The Ruby Interpreter

Graham Hutton
Chalmers University of Technology
Göteborg, Sweden
graham@cs.chalmers.se

November 25, 1994

Abstract

Ruby is a relational language developed by Jones and Sheeran for describing and designing circuits. This document is a guide to the Ruby interpreter, which allows Ruby programs to be executed.

Contents

Chapter 1

Introduction

Ruby is a relational language developed by Jones and Sheeran for describing and designing circuits [6, 3, 4]. Ruby programs denote binary relations, and programs are built-up inductively from primitive relations using a pre-defined set of relational operators. Ruby programs also have a geometric interpretation as networks of primitive relations connected by wires, which is important when layout is considered in circuit design. Ruby has been continually developed since 1986, and has been used to design many different kinds of circuits, including systolic arrays [7], butterfly networks [8] and arithmetic circuits [5].

The Ruby approach to circuit design is to derive implementations from specifications in the following way. We first formulate a Ruby program that clearly expresses the desired relationship between inputs and outputs, but typically has no direct translation as a circuit. We then transform this program using algebraic laws for the relational operators of Ruby, aiming towards a program that does represent a circuit. There are several reasons why Ruby is based upon relations rather than functions. Relational languages offer a rich set of operators and laws for combining and transforming programs, and a natural treatment of non-determinism. Furthermore, many methods for combining circuits (viewed as networks of functions) are unified if the distinction between input and output is removed [6].

This document is a guide to the *Ruby interpreter*, which allows Ruby programs to be executed. 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 `129.16.226.10`) in directories `pub/misc/ruby` and `pub/haskell/chalmers` respectively.

The remainder of this document is structured as follows. Chapter 2 reviews the Ruby language. Chapter 3 defines the class of Ruby programs that the interpreter accepts. Chapter 4 introduces the interpreter by means of a number of worked examples. And finally, chapter 5 gives some reference material.

Chapter 2

Ruby

In this chapter we give a short introduction to Ruby. We don't discuss how Ruby is used to derive programs, or give algebraic laws for the operators of Ruby.

Recall that a (binary) 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 sets $\mathcal{B}$ and $\mathcal{Z}$ of booleans and integers, and closed under finite tupling. If R is a relation, then $a\ R\ b$ means $\langle a, b\rangle \in R$, and the *domain* and *range* of R are the sets defined by $dom(R) = \{a \mid \exists b.\ a\ R\ b\}$ and $rng(R) = \{b \mid \exists a.\ a\ R\ b\}$. We write $f : A \to B$ to mean that f is a total function from set A to set B, and $R : A \leftrightarrow B$ to mean that R is a relation between sets A and B, i.e. that $R \subseteq A \times B$.

Given a function $f : A \to B$, the relation $G(f) : A \leftrightarrow B$ (called the *graph* of f) is defined by $G(f) = \{\langle a, f a\rangle \mid a \in A\}$. In the sequel we mostly leave occurrences of G implicit. For example, id denotes the identity relation $G(id) = \{\langle a, a\rangle \mid a \in \mathcal{U}\}$ (where the function $id : \mathcal{U} \to \mathcal{U}$ is defined by $id(a) = a$ for all $a \in \mathcal{U}$) and $+$ denotes the addition relation $G(+) = \{\langle\langle x, y\rangle, x + y\rangle \mid x, y \in \mathcal{Z}\}$.

The basic operators of Ruby are *composition*, *product* (often called *par*, abbreviating parallel composition), and *converse*, defined as follows:

Definition 1:

$$a\ (R\ ;\ S)\ c \ \Leftrightarrow\ \exists b.\ a\ R\ b\ \wedge\ b\ S\ c,$$

$$\langle a, b\rangle\ [R, S]\ \langle c, d\rangle \ \Leftrightarrow\ a\ R\ c\ \wedge\ b\ S\ d,$$

$$b\ (R^{-1})\ a \ \Leftrightarrow\ a\ R\ b.$$

Par generalises in the obvious way to n-arguments $[R_1, R_2, \ldots, R_n]$. Two common uses of par are abbreviated: fst $R = [R, id]$ and snd $R = [id, R]$.

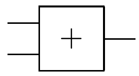
The identity relation id is the simplest example of a *restructuring* or *wiring* relation in Ruby; the other common wiring relations are defined as follows:

Definition 2:

$$\langle a, b\rangle\ swap\ \langle c, d\rangle \ \Leftrightarrow\ a = d\ \wedge\ b = c,$$

$$\langle a, b \rangle \; \pi_1 \; c \;\Leftrightarrow\; a = c,$$
$$\langle a, b \rangle \; \pi_2 \; c \;\Leftrightarrow\; b = c,$$
$$a \; \mathit{fork} \; \langle b, c \rangle \;\Leftrightarrow\; a = b \;\wedge\; a = c,$$
$$\langle \langle a, b \rangle, c \rangle \; \mathit{lsh} \; \langle d, \langle e, f \rangle \rangle \;\Leftrightarrow\; a = d \;\wedge\; b = e \;\wedge\; c = f,$$
$$\langle a, \langle b, c \rangle \rangle \; \mathit{rsh} \; \langle \langle d, e \rangle, f \rangle \;\Leftrightarrow\; a = d \;\wedge\; b = e \;\wedge\; c = f.$$

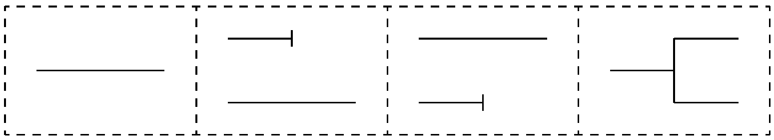
As well as denoting binary relations, Ruby terms have a pictorial interpretation as networks of primitive relations connected by wires. For example, the primitive relation $+$ can be pictured as a single node:



.

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.

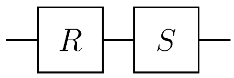
The wiring primitives of Ruby are pictured just as wires. Here are some examples:



id $\quad$ π_1 $\quad$ π_2 $\quad$ fork

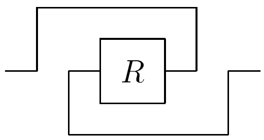
.

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:



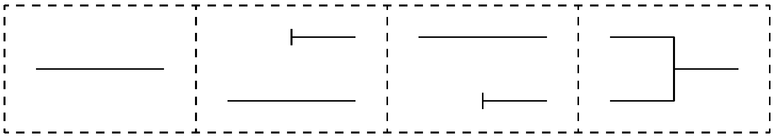
.

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



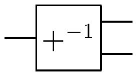
.

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,



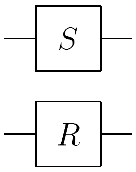
id^{-1} $\qquad$ ${\pi_1}^{-1}$ $\qquad$ ${\pi_2}^{-1}$ $\qquad$ $fork^{-1}$.

Note that 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:



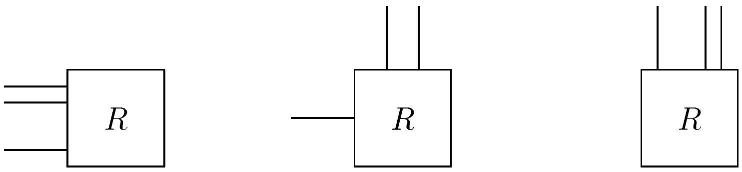
.

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



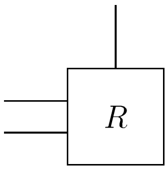
.

Suppose that R is a relation defined by $\langle a, \langle b, c\rangle\rangle \; R \; d \;\Leftrightarrow\; P\langle a, b, c, d\rangle$, where P is some predicate. With what we have said about pictures up till now, the relation R would be drawn as a node 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 node, 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 this relation R:



.

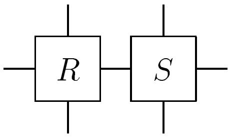
Here is a way that is not acceptable:

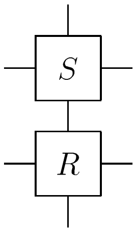


.

This last picture implies that R was defined in the form $\langle\langle a,b\rangle,c\rangle\ R\ d \ \Leftrightarrow\ P\langle a,b,c,d\rangle$, 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 3:

$$\langle a,\langle b,c\rangle\rangle\ R \leftrightarrow S\ \langle\langle d,e\rangle,f\rangle \ \Leftrightarrow\ \exists x.\ \langle a,b\rangle\ R\ \langle d,x\rangle\ \wedge\ \langle x,c\rangle\ S\ \langle e,f\rangle.$$

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

Definition 4: $R \updownarrow S = (R^{-1} \leftrightarrow S^{-1})^{-1}$.

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

Definition 5:

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

For example, $R^4 = R\ ;\ R\ ;\ R\ ;\ R$.

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 6:

$$xs\ rev_n\ ys \ \Leftrightarrow\ \sharp xs = \sharp ys = n\ \wedge\ \forall i < n.\ ys_i = xs_{n-(i+1)},$$

$$\langle x, ys\rangle\ apl_n\ zs \ \Leftrightarrow\ \sharp ys = n\ \wedge\ zs = \langle x\rangle +\!\!+ ys,$$

$$\langle xs, y\rangle\ apr_n\ zs \ \Leftrightarrow\ \sharp xs = n\ \wedge\ zs = xs +\!\!+ \langle y\rangle,$$

$$\langle x, ys\rangle\ distl_n\ zs \ \Leftrightarrow\ \sharp ys = \sharp zs = n\ \wedge\ \forall i < n.\ zs_i = \langle x, ys_i\rangle,$$

$$
\begin{aligned}
&\langle xs, y\rangle\ distr_n\ zs \ \Leftrightarrow\ \sharp xs = \sharp zs = n \ \wedge\ \forall i < n.\ zs_i = \langle xs_i, y\rangle,\\
&\langle xs, ys\rangle\ zip_n\ xs \ \Leftrightarrow\ \sharp xs = \sharp ys = \sharp zs = n \ \wedge\ \forall i < n.\ zs_i = \langle xs_i, ys_i\rangle,\\
&xs\ halve_n\ \langle ys, zs\rangle \ \Leftrightarrow\ \sharp ys = \sharp zs = n \ \wedge\ xs = ys + \!\!+ zs,\\
&xs\ pair_n\ ys \ \Leftrightarrow\ \sharp xs = 2n \ \wedge\ \sharp ys = n \ \wedge\ \forall i < n.\ ys_i = \langle xs_{2i}, xs_{2i+1}\rangle.
\end{aligned}
$$

Here are some examples:

$$
\begin{aligned}
&\langle 1,2,3,4\rangle\ rev_4\ \langle 4,3,2,1\rangle,\\
&\langle 1,\langle 2,3,4\rangle\rangle\ apl_3\ \langle 1,2,3,4\rangle,\\
&\langle\langle 1,2,3\rangle,4\rangle\ apr_3\ \langle 1,2,3,4\rangle,\\
&\langle 1,\langle 2,3,4\rangle\rangle\ distl_3\ \langle\langle 1,2\rangle,\langle 1,3\rangle,\langle 1,4\rangle\rangle,\\
&\langle\langle 1,2,3\rangle,4\rangle\ distr_3\ \langle\langle 1,4\rangle,\langle 2,4\rangle,\langle 3,4\rangle\rangle,\\
&\langle\langle 1,2,3\rangle,\langle 4,5,6\rangle\rangle\ zip_3\ \langle\langle 1,4\rangle,\langle 2,5\rangle,\langle 3,6\rangle\rangle,\\
&\langle 1,2,3,4,5,6\rangle\ halve_3\ \langle\langle 1,2,3\rangle,\langle 4,5,6\rangle\rangle,\\
&\langle 1,2,3,4,5,6\rangle\ pair_3\ \langle\langle 1,2\rangle,\langle 3,4\rangle,\langle 5,6\rangle\rangle.
\end{aligned}
$$

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

Definition 7:

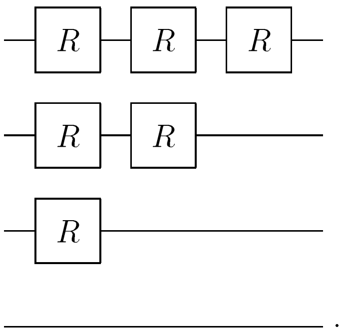
$$
\begin{aligned}
\mathrm{map}_0\ R &= [],\\
\mathrm{map}_{n+1}\ R &= {apl_n}^{-1}\ ;\ [R, \mathrm{map}_n\ R]\ ;\ apl_n.
\end{aligned}
$$

(Note that the $[]$ in the $\mathrm{map}_0\ R$ definition is the 0–width par.) For example, $\mathrm{map}_4\ R = [R,R,R,R]$. Similar to map is the *triangle* construction:

Definition 8:

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

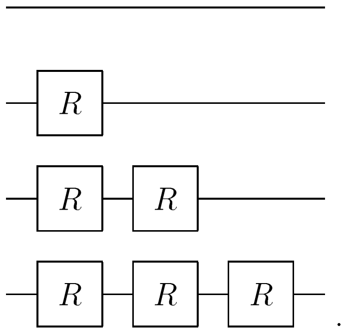
For example, $\mathrm{tri}_4\ R = [R^0, R^1, R^2, R^3]$, or in pictures:



Triangles that grow in the opposite direction are also useful:

Definition 9: $\mathrm{irt}_n\ R = rev_n\ ;\ \mathrm{tri}_n\ R\ ;\ rev_n$.

For example, $\mathrm{irt}_4\ R = [R^3, R^2, R^1, R^0]$, or in pictures:

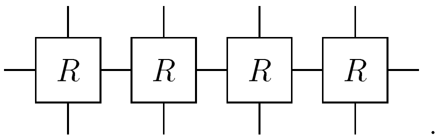


The generic version of $\leftrightarrow$ is the *row* construction:

Definition 10:

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

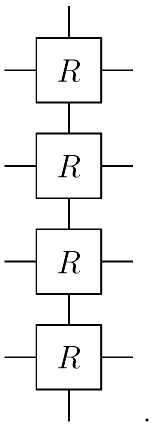
For example, here is a picture of $\mathrm{row}_4\ R$:



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

Definition 11: $\mathrm{col}_n\ R = (\mathrm{row}_n\ R^{-1})^{-1}$.

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



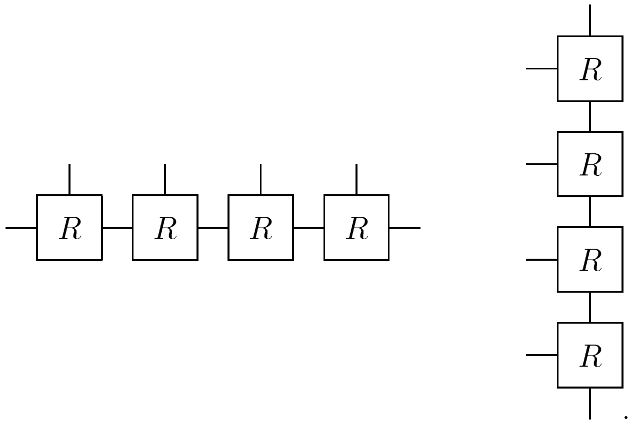
Relational versions of the familiar reduce (or fold) operators from functional programming can be defined in Ruby by using row and col:

Definition 12:

$\text{rdl}_n\ R = \text{row}_n\ (R\ ;\ \mathit{fork})\ ;\ \pi_2,$

$\text{rdr}_n\ R = \text{col}_n\ (R\ ;\ \mathit{fork})\ ;\ \pi_1.$

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



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

$$\langle a, \langle b, c, d, e\rangle\rangle\ \text{rdl}_4\ G(\oplus)\ x \ \Leftrightarrow\ x = (((a \oplus b) \oplus c) \oplus d) \oplus e,$$
$$\langle\langle a, b, c, d\rangle, e\rangle\ \text{rdr}_4\ G(\oplus)\ x \ \Leftrightarrow\ x = a \oplus (b \oplus (c \oplus (d \oplus e))).$$

Many circuits operate with streams of values rather than single values. A stream is modelled in Ruby as a function from the natural numbers $\mathcal{N}$ to the universe $\mathcal{U}$. Given a relation R, the relation $\hat{R}$ on streams is defined by $a\ \hat{R}\ b \ \Leftrightarrow\ \forall t \in \mathcal{N}.\ a(t)\ R\ b(t)$; 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 when working with streams, a unit-delay primitive $\mathcal{D}_s$ (parameterised with a starting value s):

Definition 13: $a\ \mathcal{D}_s\ b \ \Leftrightarrow\ b(0) = s\ \wedge\ \forall t \in \mathcal{N}.\ b(t+1) = a(t).$

Chapter 3

Executable terms

The Ruby interpreter does not accept all Ruby terms, but only those that are *executable*. In this chapter we define the class of executable terms, using the notion of the network of primitive relations denoted by a Ruby term.

> **Definition 14:** 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).

We can define a translation function f from Ruby terms to networks, and semantic functions $[-]_T$ and $[-]_N$ respectively mapping terms and networks to binary relations [2, 1]. The translation is correct with respect to these semantic functions: if t is a term, then $[t]_T = [f(t)]_N$. The reader is referred to [2, 1] for the details. Here we just give a few example translations; see figures 3.1 and 3.2.

Before defining the class of executable networks, we make a number of definitions about wire names. Firstly, 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 for a node $\langle D, P, R \rangle$ where P is a delay primitive is the empty-set $\emptyset$.

> **Definition 15:** A network is *executable* if:
>
> - For every node $\langle D, P, R \rangle$ the relation P is functional;

- For each external wire name x there is at most one node $\langle D, P, R\rangle$ for which x occurs in R, and moreover, x must occur precisely once in this R;
- For each internal wire name x there is precisely one node $\langle D, P, R\rangle$ for which x occurs in R, and moreover, x must occur precisely 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 3.1 and 3.2 give examples of executable and non-executable networks.

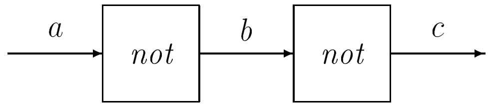
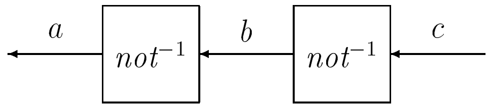
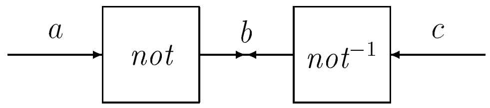
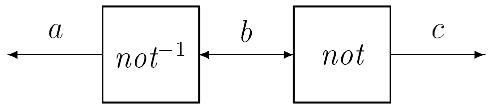
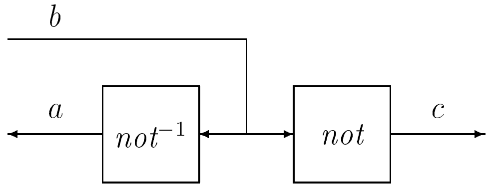
Term	Network	Executable
$not \; ; \; not$	 $\langle\{\langle a, not, b\rangle, \langle b, not, c\rangle\}, a, c\rangle$	yes
$not^{-1} \; ; \; not^{-1}$	 $\langle\{\langle b, not, a\rangle, \langle c, not, b\rangle\}, a, c\rangle$	yes
$not \; ; \; not^{-1}$	 $\langle\{\langle a, not, b\rangle, \langle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

Figure 3.1: Examples of executable and non-executable Ruby terms

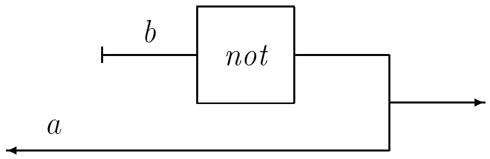
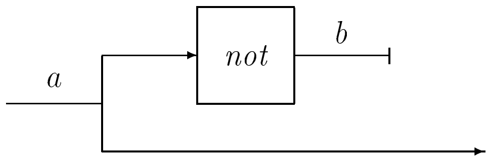
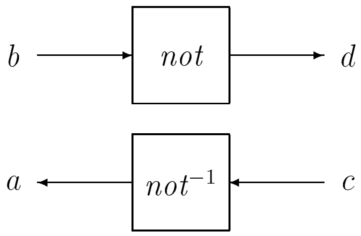
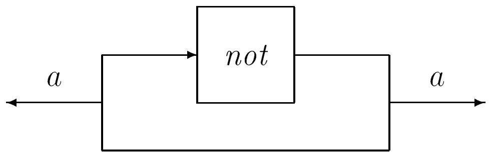
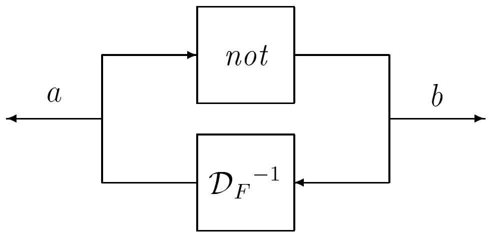
Term	Network	Executable
${\pi_1}^{-1}$; snd not ; $fork^{-1}$	 $\langle\{\langle b, not, a\rangle\}, a, a\rangle$	no (b is undriven)
$fork$; snd not ; π_1	 $\langle\{\langle a, not, b\rangle\}, a, a\rangle$	yes
$[not^{-1}, not]$	 $\langle\{\langle c, not, a\rangle, b, not, d\}, \langle a, b\rangle, \langle c, d\rangle\rangle$	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 3.2: More examples of executable and non-executable Ruby terms

Chapter 4

Worked examples

In this chapter we illustrate the features of the Ruby interpreter by working through a number of simple examples. You should try these out for yourself.

To begin with, load the interactive LML system:

```
graham% lmli

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 load the Ruby interpreter: (you should replace `/u/graham/LML/NewRuby` below with the path for the interpreter as installed at your site)

```
> source "/u/graham/LML/NewRuby/rubysim";

Loading "/u/graham/LML/NewRuby/lml-lib.o"
$set_diff: (List *a)->(List *a)->List *a
split_on: (*a->Bool)->(List *a)->(List *a)#(List *a)
.
.
!!: prog->prog->prog
..: prog->prog->prog

+------------------------------+
|    'The Ruby Interpreter'    |
|      version of 19/5/93      |
|                              |
| Copyright 1993 Graham Hutton |
|    graham@cs.chalmers.se     |
+------------------------------+
```

As shown above, commands typed after the > prompt from the LML system must be ended with a semi-colon. Loading the Ruby interpreter produces a lot of type information; you can ignore all of this.

Ruby programs are compiled using the `rc` function, which translates a program to a network of primitive relations. The function `rc` takes an argument of type `prog`, which is the LML type of Ruby programs. What you need know about the `prog` type is explained in chapter 5. (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 combinational and sequential) 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 3.1 and 3.2:

```
> 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)
```

We 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:

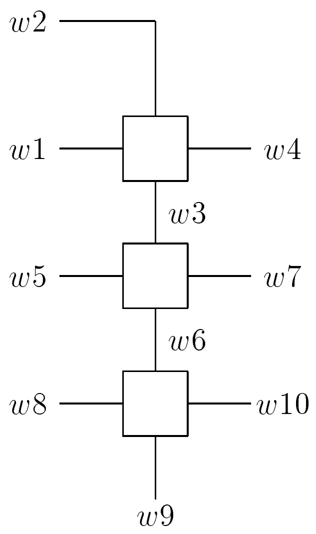
```
> let minim n = inv (apr (n-1)) .. col (n-1) sort2;

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 uses the old definition 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.

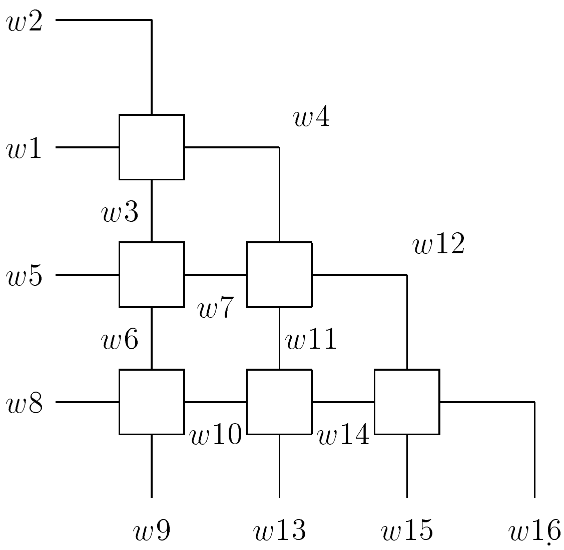
Using minim we can define a sorting program. An n-tuple ($n > 0$) of numbers can be sorted by first selecting the minimum number, and then recursively sorting the remaining ($n-1$)-tuple of numbers. A 1-tuple of numbers requires no sorting, and forms the base-case for the definition:

```
> 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
```

Chapter 5

Reference material

This chapter gives some reference material. Section 5.1 documents the differences between Ruby syntax and the LML syntax for Ruby terms. Section 5.2 defines the built-in logical and arithmetic primitives of the interpreter. Sections 5.3 and 5.4 give the LML definitions for the built-in wiring primitives and combining forms.

5.1 LML syntax for Ruby terms

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`
$\mathrm{map}_n\ r$	`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`.

5.2 Logical and arithmetic primitives

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3 Wiring primitives

```
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)
```

5.4 Combining forms

```
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
```

Bibliography

[1] Carolyn Brown and Graham Hutton. The geometry of relational programs. Chalmers University, February 1993.

[2] Graham Hutton. *Between Functions and Relations in Calculating Programs*. PhD thesis, University of Glasgow, October 1992.

[3] Geraint Jones. Designing circuits by calculation. Technical Report PRG-TR-10-90, Oxford University, April 1990.

[4] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Staunstrup, editor, *Formal Methods for VLSI Design*, Amsterdam, 1990. Elsevier Science Publications.

[5] 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.

[6] Mary Sheeran. Describing and reasoning about circuits using relations. In Tucker et al., editors, *Proc. Workshop in Theoretical Aspects of VLSI*, Leeds, 1986.

[7] Mary Sheeran. Retiming and slowdown in ruby. In Milne, editor, *The Fusion of Hardware Design and Verification*. North-Holland, 1988.

[8] Mary Sheeran. Describing butterfly networks in Ruby. In *Proc. Glasgow Workshop on Functional Programming*, Fraserburgh, 1989. Springer-Verlag.