Implementing Ruby in a Higher-Order Logic Programming Language

Richard McPhee*

April 20, 1995

Abstract

Ruby is a relational language for describing hardware circuits. In the past, programming tools existed which only catered for the execution of functional Ruby expressions rather than the complete set of relational ones. In this paper, we develop an implementation of Ruby in λProlog—a higher-order logic programming language—allowing the execution of arbitrary, relational Ruby programs.

1 Introduction

Programming problems can be tackled by specifying a program's behaviour in an abstract mathematical specification and then, through the application of some appropriate calculus, converting this into an efficient and implementable program. Until recently, the art of deriving computer programs from specification has been performed equationally in a functional calculus [Bir87]. However, it has become evident that a relational calculus affords us a greater degree of expression and flexibility in both specification and proof since a relational calculus naturally captures the notions of non-determinism along with function converses.

Ruby is one such relational language developed by Jones and Sheeran [Jon90, JS90] to describe the behaviour of hardware circuits. However, existing Ruby programming tools only provide for the execution of functional Ruby expressions [LP95, Hut93, Hut92]. In this paper, a purely relational implementation of Ruby is developed in λProlog [LHRB94, NM88]—a higher-order logic programming language—allowing the execution of abstract Ruby circuit specifications. The implementation is purely relational since expressions have a well-defined converse.

The remainder of the paper is organised into four sections. In Section 2, we introduce the Ruby relational language. In Section 3, an implementation of Ruby is developed in λProlog. An example of using the Ruby implementation is presented in Section 4 and we draw several conclusions in Section 5.

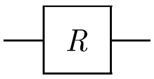
*Author's Address: Richard McPhee, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, U.K. Electronic mail: rm@comlab.ox.ac.uk. Telephone: 01865 273865. Fax: 01865 273839.

2 Ruby

Ruby is a relational language developed by Jones and Sheeran [Jon90, JS90] for describing hardware circuits. The behaviour of a circuit is specified by a relational program which can then be transformed into an equivalent program amenable to implementation in hardware. A circuit in Ruby is represented as a *binary relation* between simple data values. Recall that a binary relation $R : A \to B$, of type A to B, is a subset of the cartesian product $(A \times B)$ of sets A and B. We write $a\ R\ b$ to denote that $(a, b) \in R$.

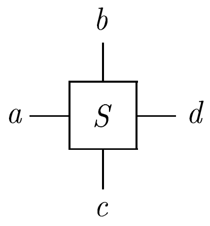
The data values on which these circuits operate consist solely of integers, and both tuples and lists of integers. A tuple is represented in Ruby by the notation (a, b) whose type is $(A \times B)$ for any types A and B, and a list is represented by $[a_0, a_1, \ldots, a_n]$ for each element a_i having type A and the type of the list is *list* A. The empty list is denoted by '[]'. It is worth mentioning that, in this paper, tuples are considered distinct from lists, i.e., a tuple and a list of length two are treated differently. Other descriptions of Ruby [JS91a, Jon90] tend to blur the distinction between tuples and lists, allowing them to be dealt with interchangeably. As a consequence, however, automatic type deduction becomes greatly complicated, so hampering the implementation of Ruby in a strongly typed language, like λProlog.

A relation in Ruby can be given pictorially, corresponding directly to its interpretation as a circuit. For example, the circuit associated with the relation $R : A \to B$ can be represented as the single node



.

The convention regarding data flow in such a circuit is that domain values "enter" at the left and top sides of the node and range values "exit" at the bottom and right sides. In addition, multiple wires to and from a node are read from bottom to top. Therefore, a relation $(a, b)\, S\, (c, d) : (A \times B) \to (C \times D)$—taking tuples to tuples—may be represented by the circuit

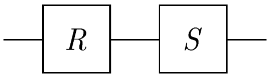


.

Ruby expressions are composed hierarchically from primitive relations and higher-order functions. The most fundamental of these are sequential composition, parallel composition, and converse. The sequential composition of two relations $R : A \to B$ and $S : B \to C$, written as $R\,;S : A \to C$, is given by

$$\begin{aligned} R : A \to B, S : B \to C \quad &\Rightarrow \quad R\,;S : A \to C, \\ a\,(R\,;S)\,c \quad &\equiv \quad \exists b . a\ R\ b \wedge b\ S\ c. \end{aligned}$$

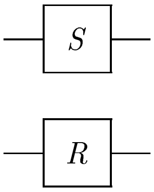
The first line in this expression denotes the type information of the function in a similar style to that adopted by Jones and Sheeran [JS91b]. The sequential composition of two circuits look like this:



The parallel composition of two relations R and S, written as $[R, S]$, is expressed by

$$\begin{aligned} R : A \to B, S : C \to D \;&\Rightarrow\; [R, S] : (A \times C) \to (B \times D), \\ (a, c)\,[R, S]\,(b, d) \;&\equiv\; a\; R\; b \wedge c\; S\; d. \end{aligned}$$

The parallel composition of two circuits simply places them over one another in the following way:



Every relation R has a well-defined converse R^{-1}, given straightforwardly by

$$\begin{aligned} R : A \to B \;&\Rightarrow\; R^{-1} : B \to A, \\ b\; R^{-1}\; a \;&\equiv\; a\; R\; b. \end{aligned}$$

Converse reverses composition in that $(R\, ;\, S)^{-1} = S^{-1}\, ;\, R^{-1}$ and is also an involution since $(R^{-1})^{-1} = R$.

Several primitive relations are used to construct and deconstruct the data values of a circuit. For tuples, we have the relations

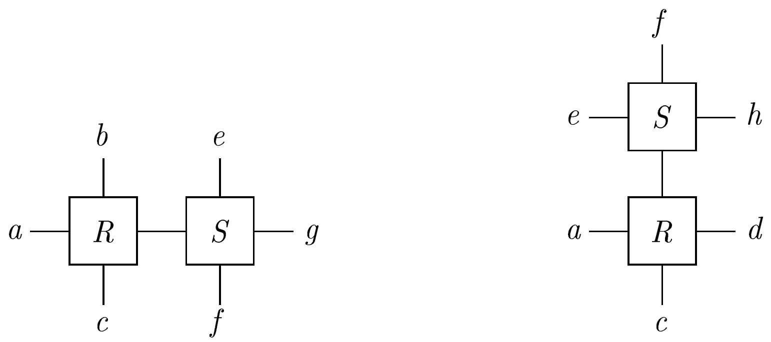
$$\begin{aligned} a\; id\; b \;&\equiv\; a = b, \\ (a, b)\; swap\; (c, d) \;&\equiv\; a = d \wedge b = c, \\ (a, b)\; outl\; c \;&\equiv\; a = c, \\ (a, b)\; outr\; c \;&\equiv\; b = c, \\ a\; fork\; (b, c) \;&\equiv\; a = b \wedge a = c, \\ ((a, b), c)\; lsh\; (d, (e, f)) \;&\equiv\; a = d \wedge b = e \wedge c = f, \\ (a, (b, c))\; rsh\; ((d, e), f) \;&\equiv\; a = d \wedge b = e \wedge c = f. \end{aligned}$$

Of these relations, only *id*, *outl*, and *outr* are essentially primitive in the presence of relational intersection. The intersection of two relations R and S, written as $R \cap S$, is defined by

$$\begin{aligned} R : A \to B, S : A \to B \;&\Rightarrow\; R \cap S : A \to B, \\ a\; (R \cap S)\; b \;&\equiv\; a\; R\; b \wedge a\; S\; b. \end{aligned}$$

The primitive Ruby relations for lists are

$$\begin{aligned} (as, bs)\; app\; cs \;&\equiv\; as +\!\!+\, bs = cs, \\ (a, as)\; apl\; bs \;&\equiv\; [a] +\!\!+\, as = bs, \\ (as, a)\; apr\; bs \;&\equiv\; as +\!\!+\, [a] = bs, \\ a\; wrap\; bs \;&\equiv\; [a] = bs, \\ a\; null\; bs \;&\equiv\; bs = [\,], \end{aligned}$$



$$(a,(b,e))\ R \leftrightarrow S\ ((c,f),g) \qquad\qquad ((a,e),f)\ R \updownarrow S\ (c,(d,h))$$

Figure 1: The circuits corresponding to beside and below.

where '$+\!\!+$' denotes list concatenation. These primitive relations provide us with the bare bones of the Ruby language. The real expressive power is obtained by building higher-order functions—taking relations to relations—from these combining forms. Two commonly used higher-order functions are simple shorthands, given by

$$\begin{aligned} \mathit{fst}\ R &\equiv [R, \mathit{id}], \\ \mathit{snd}\ R &\equiv [\mathit{id}, R]. \end{aligned}$$

Since Ruby is a language for designing circuits, a plethora of functions exist for combining circuit elements. For two circuits, denoted respectively by the relations R and S, the function '$\leftrightarrow$' (pronounced 'beside') joins R and S together in the following fashion:

$$\left.\begin{array}{l} R : (A \times B) \to (C \times D) \\ S : (D \times E) \to (F \times G) \end{array}\right\} \Rightarrow R \leftrightarrow S : (A \times (B \times E)) \to ((C \times F) \times G),$$

$$R \leftrightarrow S \equiv \mathit{rsh}\ ;\ \mathit{fst}\ R\ ;\ \mathit{lsh}\ ;\ \mathit{snd}\ S\ ;\ \mathit{rsh}.$$

The beauty of Ruby is exemplified when we define the *dual* of existing functions in terms of themselves, providing a convenient way to produce new circuits. We can obtain the dual of any higher-order function by first taking the converse of all its arguments and then taking the converse of the entire expression. For example, we can manufacture the function '$\updownarrow$' (pronounced 'below') in terms of its dual function beside by

$$\left.\begin{array}{l} R : (A \times B) \to (C \times D) \\ S : (E \times F) \to (B \times H) \end{array}\right\} \Rightarrow R \updownarrow S : ((A \times E) \times F) \to (C \times (D \times H)),$$

$$R \updownarrow S \equiv (R^{-1} \leftrightarrow S^{-1})^{-1}.$$

Both beside and below, interpreted as circuits, are shown in Figure 1.

Using beside, we can create a new function *row* which forms a linear array, cascading the circuit R a number of times, defined by

$$\begin{aligned} R : (A \times B) \to (C \times A) &\Rightarrow \mathit{row}\ R : (A \times \mathit{list}\ B) \to (\mathit{list}\ C \times A), \\ \mathit{row}\ R &\equiv \mathit{snd}\ \mathit{wrap}^{-1}\ ;\ R\ ;\ \mathit{fst}\ \mathit{wrap} \\ &\quad \vee\ \mathit{snd}\ \mathit{apl}^{-1}\ ;\ (R \leftrightarrow \mathit{row}\ R)\ ;\ \mathit{fst}\ \mathit{apl}. \end{aligned}$$

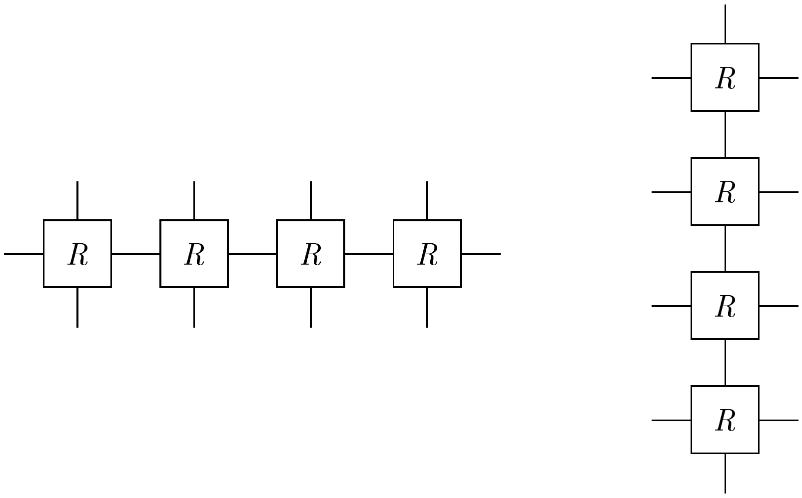


Figure 2: Possible circuits corresponding respectively to *row* R and *col* R.

Notice that *row* is defined as a disjunction; the first clause deals with the case where the second component of the domain tuple is the singleton list since *row* takes non-empty lists. The function *col*, forming a vertical column of circuits, is defined in terms of its dual function *row* via

$$\begin{aligned} R : (A \times B) \to (B \times C) \quad &\Rightarrow \quad col\ R : (list\ A \times B) \to (B \times list\ C), \\ col\ R \quad &\equiv \quad (row\ R^{-1})^{-1}. \end{aligned}$$

The circuits represented by both *row* and *col* are shown in Figure 2.

The *map* function represents repeated parallel composition of the relation R and is defined by

$$\begin{aligned} R : A \to B \quad &\Rightarrow \quad map\ R : list\ A \to list\ B, \\ map\ R \quad &\equiv \quad null^{-1} \, ; null \\ & \quad\ \ \vee\ apl^{-1} \, ; [R, map\ R] \, ; apl. \end{aligned}$$

Finally, the two functions *rdl* and *rdr* relate tuples of values to single values and correspond to the circuit equivalent of "reducing" (or "folding") in functional programming. They are defined as

$$\begin{aligned} R : (A \times B) \to A \quad &\Rightarrow \quad rdl\ R : (A \times list\ B) \to A, \\ rdl\ R \quad &\equiv \quad row\ (R \, ; outr^{-1}) \, ; outr, \end{aligned}$$

and

$$\begin{aligned} S : (A \times B) \to B \quad &\Rightarrow \quad rdr\ S : (list\ A \times B) \to B, \\ rdr\ S \quad &\equiv \quad col\ (S \, ; outl^{-1}) \, ; outl, \end{aligned}$$

respectively. The circuits corresponding to these functions are illustrated in Figure 3.

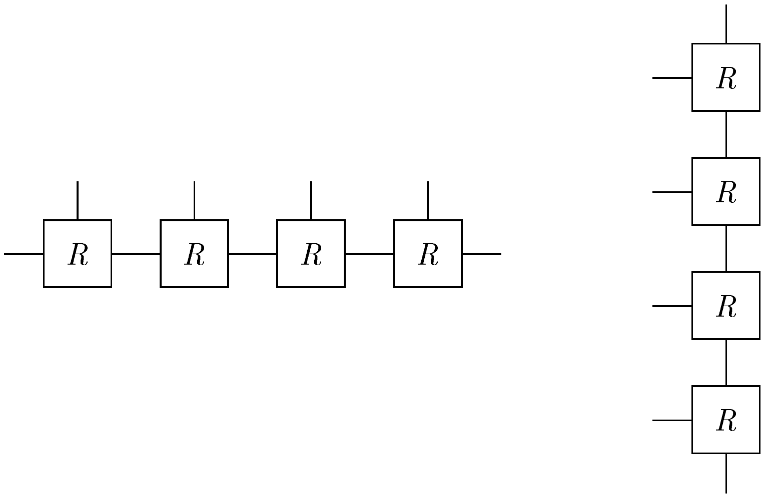


Figure 3: An illustration of *rdl R* and *rdr R*.

3 Programming Ruby in λProlog

The higher-order logic language λProlog, developed by Nadathur and Miller [NM88], naturally supports higher-order programming features in a manner consistent with its underlying logical foundation. In what follows, we examine how to implement Ruby primitives in λProlog [LHRB94]. The syntax of λProlog is introduced informally as the section progresses.

We can define Ruby's tuple data value in λProlog via the declaration

$$\begin{array}{lll} \text{kind} & \textit{tuple} & \text{type} \to \text{type} \to \text{type}. \\ \text{type} & : & A \to B \to (\textit{tuple}\ A\ B). \end{array}$$

The 'kind' operator creates a new type which is simply a type constructor of arity one less than the number of occurrences of the keyword 'type' in the declaration. In the case above, the type constructor *tuple* is declared taking two arguments to this new type. Function and predicate symbols are introduced via a 'type' declaration. The infix function symbol ':' declared above takes two arbitrary types, A and B, and returns a value of type *tuple* A B.

Notice that λProlog is polymorphically typed and is curried in the same sense as in many functional programming languages, like Haskell, except that type signatures must be given explicitly for all function and predicate symbols. Type variables in λProlog are denoted by identifiers beginning with an upper case letter. The type of propositions (or "truth values") is the special type 'o' and so the type signature of a predicate in λProlog terminates with this type. A small λProlog program defining "Peano" natural numbers, given in Figure 4, demonstrates some of these syntactic points. Natural numbers are thus represented by *zero*, (*succ zero*), (*succ* (*succ zero*)), and so forth.

We make use of the built-in *list* type in λProlog in order to implement the primitive

$$
\begin{array}{lll}
\text{kind} & \mathit{nat} & \text{type.} \\
\text{type} & \mathit{zero} & \mathit{nat}. \\
\text{type} & \mathit{succ} & \mathit{nat} \to \mathit{nat}. \\
\text{type} & \mathit{add} & (\mathit{tuple}\ \mathit{nat}\ \mathit{nat}) \to \mathit{nat} \to o.
\end{array}
$$

$$
\begin{array}{l}
\mathit{add}\ (M : \mathit{zero})\ M. \\
\mathit{add}\ (M : \mathit{succ}\ N)\ (\mathit{succ}\ Y) \ \leftarrow \ \mathit{add}\ (M : N)\ Y.
\end{array}
$$

Figure 4: The implementation of Peano natural numbers in λProlog.

relations pertaining to the Ruby list data value. Lists in λProlog are declared as

$$
\begin{array}{lll}
\text{kind} & \mathit{list} & \text{type} \to \text{type.} \\
\text{type} & \mathit{nil} & \mathit{list}\ A. \\
\text{type} & :: & A \to (\mathit{list}\ A) \to (\mathit{list}\ A).,
\end{array}
$$

where *nil* denotes the empty list and '::' denotes 'cons'. The λProlog predicate for list concatenation is *append* and the basic Ruby relations can now be implemented as shown in Figure 5.

We represent the Ruby primitives of parallel composition and converse by the respective predicates

$$
\begin{array}{lll}
\text{type} & \mathit{par} & (A \to B \to o) \to (C \to D \to o) \to (\mathit{tuple}\ A\ C) \to (\mathit{tuple}\ B\ D) \to o. \\
\text{type} & \mathit{conv} & (A \to B \to o) \to B \to A \to o.
\end{array}
$$

$$
\mathit{par}\ R\ S\ (A : C)\ (B : D) \ \leftarrow \ R\ A\ B, S\ C\ D.
$$

$$
\mathit{conv}\ R\ B\ A \ \leftarrow \ R\ A\ B.
$$

Several syntactic points are illustrated in this example. We can see from the type declarations of *par* and *conv* that they are both higher-order predicates taking other predicates as arguments. Predicates in λProlog may be defined as "infix" and are also *curried* facilitating substantial notational convenience. For instance, $(R\ \mathit{par}\ (\mathit{conv}\ S))$ denotes a valid predicate in itself and can be passed as an argument to other higher-order predicates. Therefore, we can construct goals from predicates which take arbitrarily complex relations as arguments.

The conspicuous absence of the definition of sequential composition is deliberate for a technical reason which we now spend a moment to resolve. The implementation of Ruby we present here unfortunately depends critically on the use of an extra-logical predicate *flex* which determines whether or not a logical variable is instantiated. The reason for using this predicate—a higher-order analogy to the predicate *var* found in Prolog [CM87]—is to overcome a standard problem encountered in most logic programming languages: As a consequence of the left to right evaluation strategy adopted in logic programming language implementations, whereby the leftmost unsolved goal is selected for

$$
\begin{array}{lll}
\text{type} & \mathit{id} & A \to A \to o. \\
\text{type} & \mathit{swap} & (\mathit{tuple}\ A\ B) \to (\mathit{tuple}\ B\ A) \to o. \\
\text{type} & \mathit{outl} & (\mathit{tuple}\ A\ B) \to A \to o. \\
\text{type} & \mathit{outr} & (\mathit{tuple}\ A\ B) \to B \to o. \\
\text{type} & \mathit{fork} & A \to (\mathit{tuple}\ A\ A) \to o. \\
\text{type} & \mathit{lsh} & (\mathit{tuple}\ (\mathit{tuple}\ A\ B)\ C) \to (\mathit{tuple}\ A\ (\mathit{tuple}\ B\ C)) \to o. \\
\text{type} & \mathit{rsh} & (\mathit{tuple}\ A\ (\mathit{tuple}\ B\ C)) \to (\mathit{tuple}\ (\mathit{tuple}\ A\ B)\ C) \to o. \\
\text{type} & \mathit{app} & (\mathit{tuple}\ (\mathit{list}\ A)\ (\mathit{list}\ A)) \to (\mathit{list}\ A) \to o. \\
\text{type} & \mathit{apl} & (\mathit{tuple}\ A\ (\mathit{list}\ A)) \to (\mathit{list}\ A) \to o. \\
\text{type} & \mathit{apr} & (\mathit{tuple}\ (\mathit{list}\ A)\ A) \to (\mathit{list}\ A) \to o. \\
\text{type} & \mathit{wrap} & A \to (\mathit{list}\ A) \to o. \\
\text{type} & \mathit{null} & A \to (\mathit{list}\ B) \to o.
\end{array}
$$

$\mathit{id}\ A\ A.$

$\mathit{swap}\ (A : B)\ (B : A).$

$\mathit{outl}\ (A : B)\ A.$

$\mathit{outr}\ (A : B)\ B.$

$\mathit{fork}\ A\ (A : A).$

$\mathit{lsh}\ ((A : B) : C)\ (A : (B : C)).$

$\mathit{rsh}\ (A : (B : C))\ ((A : B) : C).$

$\mathit{apl}\ (A : AS)\ BS \quad \leftarrow \quad \mathit{append}\ (A :: \mathit{nil})\ AS\ BS.$

$\mathit{apr}\ (AS : A)\ BS \quad \leftarrow \quad \mathit{append}\ AS\ (A :: \mathit{nil})\ BS.$

$\mathit{app}\ (A : B)\ C \quad \leftarrow \quad \mathit{append}\ A\ B\ C.$

$\mathit{wrap}\ A\ (A :: \mathit{nil}).$

$\mathit{null}\ A\ \mathit{nil}.$

Figure 5: The implementation of the most primitive Ruby relations in λProlog.

resolution, certain queries can result in infinite computation. A simple example which demonstrates this undesirable property is the query

$$?- \quad add\,(X : Y)\,Z, add\,(X : Y)\,(succ\ succ\ zero).,$$

assuming the previous definition of the predicate *add*. Since the leftmost subgoal $add\,(X : Y)\,Z$ can be satisfied in an infinite number of ways, the whole query results in non-termination in λProlog when all possible solutions are requested. The situation is, in theory, rectified by simply swapping the two subgoals in the conjunction. The problem can essentially be reduced to the fact that conjunction is not commutative in λProlog.

This evaluation problem particularly affects an implementation of Ruby in λProlog since we desire the ability to run queries "in reverse," i.e., solving for logical variables in either the range or domain positions of a relation. By doing so, we satisfy the requirement that each relation in Ruby has a well defined converse. However, the fundamental Ruby primitive of sequential composition is directly affected by this problem. To see why, consider a naïve, infix implementation of sequential composition in λProlog:

$$\text{type} \quad comp' \quad (A \to B \to o) \to (B \to C \to o) \to A \to C \to o.$$

$$comp'\ R\ S\ A\ C \quad \leftarrow \quad \text{sigma}\, B\backslash\,(R\ A\ B, S\ B\ C).$$

The λProlog primitive 'sigma' denotes existential quantification and the operator '\' denotes infix λ-abstraction, providing a way to create predicates. Then the query

$$?- \quad add\ comp'\ (conv\ add)\ A\ (succ\ zero : succ\ zero).,$$

decomposes into the expression

$$?- \quad \text{sigma}\, B\backslash\,(add\ A\ B, add\,(succ\ zero : succ\ zero)\ B).,$$

which is exactly the problematic situation we wish to avoid.

The solution is straightforward although admittedly less declarative than one might prefer. In the definition of sequential composition, we check whether the domain parameter is uninstantiated. If it is, we resolve the clauses in the composition in the reverse order to avoid slipping into an infinite computation. This results in a revised implementation of composition, given in Figure 6. The appearance of the cut '!' in the first clause of *comp* simply avoids the second clause being attempted should the first one fail. Another point to note is that some type obscuring takes place simply to overcome the fact that a Ruby value comprises of either a number, or a tuple or list of numbers. An alternative method for tackling this difficulty would be to attach an explicit type constructor to each Ruby value to differentiate between them.

Now that we have implemented *comp*, we can easily implement the other Ruby higher-order functions, as illustrated in Figure 7. An attractive aspect of using λProlog to implement Ruby is the ease with which Ruby operators translate into λProlog syntax; the higher-order functions translate almost identically from their abstract Ruby definitions to their corresponding implementations in λProlog. In the next section, we look at a simple example of how to create new Ruby relations in λProlog.

$$
\begin{array}{lll}
\text{type} & \mathit{comp} & (A \to B \to o) \to (B \to C \to o) \to A \to C \to o. \\[1ex]
\text{kind} & \mathit{anonymous} & \text{type.} \\
\text{type} & \mathit{untype} & A \to \mathit{anonymous}. \\
\text{type} & \mathit{uninstantiated} & A \to o. \\
\text{type} & \mathit{flexible} & \mathit{anonymous} \to o.
\end{array}
$$

$$
\begin{array}{lll}
\mathit{comp}\ R\ S\ A\ C & \leftarrow & \mathit{uninstantiated}\ A, !, \text{sigma}\ B\backslash\ (S\ B\ C, R\ A\ B). \\
\mathit{comp}\ R\ S\ A\ C & \leftarrow & \text{sigma}\ B\backslash\ (R\ A\ B, S\ B\ C).
\end{array}
$$

$$
\mathit{uninstantiated}\ A \ \leftarrow\ \mathit{flexible}\ (\mathit{untype}\ A).
$$

$$
\begin{array}{lll}
\mathit{flexible}\ (\mathit{untype}\ A) & \leftarrow & \mathit{flex}\ A, !. \\
\mathit{flexible}\ (\mathit{untype}\ (A : B)) & \leftarrow & \mathit{flexible}\ (\mathit{untype}\ A), \mathit{flexible}\ (\mathit{untype}\ B). \\
\mathit{flexible}\ (\mathit{untype}\ (A :: \mathit{nil})) & \leftarrow & \mathit{flexible}\ (\mathit{untype}\ A). \\
\mathit{flexible}\ (\mathit{untype}\ (A :: AS)) & \leftarrow & \mathit{flexible}\ (\mathit{untype}\ A), \mathit{flexible}\ (\mathit{untype}\ AS).
\end{array}
$$

Figure 6: The implementation of sequential composition *comp* in λProlog.

4 A Simple Sorting Circuit in Ruby

In this section, we present a circuit which sorts an arbitrary, non-empty list of natural numbers. We noted earlier that we require the ability to solve logic variables in both the range and domain positions of a relation, so preserving the converse of each Ruby relation. However, the arithmetic primitives of λProlog are unable to support such a flexible computational requirement. As a consequence, we utilise the alternative definition of "Peano" natural numbers introduced in the previous section.

Two alternative methods for describing the sorting circuit have been given in the past, one by Sheeran and Jones [SJ87] and the other by Hutton [Hut92]. We implement the former presentation in this section. The sorting circuit is illustrated in Figure 8. Each node *cmp* is a comparison operation—taking tuples to sorted tuples—and we can see from the figure that some kind of reduction is being performed with a column of comparisons. In fact, the reduction is the converse of a reduce right, rdr^{-1}. In Ruby notation, we can define the sorting circuit *sort* as

$$
\begin{array}{lll}
\mathit{sort} : \mathit{list\ nat} \to \mathit{list\ nat}, & & \\
\mathit{sort} & \equiv & \mathit{wrap}^{-1}\ ;\ \mathit{wrap} \\
 & & \vee\ \mathit{rdr}^{-1}\ (\mathit{apr}^{-1}\ ;\ \mathit{col\ cmp})^{-1}\ ;\ \mathit{snd\ wrap}^{-1}\ ;\ \mathit{apr}.
\end{array}
$$

The translation of this expression into λProlog is shown in Figure 9 including a suitable definition of *cmp*. We can then query

$$
?-\quad \mathit{sort}\ (\mathit{succ\ succ\ zero} :: \mathit{zero} :: \mathit{succ\ succ\ succ\ zero} :: \mathit{succ\ zero} :: \mathit{nil})\ B.,
$$

$$
\begin{array}{lll}
\text{type} & \mathit{map} & (A \to B \to o) \to (\mathit{list}\ A) \to (\mathit{list}\ B) \to o. \\
\text{type} & \mathit{beside} & ((\mathit{tuple}\ A\ B) \to (\mathit{tuple}\ C\ D) \to o) \to \\
 & & ((\mathit{tuple}\ D\ E) \to (\mathit{tuple}\ F\ G) \to o) \to \\
 & & (\mathit{tuple}\ A\ (\mathit{tuple}\ B\ E)) \to (\mathit{tuple}\ (\mathit{tuple}\ C\ F)\ G) \to o. \\
\text{type} & \mathit{below} & ((\mathit{tuple}\ A\ B) \to (\mathit{tuple}\ C\ D) \to o) \to \\
 & & ((\mathit{tuple}\ E\ F) \to (\mathit{tuple}\ B\ H) \to o) \to \\
 & & (\mathit{tuple}\ (\mathit{tuple}\ A\ E)\ F) \to (\mathit{tuple}\ C\ (\mathit{tuple}\ D\ H)) \to o. \\
\text{type} & \mathit{row} & ((\mathit{tuple}\ A\ B) \to (\mathit{tuple}\ C\ A) \to o) \to \\
 & & (\mathit{tuple}\ A\ (\mathit{list}\ B)) \to (\mathit{tuple}\ (\mathit{list}\ C)\ A) \to o. \\
\text{type} & \mathit{col} & ((\mathit{tuple}\ A\ B) \to (\mathit{tuple}\ B\ C) \to o) \to \\
 & & (\mathit{tuple}\ (\mathit{list}\ A)\ B) \to (\mathit{tuple}\ B\ (\mathit{list}\ C)) \to o. \\
\text{type} & \mathit{rdl} & ((\mathit{tuple}\ A\ B) \to A \to o) \to ((\mathit{tuple}\ A\ (\mathit{list}\ B))) \to A \to o. \\
\text{type} & \mathit{rdr} & ((\mathit{tuple}\ A\ B) \to B \to o) \to ((\mathit{tuple}\ (\mathit{list}\ A)\ B)) \to B \to o.
\end{array}
$$

$$
\begin{array}{lll}
\mathit{map}\ R\ A\ B & \leftarrow & (\mathit{conv}\ \mathit{apl})\ \mathit{comp}\ (\mathit{par}\ R\ (\mathit{map}\ R))\ \mathit{comp}\ \mathit{apl}\ A\ B. \\
\mathit{map}\ R\ A\ B & \leftarrow & (\mathit{conv}\ \mathit{null})\ \mathit{comp}\ \mathit{null}\ A\ B.
\end{array}
$$

$$\mathit{beside}\ R\ S\ A\ B \ \leftarrow\ \mathit{rsh}\ \mathit{comp}\ (\mathit{fst}\ R)\ \mathit{comp}\ \mathit{lsh}\ \mathit{comp}\ (\mathit{snd}\ S)\ \mathit{comp}\ \mathit{rsh}\ A\ B.$$

$$\mathit{below}\ R\ S\ A\ B \ \leftarrow\ \mathit{conv}\ ((\mathit{conv}\ R)\ \mathit{beside}\ (\mathit{conv}\ S))\ A\ B.$$

$$
\begin{array}{lll}
\mathit{row}\ R\ A\ B & \leftarrow & (\mathit{snd}\ (\mathit{conv}\ \mathit{apl}))\ \mathit{comp}\ (R\ \mathit{beside}\ (\mathit{row}\ R))\ \mathit{comp}\ (\mathit{fst}\ \mathit{apl})\ A\ B. \\
\mathit{row}\ R\ A\ B & \leftarrow & (\mathit{snd}\ (\mathit{conv}\ \mathit{wrap}))\ \mathit{comp}\ R\ \mathit{comp}\ (\mathit{fst}\ \mathit{wrap})\ A\ B.
\end{array}
$$

$$\mathit{col}\ R\ A\ B \ \leftarrow\ \mathit{conv}\ (\mathit{row}\ (\mathit{conv}\ R))\ A\ B.$$

$$\mathit{rdl}\ R\ A\ B \ \leftarrow\ (\mathit{row}\ (R\ \mathit{comp}\ (\mathit{conv}\ \mathit{outr})))\ \mathit{comp}\ \mathit{outr}\ A\ B.$$

$$\mathit{rdr}\ R\ A\ B \ \leftarrow\ (\mathit{col}\ (R\ \mathit{comp}\ (\mathit{conv}\ \mathit{outl})))\ \mathit{comp}\ \mathit{outl}\ A\ B.$$

Figure 7: The implementation of Ruby higher-order functions in λProlog.

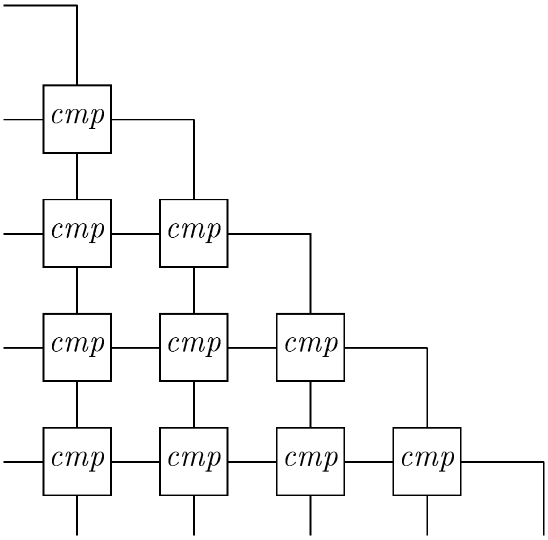


Figure 8: A possible circuit corresponding to *sort*.

$$
\begin{array}{lll}
\text{type} & leq & nat \to nat \to o. \\
\text{type} & min & (tuple\ nat\ nat) \to nat \to o. \\
\text{type} & max & (tuple\ nat\ nat) \to nat \to o. \\
\text{type} & cmp & (tuple\ nat\ nat) \to (tuple\ nat\ nat) \to o. \\
\text{type} & sort & (list\ nat) \to (list\ nat) \to o.
\end{array}
$$

$leq\ zero\ M.$

$leq\ (succ\ M)\ (succ\ N) \quad \leftarrow \quad leq\ M\ N.$

$min\ (M : N)\ M \quad \leftarrow \quad leq\ M\ N.$

$min\ (M : N)\ N \quad \leftarrow \quad leq\ N\ M.$

$max\ (M : N)\ M \quad \leftarrow \quad leq\ N\ M.$

$max\ (M : N)\ N \quad \leftarrow \quad leq\ M\ N.$

$cmp\ A\ B \quad \leftarrow \quad fork\ comp\ (min\ par\ max)\ A\ B.$

$sort\ A\ B \quad \leftarrow \quad (conv\ (rdr\ (conv\ ((conv\ apr)\ comp\ (col\ cmp)))))\ comp$
$\qquad\qquad (snd\ (conv\ wrap))\ comp\ apr\ A\ B.$

$sort\ A\ B \quad \leftarrow \quad (conv\ wrap)\ comp\ wrap\ A\ B.$

Figure 9: The implementation of *sort* in λProlog.

to obtain the result $B = (zero :: succ\ zero :: succ\ succ\ zero :: succ\ succ\ succ\ zero :: nil)$. We encounter a more interesting example by solving the converse of *sort*. The query

$$?- \quad conv\ sort\ (zero :: succ\ zero :: succ\ succ\ zero :: succ\ succ\ succ\ zero :: nil)\ B.,$$

computes every permutation of the given list, corresponding to all the lists which, when sorted, result in the given list.

5 Conclusions

In this paper, a relational implementation of Ruby[1] was developed in λProlog allowing the execution of abstract relational Ruby specifications for the first time. Moreover, the translation of a Ruby specification to its concrete representation in λProlog is natural and straightforward. The implementation of relations presented here demonstrates a previously undocumented use of λProlog, perhaps suggesting a new area for the practical application of higher-order logic programming.

The left to right computational strategy employed in λProlog does, unfortunately, hamper a completely declarative solution since we artificially affect the flow of control in a Ruby program. However, this indicates more of a problem with logic programming languages in general rather than with the Ruby implementation in particular.

As a consequence of this problem, no correctness results are given for the Ruby implementation. It would appear, though, that creating new Ruby relations using only the primitive set presented in this paper suggests that the resulting program should execute correctly.

One of the most interesting aspects of the Ruby implementation is that each relation has a well-defined converse which greatly eases the definition of some other relations. The value of the Ruby implementation presented here is not one of circuit synthesis [LP95, Hut92] but rather of the effective demonstration that a relational calculus can be programmed in λProlog. At this stage, it remains unclear, but certainly an intriguing question, whether higher-order logic programming languages can be used to implement more expressive relational languages [BdM94, ABH+92, Möl92, SS88] than Ruby, particularly given the dependence of this implementation on an extra-logical feature.

Acknowledgements

I would like to extend my sincere thanks to Oege de Moor. Also, I would like to thank Geraint Jones, Graham Hutton, and Wayne Luk for their useful comments. I gratefully acknowledge the financial support of the Engineering and Physical Sciences Research Council of the United Kingdom.

[1] The λProlog implementation of Ruby is available via anonymous ftp from the site

`ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.McPhee/ruby`

References

[ABH$^+$92] C. J. Aarts, R. C. Backhouse, P. Hoogendijk, E. Voermans, and J. C. S. P. Van der Woude. A relational theory of datatypes. Available via anonymous ftp from `ftp.win.tue.nl` in directory `pub/math.prog.construction`, September 1992.

[BdM94] R. S. Bird and O. de Moor. Relational program derivation and context-free language recognition. In A.W. Roscoe, editor, *A Classical Mind: Essays dedicated to C.A.R. Hoare*. Prentice Hall International, 1994.

[Bir87] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 3–42. Springer–Verlag, 1987.

[CM87] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer-Verlag, third edition, 1987.

[Hut92] G. Hutton. *Between Functions and Relations in Calculating Programs*. Research report, Department of Computer Science, Glasgow University, 1992.

[Hut93] G. Hutton. *The Ruby Interpreter*. Chalmers University of Technology, Sweden, 1993.

[Jon90] G. Jones. Designing circuits by calculation. Technical Report PRG–TR–10–90, Programming Research Group, 11 Keble Road, Oxford OX1 3QD, England, 1990.

[JS90] G. Jones and M. Sheeran. Circuit design in Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*, pages 13–70. Elsevier Science Publications, 1990.

[JS91a] G. Jones and M. Sheeran. Collecting butterflies. Technical Monograph PRG-91, University of Oxford, Programming Research Group, 1991.

[JS91b] G. Jones and M. Sheeran. Relations and refinement in circuit design. In J. Woodcock and C. Morgan, editors, *Third Refinement Workshop, 1990*. Springer Workshops in Computing, 1991.

[LHRB94] S. Le Huitouze, O. Ridoux, and P. Brisset. Prolog/Mali reference manual. Available via anonymous ftp from `ftp://ftp.irisa.fr/local/lande/pm`, 1994.

[LP95] W. Luk and C. Pitcher. *Simulation Facilities in Rebecca*. Imperial College of Science, Technology, and Medicine, London, U.K., 1995.

[Möl92] B. Möller. Derivation of graph and pointer algorithms. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development, Proceedings of the IFIP TC2/WG2.1 State of the Art Seminar*. Springer, 1992.

[NM88] G. Nadathur and D. Miller. An overview of λProlog. In R. Kowalski and K. A. Bowen, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, U.S.A., 1988. MIT Press.

[SJ87] M. Sheeran and G. Jones. Relations + higher-order functions = hardware descriptions. In W. E. Proebster and H. Reiner, editors, *Proceedings of the IEEE Comp Euro 87: VLSI and Computers*, pages 303–306, Hamburg, 1987.

[SS88] G. Schmidt and T. Ströhlein. *Relationen und Grafen*. Springer-Verlag, 1988.