rfloor.$

Lemma 131: $\mathit{csv}_n{}^{<} \;\; \lhd\!| \;\; \mathrm{map}_n\, [b, b].$

Results 130 and 131 together express that every carry–save number represents one natural number and a natural number can have many carry–save representations. (Recall that $A \lhd\!| \; B$ says that per A is 'an equivalence relation on' per B.)

Binary and carry–save numbers are related by the following laws:

Lemma 132: $[\mathit{bin}_n, \mathit{bin}_n] \; ; + \; = \; \mathit{zip}_n \; ; \; \mathit{csv}_n$.

Lemma 133: $[\mathit{bin}_n, \mathit{csv}_n] \; ; + \; = \; \mathit{zip}_n \; ; \mathrm{map}_n \; ([b, [b, b] \; ; \; +] \; ; \; +) \; ; \mathrm{lad}_n \, 2$.

We prove lemma 132 below; lemma 133 follows by a similar argument:

$$
\begin{array}{ll}
 & [\underline{\mathit{bin}_n}, \underline{\mathit{bin}_n}] \; ; \; + \\
= & \qquad \{ \text{ def } 127 \text{ } \} \\
 & \underline{[\mathrm{map}_n \, b \; ; \; \mathrm{lad}_n \, 2, \mathrm{map}_n \, b \; ; \; \mathrm{lad}_n \, 2]} \; ; \; + \\
= & \qquad \{ \text{ par } \} \\
 & [\mathrm{map}_n \, b, \mathrm{map}_n \, b] \; ; \; \underline{[\mathrm{lad}_n \, 2, \mathrm{lad}_n \, 2] \; ; \; +} \\
= & \qquad \{ \text{ ladders (119) } \} \\
 & \underline{[\mathrm{map}_n \, b, \mathrm{map}_n \, b] \; ; \; \mathit{zip}_n} \; ; \; \mathrm{map}_n \, + \; ; \; \mathrm{lad}_n \, 2 \\
= & \qquad \{ \text{ shunting (30) } \} \\
 & \mathit{zip}_n \; ; \; \underline{\mathrm{map}_n \, [b, b] \; ; \; \mathrm{map}_n \, + \; ; \; \mathrm{lad}_n \, 2} \\
= & \qquad \{ \text{ def } 128 \text{ } \} \\
 & \mathit{zip}_n \; ; \; \mathit{csv}_n
\end{array}
$$

7.3 Representation changers

We use the term *representation changer* for a program that converts an abstract value from one concrete representation to another. A simple example of a representation changer is a base-conversion function *conv* that converts an n–digit number in base x to an m–digit number in base y. In this case, abstract values are natural numbers, and concrete values are numbers in base x and base y. Representation changers have a natural specification as a composition $R \mathbin{;} S^{-1}$, where R is a relation that converts from one concrete type to the abstract type, and S converts from the other concrete type to the abstract type. For example, the function *conv* can be specified by the requirement that $conv = eval_n\, x \ ; \ (eval_m\, y)^{-1}$.

In many cases the components R and S of a representation changer $R \ ; \ S^{-1}$ will be functional relations. In section 6.2.2 we observed that precisely the difunctional relations can be expressed as the composition of a functional relation and the converse of a functional relation. In making the following definitions, Jones and Sheeran [43, 42] generalise a little, allowing the components to be difunctional relations, but still requiring that the composition be difunctional:

> **Definition 134:** A *representation changer* is a difunctional program specified as a composition $R \mathbin{;} S^{-1}$, where R and S are also difunctional relations. We refer to R and S as *representation relations*.

> **Definition 135:** The *concrete* domain of a representation relation R is the per $R{<} = R \mathbin{;} R^{-1}$, and the *abstract* domain is the per $R{>} = R^{-1} \mathbin{;} R$.

For a representation relation R, read $a\,R\,b$ as "concrete value a represents abstract value b." That a representation relation need not be functional means that a concrete value can represent more than one abstract value; e.g. one might imagine a representation relation *sign* with concrete domain $[\![-1, 0, 1]\!]$ and abstract domain *int*, in which -1 represents the negative integers, 0 represents 0, and 1 represents the positive integers. That a representation relation need not be the converse of a functional relation means that an abstract value can have more than one concrete representation; e.g. under $+$ both $(1, 1)$ and $(2, 0)$ represent 2.

Commonly for representation changers it is easy to define R and S, but not obvious how to implement $R \ ; \ S^{-1}$. Refinement of a program specified in this way proceeds by sliding parts of R and S through one another (using laws 112 and 113 for example), aiming towards a new program with components that are representation changers with smaller abstract types. In this sense, 'thinking about types' guides our calculations. The process is repeated until the remaining representation changers can be implemented directly using standard primitives.

A representation changer that is much used in Ruby is a *half-adder*, which gives the binary carry and sum of a pair of bits. A half–adder is a representation changer which converts a natural number $n \in \{0,1,2\}$ represented as a pair of bits (x,y) with $x+y=n$ to a pair of bits (x',y') with $2x'+y'=n$:

Definition 136: $\mathit{HA} \;\;\widehat{=}\;\; [b,b] \;;\; + \;;\; ([b \;;\; *2, b] \;;\; +)^{-1}.$

The half–adder can be implemented as follows:

Lemma 137: $\mathit{HA} \;\;=\;\; [b,b] \;;\; + \;;\; \mathit{fork} \;;\; [\mathit{div}\,2 \;;\; b, \mathit{mod}\,2 \;;\; b].$

This result follows quickly from the following:

Lemma 138: $x > 0 \quad\Rightarrow\quad ([*x, \mid x \mid] \;;\; +)^{-1} \;=\; \mathit{fork} \;;\; [\mathit{div}\; x, \mathit{mod}\; x].$

We conclude with another useful representation changer; a *full–adder* gives the binary carry and sum of three bits, arranged here as a single bit and a pair of bits:

Definition 139: $\mathit{FA} \;\;\widehat{=}\;\; [b, [b,b] \;;\; +] \;;\; + \;;\; ([b \;;\; *2, b] \;;\; +)^{-1}.$

Here are two properties of *FA* that are used later on:

Lemma 140: $\mathit{lsh} \;;\; \mathit{FA} \;\;=\;\; [[b,b] \;;\; +, b] \;;\; + \;;\; ([b \;;\; *2, b] \;;\; +)^{-1}.$

Lemma 141: $\mathit{FA} \;;\; \mathrm{fst}\; *2 \;;\; + \;\;=\;\; [b, [b,b] \;;\; +] \;;\; +.$

Proof 140

$$\begin{array}{ll}
 & \mathit{lsh} \ ; \ \underline{\mathit{FA}} \\
= & \quad \{ \text{ def } 139 \ \} \\
 & \underline{\mathit{lsh}} \ ; \ [\underline{b}, \underline{[b,b]} \ ; \ +] \ ; \ + \ ; \ ([b \ ; \ *2, b] \ ; \ +)^{-1} \\
= & \quad \{ \text{ shunting } \} \\
 & [[b,b] \ ; \ b] \ ; \ \underline{\mathit{lsh} \ ; \ \mathrm{snd} \ + \ ; \ +} \ ; \ ([b \ ; \ *2, b] \ ; \ +)^{-1} \\
= & \quad \{ \text{ associativity } (101) \ \} \\
 & [[b,b] \ ; \ +, b] \ ; \ + \ ; \ ([b \ ; \ *2, b] \ ; \ +)^{-1}
\end{array}$$

Proof 141

$$\begin{array}{ll}
 & \underline{\mathit{FA}} \ ; \ \mathrm{fst} \ *2 \ ; \ + \\
= & \quad \{ \text{ def } 139 \ \} \\
 & [b, [b,b] \ ; \ +] \ ; \ + \ ; \ +^{-1} \ ; \ [\underline{*2^{-1} \ ; \ b \ ; \ *2}, b] \ ; \ + \\
= & \quad \{ \text{ expanding } \} \\
 & [b, [b,b] \ ; \ +] \ ; \ + \ ; \ \underline{+^{-1} \ ; \ [\llbracket 0,1,2 \rrbracket, b] \ ; \ +} \\
= & \quad \{ \text{ expanding } \} \\
 & [b, [b,b] \ ; \ +] \ ; \ + \ ; \ \llbracket 0,1,2,3 \rrbracket \\
= & \quad \{ \text{ domains } \} \\
 & [b, [b,b] \ ; \ +] \ ; \ +
\end{array}$$

As is well known, a full–adder can be implemented using two half–adders:

Lemma 142: $\mathit{FA} \ = \ \mathit{HA} \leftrightarrow \mathit{HA} \ ; \ \mathrm{fst} \ +.$

7.4 Binary addition

In this section we derive our first program using Ruby. The program takes an n–bit binary number ($n > 0$) and a single bit, and gives a carry bit and an n–bit sum [34].

Given below is our specification for the addition program: R converts the binary number to an integer and adds the bit, S^{-1} separates off the carry bit, and converts the remaining integer back to binary.

Definition 143: $\mathit{add1}_n \;\widehat{=}\; R \;;\; S^{-1}$, where

$$\begin{aligned} R &= [\mathit{bin}_n, b] \;;\; +, \\ S &= [b \;;\; (*2)^n, \mathit{bin}_n] \;;\; +. \end{aligned}$$

Since R is functional and S is bijective, $\mathit{add1}_n = R \;;\; S^{-1}$ is functional, and hence implementable in the sense of definition 45. In fact, $\mathit{add1}_n$ is already an implementation: take any one of the range wires in its network as the output wire, and the remaining external wires as input wires. The program is not however an implementation for the direction that we want to use it, namely from domain to range. Moreover, one might expect that internally the binary addition program should manipulate only 0's and 1's, whereas natural numbers are used with $\mathit{add1}_n$ as defined above. We shall calculate an implementation of $\mathit{add1}_n$ that is executable from domain to range and manipulates bits rather than natural numbers.

We begin with some simple rearranging steps:

$$\begin{aligned}
& \mathit{add1}_n \\
= & \qquad \{ \text{ def } 143 \text{ } \} \\
& [\underline{\mathit{bin}_n}, b] \;;\; + \;;\; ([b \;;\; (*2)^n, \underline{\mathit{bin}_n}] \;;\; +)^{-1} \\
= & \qquad \{ \text{ def } 127 \text{ } \} \\
& [\mathrm{map}_n\ b \;;\; \mathrm{tri}_n\ *2 \;;\; \underline{\mathit{sum}_n}, b] \;;\; \underline{+} \;;\; ([b \;;\; (*2)^n, \mathrm{map}_n\ b \;;\; \mathrm{tri}_n\ *2 \;;\; \underline{\mathit{sum}_n}] \;;\; \underline{+})^{-1} \\
= & \qquad \{ \text{ sums } (117{,}118) \text{ } \} \\
& [\mathrm{map}_n\ b \;;\; \mathrm{tri}_n\ *2, b] \;;\; \mathrm{rdr}_n\ + \;;\; \underline{([b \;;\; (*2)^n, \mathrm{map}_n\ b \;;\; \mathrm{tri}_n\ *2] \;;\; \mathrm{rdl}_n\ +)^{-1}} \\
= & \qquad \{ \text{ converse } \} \\
& [\mathrm{map}_n\ b \;;\; \mathrm{tri}_n\ *2, b] \;;\; \underline{\mathrm{rdr}_n\ + \;;\; (\mathrm{rdl}_n\ +)^{-1}} \;;\; [(*2)^{-n} \;;\; b, \mathrm{tri}_n\ *2^{-1} \;;\; \mathrm{map}_n\ b]
\end{aligned}$$

The right–reduction and converse left–reduction can now be combined using lemma 112, with $R, S = +$. Under these assignments, the precondition is given by lemma 104, a property of addition much used in Ruby. We continue:

$$
\begin{array}{ll}
 & [\mathrm{map}_n\ b\ ;\ \mathrm{tri}_n\ {*2}, b]\ ;\ \underline{\mathrm{rdr}_n\ +\ ;\ (\mathrm{rdl}_n\ +)^{-1}}\ ;\ [({*2})^{-n}\ ;\ b, \mathrm{tri}_n\ {*2^{-1}}\ ;\ \mathrm{map}_n\ b] \\
= & \qquad \{\ \text{sliding reductions (112)}\ \} \\
 & [\mathrm{map}_n\ b\ ;\ \underline{\mathrm{tri}_n\ {*2}}, b]\ ;\ \underline{\mathrm{col}_n\ (+\ ;\ +^{-1})}\ ;\ [({*2})^{-n}\ ;\ b, \underline{\mathrm{tri}_n\ {*2^{-1}}}\ ;\ \mathrm{map}_n\ b]
\end{array}
$$

Now the triangles can be pushed inside the column using a variant of Horner's rule, lemma 111. This law is used here with $R = {*\,2}$, $S = (+\ ;\ +^{-1})$, and $T\ =\ {*2^{-1}}$. We verify the precondition of the law under these assignments as follows:

$$
\begin{array}{ll}
 & \mathrm{fst}\ R\ ;\ S\ ;\ [T, T]\ =\ \mathrm{snd}\ T\ ;\ S \\
\equiv & \qquad \{\ \text{assignments}\ \} \\
 & \mathrm{fst}\ {*2}\ ;\ +\ ;\ \underline{+^{-1}\ ;\ [{*2^{-1}}, {*2^{-1}}]}\ =\ \mathrm{snd}\ {*2^{-1}}\ ;\ +\ ;\ +^{-1} \\
\equiv & \qquad \{\ \text{distribution (102)}\ \} \\
 & \mathrm{fst}\ {*2}\ ;\ +\ ;\ {*2^{-1}}\ ;\ +^{-1}\ =\ \mathrm{snd}\ {*2^{-1}}\ ;\ +\ ;\ +^{-1} \\
\Leftarrow & \qquad \{\ \text{Liebniz}\ \} \\
 & \mathrm{fst}\ {*2}\ ;\ +\ ;\ {*2^{-1}}\ =\ \mathrm{snd}\ {*2^{-1}}\ ;\ + \\
\equiv & \qquad \{\ \text{lemma 103}\ \} \\
 & \mathit{true}
\end{array}
$$

Continuing with the *add1*$_n$ calculation:

$$
\begin{array}{ll}
 & [\mathrm{map}_n\ b\ ;\ \underline{\mathrm{tri}_n\ {*2}}, b]\ ;\ \underline{\mathrm{col}_n\ (+\ ;\ +^{-1})}\ ;\ [({*2})^{-n}\ ;\ b, \underline{\mathrm{tri}_n\ {*2^{-1}}}\ ;\ \mathrm{map}_n\ b] \\
= & \qquad \{\ \text{Horner's rule (111)}\ \} \\
 & [\underline{\mathrm{map}_n\ b}, b]\ ;\ \underline{\mathrm{col}_n\ (+\ ;\ +^{-1}\ ;\ \mathrm{fst}\ {*2^{-1}})}\ ;\ [b, \underline{\mathrm{map}_n\ b}] \\
= & \qquad \{\ \text{map through column (106)}\ \} \\
 & \underline{\mathrm{snd}\ b\ ;\ \mathrm{col}_n\ (\mathrm{fst}\ b\ ;\ +\ ;\ +^{-1}\ ;\ [{*2^{-1}}, b])}\ ;\ \mathrm{fst}\ b
\end{array}
$$

Observe that the component of the column has the form of a representation changer. Its abstract domain is however still a large type, namely the identity relation $id_{\mathcal{Z}}$ on the integers. We make it a small type, in fact $\llbracket 0,1,2,3 \rrbracket$, by pushing the type constraint $\text{snd}\ b$ on the domain of the column through to the argument program, using column rippling (lemma 109). This law is used here with $A = b$ and $S = \text{fst}\ b\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}, b]$. Let us verify the precondition:

$$\text{snd}\ A\ ;\ S\ \vdash\ \text{fst}\ A$$

$$\equiv \qquad \{\ \text{assignments}\ \}$$

$$[b,b]\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}, b]\ =\ [b,b]\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}\ ;\ b, b]$$

We verify this identity as follows:

$$\underline{[b,b]\ ;\ +}\ ;\ +^{-1}\ ;\ [*2^{-1}, b]$$

$$= \qquad \{\ \text{addition}\ \}$$

$$[b,b]\ ;\ +\ ;\ \underline{\llbracket 0,1,2 \rrbracket\ ;\ +^{-1}\ ;\ [*2^{-1}, b]}$$

$$= \qquad \{\ \text{converse}\ \}$$

$$[b,b]\ ;\ +\ ;\ ([*2, \underline{b}]\ ;\ \underline{+\ ;\ \llbracket 0,1,2 \rrbracket})^{-1}$$

$$= \qquad \{\ \text{shunting}\ \}$$

$$[b,b]\ ;\ +\ ;\ ([\underline{*2\ ;\ \llbracket -1,0,1,2 \rrbracket}, b]\ ;\ +)^{-1}$$

$$= \qquad \{\ \text{shunting}\ \}$$

$$[b,b]\ ;\ +\ ;\ \underline{([b\ ;\ *2, b]\ ;\ +)^{-1}}$$

$$= \qquad \{\ \text{converse}\ \}$$

$$[b,b]\ ;\ +\ ;\ +^{-1}\ ;\ [*2^{-1}\ ;\ b, b]$$

The precondition can also be verified, as shown below, in the form using $>$:

$$(\text{snd}\ A\ ;\ S)>\ \lhd\ \text{fst}\ A$$

$$\equiv \qquad \{\ \text{assignments}\ \}$$

$$([b, b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1}, b])_{>} \quad \lhd \quad \text{fst } b$$

We verify this approximation as follows:

$$\underline{([b, b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1}, b])_{>}}$$

$$\lhd \qquad \{ \text{ composition } (77) \ \}$$

$$\underline{(([b, b] \ ; \ +)_{>}} \ ; \ +^{-1} \ ; \ [*2^{-1}, b])_{>}$$

$$= \qquad \{ \text{ addition } \}$$

$$\underline{(\llbracket 0, 1, 2 \rrbracket \ ; \ +^{-1} \ ; \ [*2^{-1}, b])_{>}}$$

$$= \qquad \{ \text{ duality } (81) \ \}$$

$$([*2, \underline{b}] \ ; \ \underline{+ \ ; \ \llbracket 0, 1, 2 \rrbracket})_{<}$$

$$= \qquad \{ \text{ shunting } \}$$

$$([\underline{*2 \ ; \ \llbracket -1, 0, 1, 2 \rrbracket}, b] \ ; \ +)_{<}$$

$$= \qquad \{ \text{ shunting } \}$$

$$\underline{([b \ ; \ *2, b] \ ; \ +)_{<}}$$

$$\lhd \qquad \{ \text{ composition } (75) \ \}$$

$$[b, b]$$

$$\lhd \qquad \{ \text{ par } \}$$

$$\text{fst } b$$

Continuing with the $\mathit{add1}_n$ calculation:

$$\underline{\text{snd } b \ ; \ \text{col}_n \ (\text{fst } b \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1}, b])} \ ; \ \text{fst } b$$

$$= \qquad \{ \text{ column rippling } (109) \ \}$$

$$\text{col}_n \ \underline{([b, b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1} \ ; \ b, b])} \ ; \ \text{fst } b$$

$$= \qquad \{ \text{ def } 136 \ \}$$

$$\text{col}_n \ \mathit{HA} \ ; \ \text{fst } b$$

Under our assumption that $n > 0$ we have $\mathrm{col}_n\ HA \vdash \mathrm{fst}\ b$, i.e. the range constraint fst b can be eliminated. This does not hold if $n = 0$. The half–adder HA (a simple representation changer) can be implemented directly by using standard primitives, as shown in lemma 137. This completes the derivation of the binary addition program, which is now in the form of an implementation.

In summary, we have made the following transformation:

$$
\begin{array}{ll}
 & \mathit{add1}_n \\
= & \qquad \{ \text{ by definition } \} \\
 & [\mathit{bin}_n, b]\ ;\ +\ ;\ ([b\ ;\ (*2)^n, \mathit{bin}_n]\ ;\ +)^{-1} \\
= & \qquad \{ \text{ by calculation } \} \\
 & \mathrm{col}_n\ ([b, b]\ ;\ +\ ;\ \mathit{fork}\ ;\ [\mathit{div}\,2\ ;\ b, \mathit{mod}\,2\ ;\ b])
\end{array}
$$

The addition program that we have derived is a functional program, in that one could define functional versions of the Ruby combining forms in a language such as ML, and execute the addition program. What has been gained in deriving the program within a relational language? Our answer is that while the addition program is deterministic as a whole, it has a natural specification as a term with considerable internal non-determinism, which the derivation process eliminates.

The sequence of transformations “converse, sliding reductions, Horner’s rule, and map through column” as used in the derivation of the binary addition program is used again in this chapter. We combine the sequence in a single law:

Lemma 144:

$$
\begin{array}{ll}
 & \mathrm{fst}\ (\mathrm{map}_n\ R\ ;\ \mathrm{lad}_n\, x)\ ;\ +\ ;\ ([(*x)^n, \mathrm{map}_n\ S\ ;\ \mathrm{lad}_n\, x]\ ;\ +)^{-1} \\
= & \\
 & \mathrm{col}_n\ (\mathrm{fst}\ R\ ;\ +\ ;\ ([*x, S]\ ;\ +)^{-1}).
\end{array}
$$

Omitting the constraints b, the implementation for $\mathit{add1}_n$ can be translated directly into the notation of the Ruby interpreter presented in chapter 5:

```
> let add1 n =

    let rec con2 = inv p1 .. second (icon 2)

    and halfadd = ADD .. fork .. ((con2 .. DIV) !! (con2 .. MOD))

    in col n halfadd;
```

Now we compile the program for some size, say $n = 2$:

```
> rc (add1 2);

   Name      Domain             Range
   ----------------------------------
   ADD       <w1,w2>            w3
   ----------------------------------
   DIV       <w3,2>             w4
   MOD       <w3,2>             w5
   ----------------------------------
   ADD       <w6,w4>            w7
   ----------------------------------
   DIV       <w7,2>             w8
   MOD       <w7,2>             w9
   ----------------------------------

   Primitives   -  6
   Delays       -  0
   Longest path -  4
   Parallelism  -  10%

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

   Wiring -  <<w6,w1>,w2> ~ <w8,<w9,w5>>

   Inputs -  w6 w1 w2
```

Now we can do some simulation:

```
> rsim "0 0 0; 0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0; 1 1 1";

    0 -  ((0,0),0) ~ (0,(0,0))
    1 -  ((0,0),1) ~ (0,(0,1))
    2 -  ((0,1),0) ~ (0,(0,1))
    3 -  ((0,1),1) ~ (0,(1,0))
```

```
4 -  ((1,0),0) ~ (0,(1,0))
5 -  ((1,0),1) ~ (0,(1,1))
6 -  ((1,1),0) ~ (0,(1,1))
7 -  ((1,1),1) ~ (1,(0,0))
```

7.5 Binary addition II

In this section we derive a program that takes two n–bit binary numbers ($n > 0$) and a carry–in, and gives a carry–out and an n–bit sum. This program generalises that of the previous section, being implemented in terms of a column of full–adders rather than half–adders. Given below is our specification; R converts the binary numbers to integers and adds them together with the carry–in, S^{-1} separates off the carry–out, and converts the remaining integer back to binary:

Definition 145: $add_n \ \widehat{=} \ R \ ; \ S^{-1}$, where

$$R = [[bin_n, bin_n] \ ; \ +, b] \ ; \ +,$$

$$S = [b \ ; \ (*2)^n, bin_n] \ ; \ +.$$

In deriving an implementation for add_n, much of the work in deriving an implementation for $add1_n$ in the previous section can be reused:

$$add_n$$

$= \quad \{ \text{ def } 145 \ \}$

$$[\underline{[bin_n, bin_n] \ ; \ +}, b] \ ; \ + \ ; \ ([b \ ; \ (*2)^n, bin_n] \ ; \ +)^{-1}$$

$= \quad \{ \text{ lemma } 132 \ \}$

$$[zip_n \ ; \ \underline{csv_n}, b] \ ; \ + \ ; \ ([b \ ; \ (*2)^n, \underline{bin_n}] \ ; \ +)^{-1}$$

$= \quad \{ \text{ defs } 128{,}127 \ \}$

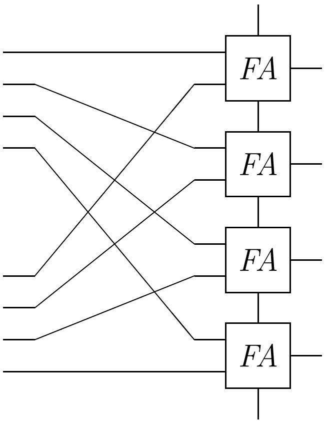
$$[zip_n \ ; \ \underline{\mathrm{map}_n \ ([b, b] \ ; \ +) \ ; \ \mathrm{lad}_n \, 2}, b] \ ; \ \underline{+} \ ; \ ([b \ ; \ \underline{(*2)^n}, \underline{\mathrm{map}_n \ b \ ; \ \mathrm{lad}_n \, 2}] \ ; \ \underline{+})^{-1}$$

$= \quad \{ \text{ as for } add1_n \ (144) \ \}$

$$[zip_n, \underline{b}] \ ; \ \mathrm{col}_n \ (\underline{\mathrm{fst} \ ([b, b] \ ; \ +)} \ ; \ + \ ; \ +^{-1} \ ; \ [\underline{*2^{-1}}, b]) \ ; \ \mathrm{fst} \ b$$

$$
\begin{array}{ll}
= & \quad \{ \text{ column rippling (109) } \} \\
 & \text{fst } \mathit{zip}_n \ ; \ \text{col}_n \ (\underline{[[b,b] \ ; \ +, b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1} \ ; \ b, b]}) \ ; \ \text{fst } b \\
= & \quad \{ \text{ lemma 140 } \} \\
 & \text{fst } \mathit{zip}_n \ ; \ \text{col}_n \ (\mathit{lsh} \ ; \ \mathit{FA}) \ ; \ \text{fst } b
\end{array}
$$

The full–adder *FA* can be implemented in terms of the half–adder, as shown in lemma 142. This completes the derivation of the binary addition program, which is now in the form of an implementation. A picture is given below:



.

Note that column rippling is used in the derivation above with $A = b$ and $S = \text{fst } ([b,b] \ ; \ +) \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1}, b]$. Verifying the precondition in this case differs little from that for *add1*$_n$, but is included for completeness:

$$
\begin{array}{ll}
 & \text{snd } A \ ; \ S \ \vdash \ \text{fst } A \\
\equiv & \quad \{ \text{ assignments } \} \\
 & [[b,b] \ ; \ +, b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1}, b] \ = \ [[b,b] \ ; \ +, b] \ ; \ + \ ; \ +^{-1} \ ; \ [*2^{-1} \ ; \ b, b]
\end{array}
$$

We verify this identity as follows:

$$
\begin{array}{ll}
 & \underline{[[b,b] \ ; \ +, b] \ ; \ +} \ ; \ +^{-1} \ ; \ [*2^{-1}, b] \\
= & \quad \{ \text{ addition } \} \\
 & [[b,b] \ ; \ +, b] \ ; \ + \ ; \ \underline{\llbracket 0,1,2,3 \rrbracket \ ; \ +^{-1} \ ; \ [*2^{-1}, b]}
\end{array}
$$

$$= \qquad \{ \text{ converse } \}$$

$$[[b,b] \ ; +,b] \ ; + \ ; ([*2, \underline{b}] \ ; \underline{+ \ ; \llbracket 0,1,2,3 \rrbracket})^{-1}$$

$$= \qquad \{ \text{ shunting } \}$$

$$[[b,b] \ ; +,b] \ ; + \ ; ([\underline{*2 \ ; \llbracket -1,0,1,2,3,}\rrbracket, b] \ ; +)^{-1}$$

$$= \qquad \{ \text{ shunting } \}$$

$$[[b,b] \ ; +,b] \ ; + \ ; \underline{([\llbracket 0,1 \rrbracket \ ; *2, b] \ ; +)^{-1}}$$

$$= \qquad \{ \text{ converse } \}$$

$$[[b,b] \ ; +,b] \ ; + \ ; +^{-1} \ ; [*2^{-1} \ ; b,b]$$

The precondition can also be verified, as shown below, in the form using ${\scriptstyle >}$:

$$(\text{snd } A \ ; S){\scriptstyle >} \ \lhd \ \text{fst } A$$

$$\equiv \qquad \{ \text{ assignments } \}$$

$$([[b,b] \ ; +] \ ; + \ ; +^{-1} \ ; [*2^{-1}, b]){\scriptstyle >} \quad \lhd \quad \text{fst } b$$

We verify this approximation as follows:

$$(\underline{[[b,b] \ ; +] \ ; +} \ ; +^{-1} \ ; [*2^{-1}, b]){\scriptstyle >}$$

$$\lhd \qquad \{ \text{ composition (77) } \}$$

$$(\underline{([[b,b] \ ; +] \ ; +){\scriptstyle >}} \ ; +^{-1} \ ; [*2^{-1}, b]){\scriptstyle >}$$

$$= \qquad \{ \text{ addition } \}$$

$$\underline{(\llbracket 0,1,2,3 \rrbracket \ ; +^{-1} \ ; [*2^{-1}, b]){\scriptstyle >}}$$

$$= \qquad \{ \text{ duality (81) } \}$$

$$([*2, \underline{b}] \ ; \underline{+ \ ; \llbracket 0,1,2,3 \rrbracket}){\scriptstyle <}$$

$$= \qquad \{ \text{ shunting } \}$$

$$([\underline{*2 \ ; \llbracket -1,0,1,2,3 \rrbracket}, b] \ ; +){\scriptstyle <}$$

$$= \qquad \{ \text{ shunting } \}$$

$$\underline{([b \ ; \ *2, b] \ ; \ +)_{<}}$$

$$\lhd \qquad \{ \text{ composition (75) } \}$$

$$[b, b]$$

$$\lhd \qquad \{ \text{ par } \}$$

$$\text{fst } b$$

7.6 Base conversion

In this section we derive a Ruby program that converts an n–digit number represented in base x to an m–digit number in base y, for $n, m, x, y > 0$. The circuit comes from pages 162–165 of the book “Digital systems, with algorithm implementation” [18]. Here is our specification for the conversion program:

Definition 146: $\mathit{conv}\, n\ x\ m\ y \ \ \widehat{=}\ \ \mathit{eval}_n\, x \ ; \ (\mathit{eval}_m\, y)^{-1}$.

The derivation of an implementation proceeds as follows:

$$\mathit{conv}\, n\ x\ m\ y$$

$$= \qquad \{ \text{ def 146 } \}$$

$$\underline{\mathit{eval}_n\, x} \ ; \ (\mathit{eval}_m\, y)^{-1}$$

$$= \qquad \{ \text{ lemma 125 } \}$$

$$\underline{\text{map}_n \,|x| \ ; \ {\pi_2}^{-1}} \ ; \ \text{fst } \llbracket 0 \rrbracket \ ; \ \text{rdl}_n \ (\text{fst } *x \ ; \ +) \ ; \ (\mathit{eval}_m\, y)^{-1}$$

$$= \qquad \{ \text{ shunting } \}$$

$${\pi_2}^{-1} \ ; \ [\llbracket 0 \rrbracket, \underline{\text{map}_n \,|x|}] \ ; \ \underline{\text{rdl}_n \ (\text{fst } *x \ ; \ +)} \ ; \ (\mathit{eval}_m\, y)^{-1}$$

$$= \qquad \{ \text{ map through left–reduce (105) } \}$$

$${\pi_2}^{-1} \ ; \ \text{fst } \llbracket 0 \rrbracket \ ; \ \underline{\text{rdl}_n \ ([*x, |x|] \ ; \ +) \ ; \ (\mathit{eval}_m\, y)^{-1}}$$

Now we use the induction rule for left–reduce (lemma 107), with $R \ = \ (\mathit{eval}_m\, y)^{-1}$, $S \ = \ \text{fst } \mathit{eval}_m\, y \ ; \ [*x, |x|] \ ; \ + \ ; \ (\mathit{eval}_m\, y)^{-1}$, and $T \ = \ [*x, |x|] \ ; \ +$. We verify the precondition of the law under these assignments as follows:

$$\text{fst}\ R\ ;\ S = T\ ;\ R$$

$$\equiv \qquad \{\ \text{assignments}\ \}$$

$$\text{fst}\ (\underline{(eval_m\, y)^{-1}\ ;\ eval_m\, y)}\ ;\ [*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1}$$

$$\quad = [*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1}$$

$$\equiv \qquad \{\ \text{lemma 122}\ \}$$

$$\text{fst}\ |\,y^m\,|\ ;\ [*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1}\ =\ [*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1}$$

$$\equiv \qquad \{\ \text{guarantees}\ \}$$

$$\text{fst}\ |\,y^m\,|\ \dashv\ [*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1}$$

$$\equiv \qquad \{\ \text{best domains (78)}\ \}$$

$$([*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1})_{<}\ \lhd\ \text{fst}\ |\,y^m\,|$$

We verify this approximation as follows:

$$\underline{([*x, |\,x\,|]\ ;\ +\ ;\ (eval_m\, y)^{-1})_{<}}$$

$$= \qquad \{\ \text{composition (77)}\ \}$$

$$([*x, |\,x\,|]\ ;\ +\ ;\ \underline{(eval_m\, y)^{-1}{}_{<}})_{<}$$

$$= \qquad \{\ \text{lemma 122}\ \}$$

$$([*x, \underline{|\,x\,|}]\ ;\ \underline{+\ ;\ |\,y^m\,|})_{<}$$

$$= \qquad \{\ \text{shunting}\ \}$$

$$(\underline{*x\ ;\ \llbracket -x+1, \ldots, y^m - 1 \rrbracket}, |\,x\,|]\ ;\ +)_{<}$$

$$= \qquad \{\ \text{shunting}\ \}$$

$$\underline{([\llbracket 0, \ldots, y^m - 1\ div\ x \rrbracket\ ;\ *x, |\,x\,|]\ ;\ +)_{<}}$$

$$\lhd \qquad \{\ \text{composition (75)}\ \}$$

$$[\llbracket 0, \ldots, y^m - 1\ div\ x \rrbracket, |\,x\,|]$$

$$\lhd \qquad \{\ \text{par}\ \}$$

$$\text{fst}\ \llbracket 0, \ldots, y^m - 1\ div\ x \rrbracket$$

$\lhd \qquad \{ \text{ arithmetic } \}$

$\text{fst}\ |\, y^m \,|$

We continue now with the *conv* calculation:

$$\pi_2{}^{-1}\ ;\ \text{fst}\ \llbracket 0 \rrbracket\ ;\ \underline{\text{rdl}_n\ ([*x, |\,x\,|]\ ;\ +)\ ;\ (\mathit{eval}_m\, y)^{-1}}$$

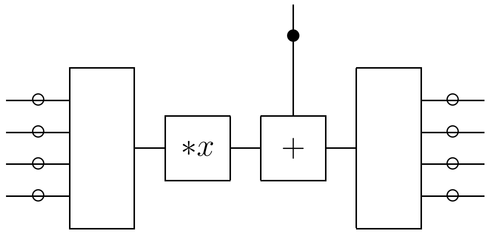
$= \qquad \{ \text{ left–reduce induction (107) } \}$

$$\underline{\pi_2{}^{-1}\ ;\ \text{fst}\ \llbracket 0 \rrbracket\ ;\ \text{fst}\ (\mathit{eval}_m\, y)^{-1}}\ ;\ \text{rdl}_n\ ([\mathit{eval}_m\, y\ ;\ *x, |\,x\,|]\ ;\ +\ ;\ (\mathit{eval}_m\, y)^{-1})$$

$= \qquad \{ \text{ fst } R\ ;\ \pi_2\ =\ \text{fst } R^{<}\ ;\ \pi_2\text{, lemma 123) } \}$

$$\pi_2{}^{-1}\ ;\ \text{fst}\ (\text{map}_m\ \llbracket 0 \rrbracket)\ ;\ \text{rdl}_n\ (\underline{[\mathit{eval}_m\, y\ ;\ *x, |\,x\,|]\ ;\ +\ ;\ (\mathit{eval}_m\, y)^{-1}})$$

It remains now to implement the underlined term, which we call *C1*. Here is a picture of *C1*, in which ○ represents $|\,y\,|$ and ● represents $|\,x\,|$:



.

Looking at this picture, the pattern of external wiring suggests that we should aim to express *C1* as a column. We begin by unfolding the definition of *C1*: (For the remainder of the calculation, $a{+}$ abbreviates fst $*a\ ;\ +$)

$\mathit{C1}$

$= \qquad \{ \text{ def } \}$

$$[\underline{\mathit{eval}_m\, y}\ ;\ *x, |\,x\,|]\ ;\ +\ ;\ (\underline{\mathit{eval}_m\, y})^{-1}$$

$= \qquad \{ \text{ lemma 124 } \}$

$$[\text{map}_m\ |\,y\,|\ ;\ \mathit{apl}_{m-1}{}^{-1}\ ;\ \text{rdl}_{m-1}\ y{+}\ ;\ *x, |\,x\,|]\ ;$$

$$+\ ;\ \underline{(\text{map}_m\ |\,y\,|\ ;\ \mathit{apl}_{m-1}{}^{-1}\ ;\ \text{rdl}_{m-1}\ y{+})^{-1}}$$

$$= \qquad \{ \text{ converse } \}$$

$$[\mathrm{map}_m\ |y|\ ;\ \mathit{apl}_{m-1}{}^{-1}\ ;\ \underline{\mathrm{rdl}_{m-1}\ y{+}\ ;\ {*x}}, |x|]\ ;$$

$$\underline{+\ ;\ (\mathrm{rdl}_{m-1}\ y{+})^{-1}}\ ;\ \mathit{apl}_{m-1}\ ;\ \mathrm{map}_m\ |y|$$

The two left–reductions can now be combined to give a column using lemma 113. This law is used here with $R = y{+}$, $S = x{+}$ and $T = y{+}$. One might expect to have to invent a W that satisfied the precondition fst R ; S ; $T^{-1} = \mathit{lsh}$; snd W ; rsh ; fst S to lemma 113, but we can in fact obtain such a W by calculation:

$$\mathrm{fst}\ R\ ;\ S\ ;\ T^{-1}$$

$$= \qquad \{ \text{ assignments } \}$$

$$\mathrm{fst}\ (\mathrm{fst}\ {*y}\ ;\ \underline{+\ ;\ {*x}})\ ;\ +\ ;\ +^{-1}\ ;\ \mathrm{fst}\ {*y}^{-1}$$

$$= \qquad \{ \text{ distribution (102) } \}$$

$$\mathrm{fst}\ ([{*y}\ ;\ {*x}, {*x}]\ ;\ \underline{+})\ ;\ \underline{+}\ ;\ +^{-1}\ ;\ \mathrm{fst}\ {*y}^{-1}$$

$$= \qquad \{ \text{ associativity (101) } \}$$

$$\underline{\mathrm{fst}\ [{*y}\ ;\ {*x}, {*x}]\ ;\ \mathit{lsh}}\ ;\ \mathrm{snd}\ +\ ;\ +\ ;\ +^{-1}\ ;\ \mathrm{fst}\ {*y}^{-1}$$

$$= \qquad \{ \text{ shunting } \}$$

$$\mathit{lsh}\ ;\ [{*y}\ ;\ {*x}, \mathrm{fst}\ {*x}]\ ;\ \mathrm{snd}\ +\ ;\ \underline{+\ ;\ +^{-1}}\ ;\ \mathrm{fst}\ {*y}^{-1}$$

$$= \qquad \{ \text{ addition (104) } \}$$

$$\mathit{lsh}\ ;\ [\underline{{*y}\ ;\ {*x}}, \mathrm{fst}\ {*x}\ ;\ +\ ;\ +^{-1}]\ ;\ \underline{\mathit{rsh}}\ ;\ \mathrm{fst}\ (+\ ;\ {*y}^{-1})$$

$$= \qquad \{ \text{ shunting } \}$$

$$\mathit{lsh}\ ;\ \mathrm{snd}\ (\mathrm{fst}\ {*x}\ ;\ +\ ;\ +^{-1})\ ;\ \mathit{rsh}\ ;\ \mathrm{fst}\ (\mathrm{fst}\ (\underline{{*y}\ ;\ {*x}})\ ;\ +\ ;\ {*y}^{-1})$$

$$= \qquad \{ \text{ multiplication } \}$$

$$\mathit{lsh}\ ;\ \mathrm{snd}\ (\mathrm{fst}\ {*x}\ ;\ +\ ;\ +^{-1})\ ;\ \mathit{rsh}\ ;\ \mathrm{fst}\ (\mathrm{fst}\ ({*x}\ ;\ \underline{{*y}})\ ;\ \underline{+\ ;\ {*y}^{-1}})$$

$$= \qquad \{ \text{ shunting (103) } \}$$

$$\mathit{lsh}\ ;\ \mathrm{snd}\ (\mathrm{fst}\ {*x}\ ;\ +\ ;\ +^{-1})\ ;\ \underline{\mathit{rsh}}\ ;\ \mathrm{fst}\ ([{*x}, \underline{{*y}^{-1}}]\ ;\ +)$$

$$= \qquad \{ \text{ shunting } \}$$

$$\mathit{lsh} \;;\; \mathrm{snd}\,(\underline{\mathrm{fst}\,{*x} \;;\; + \;;\; +^{-1} \;;\; \mathrm{fst}\,{*y^{-1}}}) \;;\; \mathit{rsh} \;;\; \mathrm{fst}\,(\mathrm{fst}\,{*x} \;;\; +)$$

$$= \qquad \{\, W := \ldots \,\}$$

$$\mathit{lsh} \;;\; \mathrm{snd}\,W \;;\; \mathit{rsh} \;;\; \mathrm{fst}\,S$$

We continue now with the *C1* calculation; by first applying a number of shunting laws we are able to push the maps through the column:

$$[\mathrm{map}_m\,|y| \;;\; {\mathit{apl}_{m-1}}^{-1} \;;\; \underline{\mathrm{rdl}_{m-1}\,y+ \;;\; {*x}}, |x|] \;;$$

$$\underline{+ \;;\; (\mathrm{rdl}_{m-1}\,y+)^{-1}} \;;\; \mathit{apl}_{m-1} \;;\; \mathrm{map}_m\,|y|$$

$$= \qquad \{\, \text{zipping reductions (113)} \,\}$$

$$[\underline{\mathrm{map}_m\,|y| \;;\; {\mathit{apl}_{m-1}}^{-1}}, |x|] \;;\; \mathit{lsh} \;;\; \mathrm{snd}\,(\mathrm{col}_{m-1}\,(x+ \;;\; y+^{-1})) \;;$$

$$\mathit{rsh} \;;\; \mathrm{fst}\,x+ \;;\; \mathit{apl}_{m-1} \;;\; \mathrm{map}_m\,|y|$$

$$= \qquad \{\, \text{shunting} \,\}$$

$$[{\mathit{apl}_{m-1}}^{-1} \;;\; \underline{[|y|, \mathrm{map}_{m-1}\,|y|]}, \underline{|x|}] \;;\; \underline{\mathit{lsh}} \;;\; \mathrm{snd}\,(\mathrm{col}_{m-1}\,(x+ \;;\; y+^{-1})) \;;$$

$$\mathit{rsh} \;;\; [x+ \;;\; |y|, \mathrm{map}_{m-1}\,|y|] \;;\; \mathit{apl}_{m-1}$$

$$= \qquad \{\, \text{shunting} \,\}$$

$$\mathrm{fst}\,({\mathit{apl}_{m-1}}^{-1}) \;;\; \mathit{lsh} \;;\; [\underline{|y|}, [\mathrm{map}_{m-1}\,|y|, |x|] \;;\; \mathrm{col}_{m-1}\,(x+ \;;\; y+^{-1})] \;;$$

$$\underline{\mathit{rsh}} \;;\; [x+ \;;\; |y|, \mathrm{map}_{m-1}\,|y|] \;;\; \mathit{apl}_{m-1}$$

$$= \qquad \{\, \text{shunting} \,\}$$

$$\mathrm{fst}\,({\mathit{apl}_{m-1}}^{-1}) \;;\; \mathit{lsh} \;;\; \mathrm{snd}\,([\mathrm{map}_{m-1}\,|y|, |x|] \;;\; \mathrm{col}_{m-1}\,(x+ \;;\; y+^{-1})) \;;$$

$$\underline{\mathit{rsh}} \;;\; [\mathrm{fst}\,|y| \;;\; x+ \;;\; |y|, \underline{\mathrm{map}_{m-1}\,|y|]} \;;\; \mathit{apl}_{m-1}$$

$$= \qquad \{\, \text{shunting} \,\}$$

$$\mathrm{fst}\,({\mathit{apl}_{m-1}}^{-1}) \;;\; \mathit{lsh} \;;\; \mathrm{snd}\,([\underline{\mathrm{map}_{m-1}\,|y|}, |x|] \;;\; \underline{\mathrm{col}_{m-1}\,(x+ \;;\; y+^{-1})} \;;$$

$$\mathrm{snd}\,(\underline{\mathrm{map}_{m-1}\,|y|})) \;;\; \mathit{rsh} \;;\; \mathrm{fst}\,(\mathrm{fst}\,|y| \;;\; x+ \;;\; |y|) \;;\; \mathit{apl}_{m-1}$$

$$= \qquad \{\, \text{map through column (106)} \,\}$$

$$\mathrm{fst}\,({\mathit{apl}_{m-1}}^{-1}) \;;\; \mathit{lsh} \;;\; \mathrm{snd}\,(\underline{\mathrm{snd}\,|x| \;;\; \mathrm{col}_{m-1}\,(\mathrm{fst}\,|y| \;;\; x+ \;;\; y+^{-1} \;;\; \mathrm{snd}\,|y|))} \;;$$

$$\mathit{rsh} \ ; \ \mathrm{fst}\,(\mathrm{fst}\,|y| \ ; \ x{+} \ ; \ |y|) \ ; \ \mathit{apl}_{m-1}$$

Observe that the component of the column has the form of a representation changer; its abstract domain is however still a large type, namely the identity relation $\mathit{id}_{\mathcal{Z}}$ on the integers. We make it a small type, in fact $|xy|$, by pushing the type constraint $(\mathrm{snd}\, x)$ on the domain of the column through to the argument program, using column rippling (lemma 109). This law is used here with $A = |x|$ and $S = \mathrm{fst}\,|y|$; $x{+}$; $y{+}^{-1}$; $\mathrm{snd}\,|y|$. Let us verify the precondition:

$$\begin{array}{ll}
 & (\mathrm{snd}\, A \ ; \ S)_{>} \ \lhd \ \mathrm{fst}\, A \\
= & \qquad \{ \text{ assignments } \} \\
 & ([|y|,|x|] \ ; \ x{+} \ ; \ y{+}^{-1} \ ; \ \mathrm{snd}\,|y|)_{>} \ \lhd \ \mathrm{fst}\,|x|
\end{array}$$

We verify this approximation as follows:

$$\begin{array}{ll}
 & \underline{([|y|,|x|] \ ; \ x{+} \ ; \ y{+}^{-1} \ ; \ \mathrm{snd}\,|y|)_{>}} \\
\lhd & \qquad \{ \text{ composition (77) } \} \\
 & (\underline{([|y|,|x|] \ ; \ x{+})_{>}} \ ; \ y{+}^{-1} \ ; \ \mathrm{snd}\,|y|)_{>} \\
= & \qquad \{ \text{ domains } \} \\
 & (|yx| \ ; \ \underline{y{+}^{-1}} \ ; \ \mathrm{snd}\,|y|)_{>} \\
= & \qquad \{ \text{ def } \} \\
 & (\underline{|yx| \ ; \ +^{-1}} \ ; \ [*y^{-1}, \underline{|y|}])_{>} \\
= & \qquad \{ \text{ shunting } \} \\
 & (+^{-1} \ ; \ [\underline{[\![-y+1,\ldots,yx-1]\!] \ ; \ *y^{-1}}, |y|])_{>} \\
= & \qquad \{ \text{ shunting } \} \\
 & \underline{(+^{-1} \ ; \ [*y^{-1} \ ; \ |x|, |y|])_{>}} \\
\lhd & \qquad \{ \text{ composition (75) } \} \\
 & [|x|, |y|]
\end{array}$$

$$\lhd \qquad \{\ \text{par}\ \}$$

$$\text{fst}\ |\,x\,|$$

We continue again with the *C1* calculation:

$$\text{fst}\ (\mathit{apl}_{m-1}{}^{-1})\ ;\ \mathit{lsh}\ ;\ \text{snd}\ (\underline{\text{snd}\ |\,x\,|\ ;\ \text{col}_{m-1}\ (\text{fst}\ |\,y\,|\ ;\ x{+}\ ;\ y{+}^{-1}\ ;\ \text{snd}\ |\,y\,|)})\ ;$$
$$\mathit{rsh}\ ;\ \text{fst}\ (\text{fst}\ |\,y\,|\ ;\ x{+}\ ;\ |\,y\,|)\ ;\ \mathit{apl}_{m-1}$$

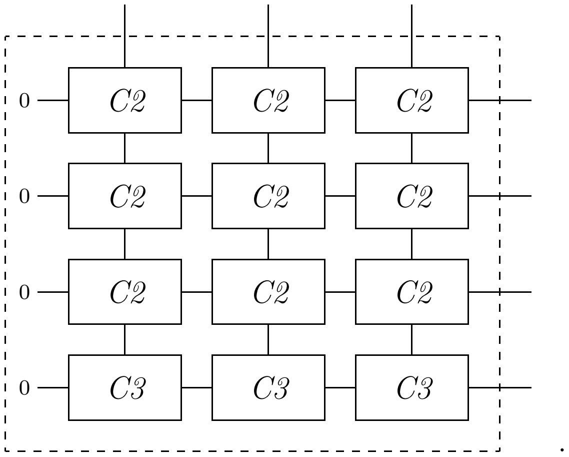
$$= \qquad \{\ \text{column rippling (109)}\ \}$$

$$\text{fst}\ (\mathit{apl}_{m-1}{}^{-1})\ ;\ \mathit{lsh}\ ;\ \text{snd}\ (\text{col}_{m-1}\ ([|\,y\,|,|\,x\,|]\ ;\ x{+}\ ;\ y{+}^{-1}\ ;\ [|\,x\,|,|\,y\,|])\ ;\ \underline{\text{fst}\ |\,x\,|})\ ;$$
$$\underline{\mathit{rsh}}\ ;\ \text{fst}\ (\text{fst}\ |\,y\,|\ ;\ x{+}\ ;\ |\,y\,|)\ ;\ \mathit{apl}_{m-1}$$

$$= \qquad \{\ \text{shunting}\ \}$$

$$\text{fst}\ (\mathit{apl}_{m-1}{}^{-1})\ ;\ \mathit{lsh}\ ;\ \text{snd}\ (\text{col}_{m-1}\ (\underline{[|\,y\,|,|\,x\,|]\ ;\ x{+}\ ;\ y{+}^{-1}\ ;\ [|\,x\,|,|\,y\,|]}))\ ;$$
$$\mathit{rsh}\ ;\ \text{fst}\ (\underline{[|\,y\,|,|\,x\,|]\ ;\ x{+}\ ;\ |\,y\,|})\ ;\ \mathit{apl}_{m-1}$$

We call the underlined terms *C2* and *C3* respectively. The *conv* program now has a grid–like layout, as shown below:



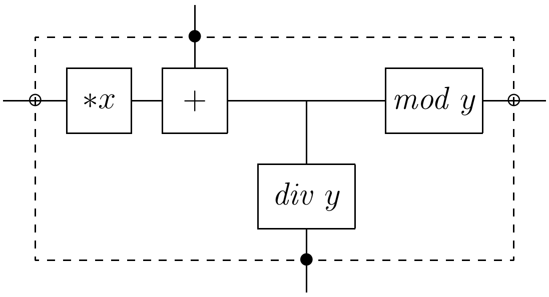
The program *C3* is already in the form of an implementation. An implementation for *C2* is obtained simply by applying lemma 138. This completes the derivation of the base conversion program, which is now in the form of an implementation.

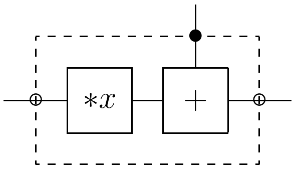
In summary, we have made the following transformation:

$$
\begin{array}{ll}
 & \mathit{conv}\, n\ x\ m\ y \\
= & \quad \{ \text{ by definition } \} \\
 & \mathit{eval}_n\, x \ ;\ (\mathit{eval}_m\, y)^{-1} \\
= & \quad \{ \text{ by calculation } \} \\
 & {\pi_2}^{-1} \ ;\ \mathrm{fst}\ (\mathrm{map}_m\ [\![0]\!]) \ ;\ \mathrm{rdl}_n\ \mathit{C1} \\
 & \text{where}
\end{array}
$$

$$
\begin{aligned}
\mathit{C1} &= \mathrm{fst}\ ({\mathit{apl}_{m-1}}^{-1}) \ ;\ \mathit{lsh} \ ;\ \mathrm{snd}\ (\mathrm{col}_{m-1}\ \mathit{C2}) \ ;\ \mathit{rsh} \ ;\ \mathrm{fst}\ \mathit{C3} \ ;\ \mathit{apl}_{m-1} \\
\mathit{C2} &= [|y| \ ;\ *x, |x|] \ ;\ + \ ;\ \mathit{fork} \ ;\ [\mathit{div}\ y \ ;\ |x|, \mathit{mod}\ y \ ;\ |y|] \\
\mathit{C3} &= [|y| \ ;\ *x, |x|] \ ;\ + \ ;\ |y|
\end{aligned}
$$

To get a feel for the layout of the *conv* implementation, mentally slot into the grid above the pictures given below for the implementations *C2* and *C3*.





Omitting the constraints $|x|$ and $|y|$, the base conversion program above can be translated directly into notation of the Ruby interpreter presented in chapter 5:

```
let conv n x m y =
   let rec cell1 = first (inv (apl (m-1))) .. lsh
                  .. second (col (m-1) cell2)
                  .. rsh .. first cell3 .. apl (m-1)

   and     cell2 = first (n2 x .. MULT) .. ADD
                  .. fork .. ((n2 y .. DIV) !! (n2 y .. MOD))

   and     cell3 = first (n2 x .. MULT) .. ADD

   and     n2 a  = inv p1 .. second (icon a)

   in inv p2 .. first (rmap m (icon 0)) .. rdl n cell1;
```

Suppose now that we want to convert an 8–digit binary number to a 5–digit ternary number. First we compile the conversion program at this particular instance: `rc (conv 8 2 5 3)`. Now we can do some simulation. For example, `rsim "0 0 0 0 0 1 0 1"` produces the output `(0,0,0,1,2)`. The input in this example is the binary representation of the number 5, the output is the ternary representation of 5. What if the n–digit base–x input number is not representible as an m–digit base–y number? Omitting the constraints $|x|$ and $|y|$ from the base conversion program (as in the interpreter version), every output digit x except the most significant digit is guaranteed to be within the expected range $0 \leq x < y$.

Chapter 8

Summary and Future Work

This thesis is concerned with deriving programs using the relational language Ruby. Our contribution to Ruby divides into two parts. Firstly, we defined what it means for a Ruby program to be an implementation, and built an interpreter for such programs. Secondly, we showed that the operators $<$ and $>$ that give the best left and right types for a relation can be used to verify the precondition of induction laws applied during a derivation. In this chapter we summarise our achievements in more detail, and suggest some directions for future work.

8.1 Summary

In [39] Jones and Sheeran talked briefly about the difference between an implementation and a specification in Ruby. In chapter 4 we made this idea precise, via the notion of a *causal relation*, and the *network* denoted by a Ruby program. Causal relations generalise functional relations; a relation is causal if it is functional in some structural way, but not necessarily from domain to range. We presented a categorical definition for causality (based upon the representation of relations as subobjects of a binary product), and showed that causal relations are closed under converse and par, but not under composition. We introduced the idea of a set R^d of *directions* for a relational program R. Directions tell us which parts of a program can be inputs, and which can be outputs. The d operator behaves well for converse and par, but not for composition. We defined a causal Ruby program as being an implementation if one can label each wire in the network denoted by the program with a direction *in* or *out*, such that a few simple conditions are satisfied. The idea is that such a labelled network is like a data-flow network (and can be executed in a similar way), except that the primitives are causal relations rather than functions. Since a causal relation can be functional in more than one way, there may be more than one way in which a network can be so labelled.

The definition of 'implementation' in chapter 4 is general, but not completely formal. In chapter 5 we gave a formal definition for a natural sub-class of the implementations, and presented an interpreter for such Ruby programs. The interpreter is based upon two key ideas. The first is the translation of a Ruby program to the corresponding network of primitive relations. Executing such a network is a simple process; trying to execute the original Ruby program directly would be complicated by the possibility that data can flow both leftwards and rightwards over composition. The other key idea in the interpreter is the use of Lazy ML as a meta-language for building Ruby programs. This avoids the need for a parser, and means that the full power of Lazy ML is available for defining new combining forms. Of the combining forms of Ruby, only composition, converse, and product are built-in to the interpreter. All the other combining forms, including all generic combining forms, are defined in terms of these three operators.

Fundamental to the use of type information in deriving programs is the idea of having types as special kinds of programs. In chapter 6 we introduced the idea of partial equivalence relations (pers) as types in Ruby. Types are used in two different ways in Ruby: as *constraints* within programs (for example, if $b = \{(0,0),(1,1)\}$ then composing the type $[b,b]$ on the left of $+$ 'constrains' it to working only with bits); and as *abstractions* of programs (for example, if A is a left or right domain of R, then A encodes something about the internal structure of R, and is in this sense an 'abstraction' of R.) The smallest left and right domains under the ordering $\lhd$ on pers are given by domain operators $<$ and $>$. Pers can be viewed as those relations that can be expressed as the union of disjoint full relations. Generalising from full relations $\mathcal{S} \times \mathcal{S}$ to products $\mathcal{S} \times \mathcal{T}$ gives rise to the notion of a *difunctional* relation. Equivalently, the difunctionals are the 'invertible' relations, the 'per–functional' relations, or the relations that can be expressed as the composition of a functional relation and the converse of a functional relation. All the programs that we derive in chapter 7 are difunctional programs specified as such a composition.

In chapter 7 we introduced the term *representation changer* for a program that converts an abstract value from one concrete representation to another concrete representation. A great many algorithms are examples of representation changers. It is natural to specify such programs as a composition $f \ ; \ g^{-1}$, where f is a functional relation that converts from the first concrete type to the abstract type, and g is a functional relation that converts from the second concrete type to the abstract type. Being a little more general, we defined a representation changer as a difunctional program that can be specified in the form $R \ ; \ S^{-1}$, where R and S are difunctional relations. The composition of two such difunctionals is not always difunctional; in chapter 6 we gave a necessary and sufficient condition (involving types) for closure under composition, and some sufficient conditions that one might use in practice to check that the composition of two difunctionals is difunctional.

In chapter 7 we use Ruby to derive implementations for a number of representation changers. Refinement of the specification $R \ ; \ S^{-1}$ proceeds by sliding parts of R and S through one another, aiming towards a new program with components that are representation changers with smaller abstract types. In this sense, thinking about types guides our derivations. The process is repeated until the remaining representation changers can be implemented directly using a few standard primitives. It is encouraging to find that the same patterns of transformations are used again and again when implementing representation changers. The preconditions to 'induction' laws that are applied during derivation of such programs typically work out to be assertions of the form $A \ ; \ R = R$. In the past, such assertions have been verified by informal arguments or by using predicate calculus, rather than by applying algebraic laws from Ruby. In chapter 7 we verify such assertions without stepping outside Ruby, by first expressing them in the equivalent form $R{<} \lhd A$, which can then be verified using algebraic properties of the $<$ operator.

On a less technical note, in this thesis I decided to underline the parts of a formula to which laws are being applied in moving from one step to the next in a calculation. This has proved to be of great benefit, guiding the eye to the parts being changed, and helping in pattern matching against the laws, particularly when the formula properly matches the law only after some simple re–arranging. As a byproduct, many of the hints between steps become simpler too.

8.2 Future work

When deriving Ruby programs, computer–based tools could be useful. An example of such a tool is our interpreter for implementations. Other tools that one could imagine include a system for checking that the steps in a derivation are valid, or a system for helping the designer during the derivation process, pattern matching against laws, applying transformations, and keeping track of assumptions and preconditions to be verified. Such tools are practical only when derivations are fully formal; our use of the domain operators to avoid stepping outside Ruby to verify assertions about types is an important advance in this respect.

Difunctional relations and causal relations both generalise the notion of a functional relation: difunctionals can be expressed as the composition of a functional relation and the converse of a functional relation; causal relations must be functional in some structural way, but are not restricted to having inputs in the domain and outputs in the range. Perhaps the two classes of relations can be combined in some useful way? A promising approach might be to define a relation as being *dicausal* if it is difunctional in some structural way, rather than just being functional in some structural way. Extending the Ruby interpreter to work with dicausal relations would allow execution of many programs before they were in the form of an implementation. By carefully exploiting the regular way in which difunctional relations are non-deterministic (difunctionals can be expressed as the union of disjoint products of sets), such an interpreter could still be efficient.

While many program derivations have been made using Ruby, no *refinement* ordering on Ruby terms has ever been defined. The idea behind such an ordering is to have a measure on terms that is decreased as a specification is transformed to an executable term. There is a standard refinement ordering $\leq$ on relational terms [31], defined by $R \leq S$ iff $R \subseteq S$ and $dom(R) = dom(S)$. This ordering expresses that we can eliminate non-determinism, but must preserve the domain of the original term. Although it has never been made precise it is clear that the notion of refinement in Ruby is different, being more concerned with the denotation of a term as a network of relations rather than just as a single relation. We hope that the notion of refinement for Petri nets introduced by Brown and Gurr [11] can be adapted to define a notion of refinement for Ruby terms.

Relations of type $\mathcal{A} \leftrightarrow \mathcal{B}$ are in one-to-one correspondence with functions of type $\mathcal{A} \rightarrow P\mathcal{B}$. Why then not just stay within a functional language, for example Squiggol, and admit sets as a type? Our answer is that the algebra of relations is much cleaner than the algebra of set–valued functions; compare our relational calculations with the functional calculations in [19]. Generalising from functions to relations brings many advantages. Are there yet more general calculi with even more advantages? In [20] de Moor observes that functions $P\mathcal{A} \rightarrow P\mathcal{B}$ (predicate transformers [24]) generalise relations in the same way that relations generalise functions. There is a wealth of work in the use of predicate transformers in deriving programs. It would be interesting to compare with the use of binary relations.

There is much categorical literature about binary relations. Indeed, relations are a central topic of Freyd and Scedrov's recent book [28]. I am hopeful that categorical results will suggest new developments in calculational programming; just recently in fact, Bird and de Moor [20, 9] have developed a calculational paradigm based in a category of relations. Conversely, experience with the use of relations in calculating programs can lead to new developments in the categorical treatment of relations; de Moor's thesis [20] contains a number of new categorical results.

Bibliography

[1] R.J.R. Back. Correctness preserving program refinements: Proof theory and applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[2] Roland Backhouse. Sums and differentials. *The Squiggolist*, 2(2), November 1991.

[3] Roland Backhouse. Demonic operators and monotype factors. Eindhoven University of Technology, February 1992.

[4] Roland Backhouse, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. In *Proc. STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[5] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *CACM*, 9, August 1978.

[6] Michael Barr. Relational algebras. In *Reports of the Midwest Category Seminar IV*, volume 137 of *Lecture Notes in Mathematics.* Springer-Verlag, 1970.

[7] Rudolf Berghammer. Relational specification of data types and programs. Report 9109, Universitat der Bundeswehr Munchen, September 1991.

[8] Richard Bird. Constructive functional programming. In *Proc. Marktoberdorf International Summer School on Constructive Methods in Computer Science.* Springer-Verlag, 1989.

[9] Richard Bird and Oege de Moor. From dynamic programming to greedy algorithms. In *Proc. STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[10] Richard Bird and Philip Wadler. *An Introduction to Functional Programming.* Prentice Hall, 1988.

[11] Carolyn Brown and Doug Gurr. A categorical linear framework for petri nets. In *Proc. LICS*, 1990.

[12] Carolyn Brown and Doug Gurr. Relations and non-commutative linear logic. Technical Report DAIMI PB – 372, Aarhus University, November 1991.

[13] Carolyn Brown and Doug Gurr. A representation theorem for quantales. *Journal of Pure and Applied Algebra*, 1991. To appear.

[14] Rod Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[15] A. Carboni, G. M. Kelly, and R. J. Wood. A 2-categorical approach to geometric morphisms, I. *Cahiers de Topologie et Geometrie Differentielle Categoriques*, 32(1):47–95, 1991.

[16] Aurelio Carboni, Stefano Kasangian, and Ross Street. Bicategories of spans and relations. *Journal of Pure and Applied Algebra*, 33:259–267, 1984.

[17] D. M. Cattrall. *The Design and Implementation of a Relational Programming System.* PhD thesis, University of York, 1992.

[18] M. Davio, J.-P. Deschamps, and A. Thayse. *Digital Systems, with Algorithm Implementation.* John Wiley & Sons, 1983.

[19] Oege de Moor. Indeterminacy in optimization problems. In *Proc. Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1989.

[20] Oege de Moor. *Categories, Relations and Dynamic Programming.* PhD thesis, Oxford University, April 1992. Available as Research Report PRG-98.

[21] J. Desharnais. *Abstract Relational Semantics.* PhD thesis, McGill University, Montreal, 1989.

[22] Edsger W. Dijkstra. A relational summary. Report EWD1047, University of Texas at Austin, November 1990.

[23] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison-Wesley, 1988.

[24] Edsger W. Dijsktra and Caroll Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, 1990.

[25] Wim Feijen and Netty van Gasteren. An introduction into the relational calculus. Report AvG91/WF140, Eindhoven University of Technology, 1991.

[26] Maarten Fokkinga. A gentle introduction to category theory: The calculational approach. In *Proc. STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[27] Maarten Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, March 1992.

[28] Peter Freyd and Andre Scedrov. *Categories, Allegories.* North-Holland, 1990.

[29] Jeremy Gibbons. *Algebras for Tree Algorithms.* PhD thesis, Oxford University, September 1991.

[30] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* Springer-Verlag, 1980.

[31] C.A.R. Hoare and Jifeng He. The weakest prespecification. *Information Processing Letters*, 24:127–132, 1987.

[32] John Hughes and John Launchbury. Reversing abstract interpretations. In *Proc. European Symposium on Programming*, Rennes, 1992.

[33] Graham Hutton. Functional programming with relations. In *Proc. Third Glasgow Workshop on Functional Programming*, Workshops in Computing. Springer-Verlag, 1990.

[34] Graham Hutton. A relational derivation of a functional program. In *Proc. STOP Summer School on Constructive Algorithmics*, Ameland, The Netherlands, September 1992.

[35] Graham Hutton and Ed Voermans. A calculational theory of pers as types. Research Report 1992/R1, Glasgow University, January 1992.

[36] A. Jaoua, A. Mili, N. Boudriga, and J. L. Durieux. Regularity of relations: A measure of uniformity. *Theoretical Computer Science*, 79:323–339, 1991.

[37] Geraint Jones. Designing circuits by calculation. Technical Report PRG-TR-10-90, Oxford University, April 1990.

[38] Geraint Jones. A certain loss of identity. Technical Report PRG-TR-14-92, Oxford University, 1992.

[39] Geraint Jones and Mary Sheeran. Relations + higher-order functions = hardware descriptions. In *Proc. IEEE Comp Euro 87: VLSI and Computers*, May 1987.

[40] Geraint Jones and Mary Sheeran. Timeless truths about sequential circuits. In Tewksbury et al., editors, *Concurrent Computations: Algorithms, Architectures and Technology*, New York, 1988. Plenum Press.

[41] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam, 1990. Elsevier Science Publications.

[42] Geraint Jones and Mary Sheeran. Relations and refinement in circuit design. In Morgan, editor, *Proc. BCS FACS Workshop on Refinement*, Workshops in Computing. Springer-Verlag, 1991.

[43] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In *Proc. Second International Conference on Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer-Verlag, 1992. To appear.

[44] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[45] G. Kahn. *The Semantics of a Simple Language for Parallel Processing.* Number 74 in Information Processing Letters. North-Holland, 1977.

[46] Wayne Luk, Geraint Jones, and Mary Sheeran. Computer-based tools for regular array design. In McCanny, McWhirter, and Schwartzlander, editors, *Systolic Array Processors.* Prentice Hall, 1989.

[47] Bruce MacLennan. Relational programming. Technical Report NPS52-83-012, Navel postgraduate School, Monterey, CA 93943, September 1983.

[48] Grant Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Groningen University, 1990.

[49] Lambert Meertens. Algorithmics: Towards programming as a mathematical activity. In *Proc. CWI Symposium*, Centre for Mathematics and Computer Science, Amsterdam, November 1983.

[50] Erik Meijer. *Calculating Compilers.* PhD thesis, Nijmegen University, February 1992.

[51] A.C. Melton, D.A. Schmidt, and G.E. Strecker. Galois connections and computer science applications. In Pitt, Poigné, and Rydeheard, editors, *Category Theory and Computer Science*, number 283 in LNCS. Springer-Verlag, 1987.

[52] Carroll Morgan. The specification statement. *ACM TOPLAS*, 10:403–419, July 1988.

[53] Joe Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:287–306, 1987.

[54] J. Riguet. Relations binaires, fermetures, correspondances de galois. *Bulletin de la Societé mathématique de France*, 76, 1948.

[55] K. Rosenthall. *Quantales and their Applications*, volume 234 of *Pitman Research Notes in Math.* Longman, 1990.

[56] G. Schmidt and T. Ströhlein. *Relations and Graphs.* Springer-Verlag, 1992.

[57] Mary Sheeran. *μFP – an Algebraic VLSI Design Language.* PhD thesis, University of Oxford, 1983. Available as Research Report PRG-39.

[58] Mary Sheeran. Describing and reasoning about circuits using relations. In Tucker et al., editors, *Proc. Workshop in Theoretical Aspects of VLSI*, Leeds, 1986.

[59] Mary Sheeran. Retiming and slowdown in ruby. In Milne, editor, *The Fusion of Hardware Design and Verification.* North-Holland, 1988.

[60] Mary Sheeran. Describing butterfly networks in Ruby. In *Proc. Glasgow Workshop on Functional Programming*, Workshops in Computing, Fraserburgh, 1989. Springer-Verlag.

[61] Mary Sheeran. A note on abstraction in Ruby. In *Proc. Glasgow Workshop on Functional Programming*, Workshops in Computing, Portree, 1991. Springer-Verlag.

[62] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, September 1941.

[63] Alfred Tarski. A lattice-theoretic fixed point theorem and its applications. *Pacific Journal of Mathematics*, pages 285–309, 1955.

[64] Jaap van der Woude. Preliminaries on lattice theory. Report JCSP16, CWI, Amsterdam, November 1988.

[65] Jaap van der Woude. Pers per se. CWI, Amsterdam, November 1991.

[66] A.J.M. van Gasteren. *On the Shape of Mathematical Arguments*. Number 445 in LNCS. Springer-Verlag, Berlin, 1990.

[67] Paulo Veloso and Armando Haeberer. A finitely relational algebra for classical first-order logic. Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 1991.

[68] Paulo Veloso and Armando Haeberer. Partial relations for program derivation. In *Proc. IFIP WG-2.1 Working Conference on Constructing Programs from Specifications*, Pacific Grove, USA, May 1991.

[69] Ed Voermans. The equivalence domain. In *Proc. EURICS Workshop on Calculational Theories of Program Structure*, Ameland, The Netherlands, September 1991.

[70] Ed Voermans. *A Relational Theory of Datatypes.* PhD thesis, Eindhoven University of Technology, 1993. In preparation.

[71] Ed Voermans and Jaap van der Woude. A relational theory of datatypes: The per version. Eindhoven University of Technology, January 1991.