# Compositional Logic Programming

Richard McPhee Oege de Moor

Programming Research Group
Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD, U.K.

## 1 Introduction

Prototyping specifications remains an important application area of declarative programming languages. The logic paradigm allows direct execution of very high-level specifications whilst the functional paradigm requires a specification to be phrased more algorithmically due to the distinction between input and output. However, functional languages provide more support for *composing* specifications.

Through the use of higher-order operators, functional programs can be split up into small, mind-sized pieces. One can define *combinators* that capture common idioms of program composition. Apart from the aesthetic benefits and the increased readability of programs, employing combinators encourages the formulation of general mathematical results about programs [Bac85]. In turn, such results facilitate concise verification of the equivalence of programs. Also, because combinators are defined by the user, they can be tailored to specific applications: for example, to parsing [Hut92b], pretty-printing [Hug95], graphical user interfaces [HC95], or capturing patterns of parallel computation [DFH$^+$93]. But there remains the problem that functional languages are less expressive than logic languages: in particular, they lack nondeterminism and program converses.

It is for this reason that various researchers from the functional programming community have, in recent years, investigated the generalisation from functional to relational programming. Here, all the combinators that have proven their worth in functional programming are redefined to operate on binary relations, rather than on functions [BVK95, BdM94, BH93, JS90]. However, this work has been mostly methodological in nature: the aim being to design calculi for the *derivation* of functional programs from relational specifications. As a result, the execution of relational specifications has received rather little attention, although [Hut92a] is one notable exception.

Of course, the logic programming community has also recognised the benefits of compositionality. The issue has been addressed in the design of functional logic languages [Han94, MNRA92] and higher-order logic languages [NM88]. Moreover, this work focuses on executable languages and even addresses questions of efficient execution. Despite this, such languages are still not as compositional as we (coming from the functional community) would like. The main problem stems from the use of the left-selection rule, effectively banning the use of a composition operator and program converses unless extra-logical annotations are added to the program. In the case of higher-order logic programming, we also have the complication that higher-order unification is undecidable.

This paper is intended to stimulate discussion on the design of a truly compositional logic language, which is at the same time as simple as possible. We illustrate our wishful thinking by two substantial examples: a set of combinators that serves both to build parsers and pretty-printers simultaneously, and an implementation of the relational hardware-description language Ruby. The primary contribution of this paper is this collection of examples which, in our view, demonstrates the desirability of a compositional logic language.

We then go on to show that our wishes are not entirely detached from reality, by sketching an implementation of the language. The main feature that makes the language compositional is the use of a fair selection rule [LM84]: this guarantees that (for the generation of all solutions) programs terminate whenever possible. The implementation is, however, grossly inefficient and we hope to benefit by interacting with those more experienced in the implementation of logic languages.

## 2 Combinator Parsing and Pretty-Printing

In functional programming, the preferred method of writing parsers is through the use of *parsing combinators* [Hut92b]. The reasons for this preferential position are the speed of creating such parsers, the simplicity of their structure, and the ease with which the parsers can be modified. Combinator parsers in functional languages make essential use of compositionality and higher-order programming. Ambiguous grammars are dealt with by appealing to the classic "list of successes" [Wad85] approach, which encapsulates a form of backtracking. Traditional logic programming languages are, for the most part, unsuitable for a combinatory approach to programming since features like currying and higher-order programming are not generally supported.

In this example, it is shown how our compositional logic programming language proves adequate for implementing combinator parsers. As we shall see consequently, our parsers are also more symmetrical than their functional counterparts; later, we choose to run our parsers "backwards"—taking full advantage of the expressive power of the logic language—thus producing a pretty printer for our grammar without expending any additional programming effort.

### Primitive Parsers

The structural simplicity of combinator parsers stems from the hierarchical nature in which they are derived; from a discrete set of primitive parsing constructs, we can build elaborate parsers for sophisticated grammars. Each parser is represented by a predicate in the logic program. The "input" to our parsers is a list of characters and the "output" consists of both the converted part of the input string, called the *value*, and the unconsumed remainder of the input. A pair is represented by `(a,b)` in the language and lists are constructed by the symbols `nil`, which denotes the empty list, and '`:`', which denotes infix cons.

The first primitive parser is

```
succeed V Xs V Xs :- .
```

that consumes no input and always succeeds with some predetermined value. In a logic programming language, we do not require a parser that always fails; we simply inherit the failure of a parser from the lack of a proof for a parse, i.e., if a parser fails then the interpreter simply indicates `no`. The parser

```
satisfy P (X:Xs) V Ys :- P X, succeed X Xs V Ys.
```

provides a good illustration of this phenomenon since any `X` that does not satisfy the predicate `P` will result in failure upon backtracking.

Another primitive parser

```
literal X Xs V Ys :- satisfy (eq X) Xs V Ys.
```

succeeds when the next character in the input string matches `X`. The predicate `eq`, defined as `eq X X :- .`, determines the equality of two terms. The curried use of `eq` above demonstrates a particularly convenient syntactic notation corresponding to that found in functional programming.

### Parsing combinators

One of the aims of combinator parsing is to facilitate an easy translation from a BNF description of a grammar to its parser. To achieve this, we provide a number of higher-order predicates,

```
(P1 $or P2) Xs V Ys :- P1 Xs V Ys.
(P1 $or P2) Xs V Ys :- P2 Xs V Ys.

(P1 $then P2) Xs (V,W) Ys :- P1 Xs V Zs, P2 Zs W Ys.

(P $using F) Xs V Ys :- P Xs W Ys, F W V.

many P Xs V Ys :- (((P $then (many P)) $using cons) $or
                   (succeed nil)) Xs V Ys.

some P Xs V Ys :- ((P $then (many P)) $using cons) Xs V Ys.

(P1 $xthen P2) Xs V Ys :- ((P1 $then P2) $using snd) Xs V Ys.

(P1 $thenx P2) Xs V Ys :- ((P1 $then P2) $using fst) Xs V Ys.
```

Figure 1: Combinators for parsing.

taking parsers to parsers, that correspond to the various constructs of BNF. For instance, we can define the choice operator of BNF as the higher-order predicate `or`, shown in Figure 1. (The prefix operator '`$`' allows a predicate to be written infix style.) Additionally, we define the sequential composition of two parsers using the predicate `then` (Figure 1).

To build useful parsers, we provide for the direct manipulation of values returned by a parser. For example, as we proceed with a parse we could envisage the construction of an abstract syntax tree or some immediate semantic evaluation of the parsed expressions. The parser `P $using F` performs this task by applying the arbitrary predicate `F` to the value returned by the parser `P`. The new value is then passed on to the next stage in the parse.

The parser `many`, given in Figure 1, embodies the repetitious application of some parser `P` to the input, analogous to the BNF construct $P^*$. As well as the construct $P^*$, BNF also provides the notation $P^+$ which selects strictly greater than zero items from term $P$. The corresponding parser `some` is a simple variant of the parser `many` and is also defined in Figure 1.

A grammar invariably contains certain reserved words and symbols which, although vital for guiding the logical structure of expressions in the parse, play no further role after performing this duty. Therefore, it is useful to discard the values of these items during the course of a parse. The two combinators `xthen` and `thenx` (Figure 1) perform exactly this task; `xthen` discards the first component of the value pair and `thenx` discards the second component before proceeding with the remainder of the parse.

**Parsing simple, numerical expressions**

We now give an example of parsing the expressions of a small calculator using the combinators presented earlier. The BNF grammar for the calculator is as follows:

$$e ::= t \texttt{ + } t \mid t \texttt{ - } t \mid t$$
$$t ::= f \texttt{ * } f \mid f \texttt{ / } f \mid f$$
$$f ::= digit^+ \mid \texttt{(} \ e \ \texttt{)}.$$

Of course, `/` and `-` are not associative and, to keep matters simple, we require that extra parentheses are inserted in expressions to explicitly dictate associativity. Therefore, an expression like `1-2-3` does not parse but `1-(2-3)` or `(1-2)-3` will.

The translation from the grammar to its parser (Figure 2) is remarkably straightforward since we provide the combinator equivalent of each BNF construct. We can write the following parse, for example, in our logic language:

```
e Xs V Ys :- ((((t $then ((literal '+') $xthen t)) $using plus) $or
               ((t $then ((literal '-') $xthen t)) $using minus)) $or
              t) Xs V Ys.

t Xs V Ys :- ((((f $then ((literal '*') $xthen f)) $using times) $or
               ((f $then ((literal '/') $xthen f)) $using divide)) $or
              f) Xs V Ys.

f Xs V Ys :- ((number $using value) $or
              ((literal '(') $xthen (e $thenx (literal ')')))) Xs V Ys.

digit X :- elem X "0123456789".
number Xs V Ys :- some (satisfy digit) Xs V Ys.

value X (num X) :- .             plus (X,Y) (add X Y) :- .
minus (X,Y) (sub X Y) :- .       times (X,Y) (mul X Y) :- .
divide (X,Y) (div X Y) :- .      parse Xs V Ys :- e Xs V Ys.
```

Figure 2: The combinator parser for a simple calculator.

```
# :- parse "2+(4-1)*3" V Ys.
V  = add (num "2") (mul (sub (num "4") (num "1")) (num "3"))
Ys = "" ;
V  = add (num "2") (sub (num "4") (num "1"))
Ys = "*3" ;
V  = num "2"
Ys = "+(4-1)*3" ;
no.
```

An expression that cannot be parsed is rejected by the parser:

```
# :- parse "+(4-1)*3" V Ys.
no.
```

By now the explicit replication of logic variables in our predicate definitions may appear tedious: after all, in functional programming we simply omit redundant parameters. Indeed, there is no fundamental reason for preventing this form of $\eta$-conversion of terms in our language. Such syntactic changes to our language compliment compositionality and their inclusion in a complete language implementation would be desirable, as discussed in Section 5.

**Pretty-printing**

One of the most interesting and appealing aspects of programming a parser in a logic programming language is that, to some extent, pretty-printing comes free with the parsing program. Pretty-printing is the converse operation of parsing and, by exploiting the declarative nature of the logic language, we immediately obtain a pretty-printer by running our parser backwards. The following example illustrates this point.

```
# :- parse Xs (add (num "2") (mul (sub (num "4") (num "1")) (num "3"))) "".
Xs = "2+(4-1)*3" ;
Xs = "2+(4-(1))*3" ;
Xs = "2+(4-((1)))*3" .
yes.
```

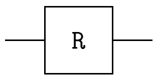
There are, of course, an infinite number of possible pretty-prints for the term since we can indefinitely supply redundant parentheses. Indeed, as we shall see in Section 4, it is important for our logic programming language to employ a particular computational strategy in order to terminate whenever possible.

Our pretty-printer inherits all its knowledge regarding sophisticated layout, e.g., offside rules and indentation conventions, from the parser. However, since our parser is relatively simple, our pretty-printer is consequently simple also. Achieving the complexity of a pretty-printer written exclusively for the task, like [Hug95], would involve extending the parser of this section with additional combinators to capture the required features. Although we do not describe the details here, combinator parsers allow such extensions to be carried out easily.

## 3 Ruby

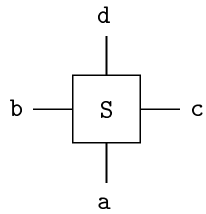
Ruby is a relational language developed by [JS90] for describing hardware circuits. The behaviour of a circuit is specified by a relational program which can then be transformed into an equivalent program amenable to implementation in hardware. A circuit in Ruby is represented as a *binary relation* between simple data values. In this example, we present an implementation of Ruby in our logic language.

The data values on which Ruby circuits operate consist solely of integers, and both tuples and lists of integers. 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` can be represented as the single node




.

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



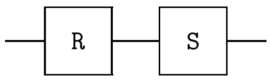

.

Since data flows from right to left in a circuit, we read our relational expressions from right to left, too.

Ruby expressions are composed hierarchically from primitive relations and higher-order predicates that take relations to relations. The primitive relations, defined in Figure 3, are used to construct and deconstruct the data values of a circuit. (Of these relations, though, only `id`, `outl`, and `outr` are essentially primitive in the presence of relational intersection.) These primitive relations provide us with the bare bones of the Ruby language whilst the real expressive power is obtained from higher-order predicates. The most fundamental of these are sequential composition, parallel composition, and converse. The sequential composition of two relations `R` and `S` is given by

```
(R $comp S) A C :- R A B, S B C.
```

The sequential composition of two circuits looks like this:




.

```
A $id A :- .                       (A,A) $fork A :- .
(A,B) $swap (B,A) :- .             nil $null A :- .
A $outl (A,B) :- .                 (A:nil) $wrap A :- .
B $outr (A,B) :- .                 Cs $app (As,Bs) :- append As Bs Cs.
(A,(B,C)) $lsh ((A,B),C) :- .      Bs $apl (A,As) :- append (A:nil) As Bs.
((A,B),C) $rsh (A,(B,C)) :- .      Bs $apr (As,A) :- append As (A:nil) Bs.
```

Figure 3: The implementation of the primitive Ruby relations.

The parallel composition of two relations `R` and `S` is expressed by

```
(R $par S) (A,C) (B,D) :- R A B, S C D.
```

The parallel composition of two circuits simply places them over one another.

Every relation `R` has a well-defined converse `conv R`, given straightforwardly by

```
conv R A B :- R B A.
```

Two commonly used higher-order predicates are simple shorthands, given by

```
fst R A B :- (R $par id) A B.
snd R A B :- (id $par R) A B.
```

Since Ruby is a language for designing circuits, a plethora of predicates exist for combining circuit elements. For two circuits, denoted respectively by the relations `R` and `S`, the predicate `beside` joins `R` and `S` together in the following fashion:

```
beside R S A B :- (rsh $comp (snd S) $comp lsh $comp (fst R) $comp rsh) A B.
```

The beauty of Ruby is exemplified when we define the *dual* of existing predicates in terms of themselves, providing a convenient way to produce new circuits. We can obtain the dual of any higher-order predicate by first taking the converse of all its arguments and then taking the converse of the entire expression. For example, we can manufacture the predicate `below` in terms of its dual predicate `beside` by defining

```
below R S A B :- conv ((conv R) $beside (conv S)) A B.
```

Both `beside` and `below`, interpreted as circuits, are shown in Figure 4.

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

```
row R A B :- ((fst wrap) $comp R $comp (snd (conv wrap))) A B.
row R A B :- ((fst apl) $comp (R $beside (row R)) $comp
                (snd (conv apl))) A B.
```

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 strictly non-empty lists. The predicate `col`, forming a vertical column of circuits, is defined in terms of its dual predicate `row` via

```
col R A B :- conv (row (conv R)) A B.
```

Finally, the two predicates `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

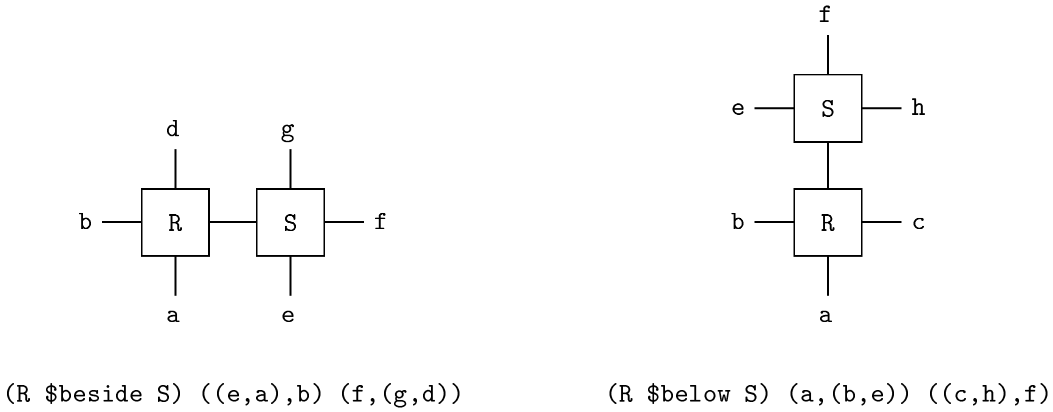



Figure 4: The circuits corresponding to `beside` and `below`.

```
rdl R A B :- (outr $comp (row ((conv outr) $comp R))) A B.
rdr R A B :- (outl $comp (col ((conv outl) $comp R))) A B.
```

respectively.

An attractive aspect of using the logic language to implement Ruby is the ease with which Ruby operators translate from abstract mathematics into a concrete program. This exemplifies the declarative nature of our language.

### A Simple Sorting Circuit in Ruby

We now present a circuit for sorting an arbitrary, non-empty list of natural numbers. (Natural numbers are represented as "Peano" naturals in the programming language.) Two alternative methods for describing the sorting circuit have been given in the past, one by [SJ87] and the other by [Hut92a]. We implement the former presentation here.

The input to the circuit is a list of naturals and the output is the same list but sorted according to the relation `cmp`. The definition of the circuit is as follows:

```
cmp A B :- (min $par max) $comp fork A B.

sort A B :- (wrap $comp (conv wrap)) A B.
sort A B :- (apr $comp (snd (conv wrap)) $comp
              (conv (rdr (conv ((col cmp) $comp (conv apr)))))) A B.
```

given the appropriate definitions of `min` and `max`. To the uninitiated, relational expressions can appear quite incomprehensible and the ability to execute such expressions is clearly a useful instruction tool.

Given the above program, we can query

```
# :- sort A (2 : 4 : 0 : 1 : 3 : 5 : nil).
A = (0 : 1 : 2 : 3 : 4 : 5 : nil) ;
no.
```

However, we encounter a more interesting example by solving the converse of `sort`. That is, the query

```
# :- conv sort A (0 : 1 : 2 : 3 : 4 : 5 : nil).
```

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

# 4 Resolution and Fairness

We now describe the resolution strategy employed in our relational interpreter and illustrate some of the inherent drawbacks of adopting resolution as the operational semantics of a programming language.

### SLD-resolution

We can visualise the proof of a goal forming a tree structure (called an *SLD-tree*); the branches of the tree represent the various choices of goal literal and program clause at each step in the resolution. Furthermore, it is not difficult to imagine that this tree quickly becomes enormous. From the point of view of systematically searching for a proof of a goal, we would certainly like to reduce the size of search space that a machine must examine before successfully finding an answer. One resolution-based method of searching for proofs that restricts the size of a search tree is called *SLD-resolution*, adopting linear resolution (always resolving a literal in the goal with a clause taken from the program) and a predetermined method of selecting the next goal literal to resolve with, dictated by the *computation rule* (or *selection strategy*). [AvE82] prove the soundness and completeness of SLD-resolution showing that if a goal has any resolution at all then it also has an SLD-resolution.

Most implementations of logic programming languages, whose computational model is based on SLD-resolution, employ a stack-based computation rule, i.e., the leftmost literal of a goal is always selected at each resolution step and any inferred literals are left in its place. However, the search for a proof can sometimes be substantially more difficult when using stack-based SLD-resolution than some other resolution strategy. For instance, consider the query

```
# :- append Xs Ys Zs, append Xs Ys (a:b:nil).
```

The strictly left-to-right strategy of selecting stacked goals effectively enumerates all lists `Xs` and `Ys` which, when concatenated, form `Zs`, and then checks each of them to see if `Xs` and `Ys` concatenate to form the list `(a:b:nil)`. Of course, there are an infinite number of such lists resulting in the non-termination of this query.

On the other hand, by selecting the rightmost literal first in the previous example, we would only have to enumerate three possibilities for `Xs` and `Ys` thus dramatically reducing the amount of search we have to perform before discovering a proof. In fact, this means that SLD-resolution can be incomplete, in a practical sense, when searching for a proof of a goal. Despite the problems associated with stack-based SLD-resolution, it remains predominant in logic programming owing almost entirely to the ease of its implementation.

### Fair SLD-resolution

We can regain completeness of SLD-resolution by ensuring that, after a finite number of resolution steps in the computation, every literal in a goal is chosen for resolution first by the selection strategy. A selection strategy that guarantees this condition is called a *fair* computation rule [LM84]. A simple way to achieve fairness in a computation rule is to "queue" the literals pending resolution, i.e., adding those literals introduced by a resolution step to the end of the current goal and always selecting the leftmost literal.

We know that varying the computation rule does not affect the answer substitutions contained in the corresponding SLD-tree—the number of success nodes in an SLD-tree is invariant with respect to the computation rule—but rather alters the respective size of the SLD-tree and hence influences the overall efficiency of the SLD-resolution. More to the point, if a "loop" in a program can be avoided by some computation rule then any fair computation rule will avoid it. We formalise this result by the following lemma—adhering to the notation of [Llo87]—whose proof is beyond the scope of the paper and is omitted.

**Lemma 1.** *For a program $D$ and a goal $G$, if some SLD-tree for $D \cup \{G\}$ is finite then every fair SLD-tree for $D \cup \{G\}$ is finite.*

In the above discussion, little emphasis has been placed on the choice of search strategy used to traverse the SLD-tree generated by an SLD-resolution. We discuss this and other related points in the following section.

## 5 Discussion

In this paper, we have presented a compositional style of logic programming. We hope to have persuaded the reader that compositional logic programming is a meaningful addition to the varieties of logic programming now in use. We have also attempted to identify what language features should be supported by a logic language to make such a compositional style possible. It is now appropriate to ask whether these features are present in existing languages, in particular higher-order languages such as λProlog. Below we summarise the main conclusions, answer questions on related languages, and indicate directions of future research.

### Language features

What language features are essential to a compositional style of logic programming? First, it is necessary to have a fair selection rule, so that a computation terminates whenever possible. As is well-known among logic programmers, the left-selection rule often necessitates multiple versions of the same predicate, depending on which variables will be instantiated or not. A good example of this phenomenon is the predicate that relates a binary tree to its frontier:

```
frontier (tip A) (A:nil) :- .
frontier (bin S T) (A:B:X) :- frontier S (A:Y), frontier T (C:Z),
                              append Y (C:Z) (B:X) .
```

Assume that a left-selection rule is used, and we attempt to solve the goal

```
# :- frontier T (a:b:c:nil) .
```

The proof search will report one solution for `T`, and then enter an infinite loop. By contrast, a fair selection rule will find both solutions for `T` and then terminate. The standard solution to this problem is to write another predicate

```
cfrontier (A:nil) (tip A) :- .
cfrontier (A:B:X) (bin S T) :- append Y (C:Z) (B:X), cfrontier (A:Y) S,
                               cfrontier (C:Z) T .
```

But rewriting a program according to the context where it is used defies the very notion of compositionality.

In particular, consider an elementary composition operator such as converse:

```
conv R A B :- R B A .
```

A minimum healthiness requirement for the above programs is the equivalence

```
conv frontier A B = cfrontier A B.
```

However, as can be seen from the above example, the two sides of the equation exhibit different behaviours when the left-selection rule is used. By contrast a fair selection rule guarantees that, for programs that only exhibit finite nondeterminism, such surprises do not occur. We conclude that a fair selection rule is necessary if we wish to implement converse. Similar arguments can be made for other elementary composition operators such as relational composition and intersection.

A second language feature that is necessary for compositional programming is the 'curried' representation of terms, because this allows partial parameterisation of predicates. Such partial parameterisation has long been one of the hallmarks of good functional programming style. To use it in a logic language, only very minor changes to the standard unification algorithm are necessary, and therefore this feature could be used in any logic language at no extra cost.

### Connection with $\lambda$Prolog

Since we wish to have curried terms, the reader might wonder where we depart from $\lambda$Prolog. Could one not write compositional logic programs in $\lambda$Prolog? Our investigation of compositional logic programming started with precisely this question, and the results were summarised in [McP95]. The most obvious departure from $\lambda$Prolog is the use of a fair selection rule. For particular examples, one can mimic the behaviour of a fair selection rule in $\lambda$Prolog through its extra-logical features. However, we were unable to do so systematically, and although we could make the examples of this paper work in $\lambda$Prolog, it was not possible to determine whether successful execution was an accident or not.

Another divergence from $\lambda$Prolog is the absence of $\lambda$-abstractions in our terms. Because we do not have $\lambda$-abstractions, we also do not need to consider the intricacies of higher-order unification. Higher-order unification is undecidable, and therefore the proof search of $\lambda$Prolog is incomplete in at least three ways: through the use of the left-selection rule, through depth-first search, and through the undecidability of unification. We conclude that, in general, little can be proven about the operational behaviour of a program in $\lambda$Prolog. It came as something of a surprise to us that we could restrict 'higher-order' features entirely to curried terms, although it has been pointed out to us since that similar techniques are employed in the type systems of certain lazy functional languages [Jon95].

True, the definitions of our predicates would be more pretty if we could make use of $\eta$-conversion, as is often done in functional programming. One could then write, for instance,

```
literal X :- satisfy (eq X) .
```

instead of the more verbose

```
literal X Xs V Ys :- satisfy (eq X) Xs V Ys .
```

However, given an appropriate type system, it is easy to *$\eta$-expand* all definitions at compile-time, so we regard this 'higher-order' language feature as a mere matter of syntax.

### On the use of breadth-first search

When we set out on this work, we expected that we would need to use a complete search strategy such as breadth-first search. Clearly if one is interested in finding all answers, even when there is an infinite number of them, one must use a complete strategy. Note, however, that the use of breadth-first search will not alter the search space: in particular, if the space is infinite, breadth-first search will not terminate. To illustrate, consider again the `frontier` example, now with the left-selection rule plus breadth-first search. The computation would find both trees `T` that have frontier `a:b:nil`, but then it would continue searching without reporting further answers. It follows that it is still not the case that `conv frontier` and `cfrontier` are equivalent programs, and therefore we ruled out the combination of breadth-first search with the left-selection rule.

From a purely declarative perspective, an ideal implementation would employ a fair selection rule plus breadth-first search. In fact, our present implementation allows the user to choose whether depth-first search or breadth-first search is used. It remains to be seen, however, whether the additional memory requirements of breadth-first search are justified by gains in compositionality. Although we believe it is, we have not yet found a convincing example to illustrate that conjecture.

### The prototype implementation

The implementation referred to in this paper is a toy, constructed for fun as part of our course material on Principles of Programming Languages. The implementation language is Caml-light, and (not counting the lexer, parser and pretty-printer) the interpreter consists of merely 250 lines of code. This demonstrates that we have not only achieved simplicity in the source language, but also in the implementation. Because the present implementation is naive in the extreme, there is ample scope for improving its efficiency.

Such improvements will be needed for the language to be pleasant to use. The fair selection rule that we use (essentially implementing goals as queues of literals) results in a tremendous number of resolutions steps being taken. It might be that fair selection rules exist with a better efficiency behaviour; we have not investigated this at all. As stated before, our prime concern is the prototyping of specifications, and so efficiency is of secondary importance.

### Future work

It would be instructive to expand the case studies of this paper, especially the parsing/pretty-printing one, to more realistic examples. Such extended case studies might shed further light on the desiderata of a compositional logic language. It is already clear that the syntax would need to be much improved, especially to support $\eta$-conversion, as outlined above.

Another syntactic, more radical improvement would be to adopt a syntax based on non-deterministic $\lambda$-expressions. Through the use of $\lambda$-lifting, it is possible to translate a large class of functional programs into the language presented here. By adding operators for nondeterministic choice, intersection and converse to a functional notation one gains expressive power similar to that of the language shown here, but in a more concise notation. Of course, the brevity of such a notation should be weighed against the loss of referential transparency in the purely functional part of the language.

On the implementation side, we are investigating techniques where, in analogy with well-known techniques for functional programming, all variables are eliminated through an appropriate set of constants. It is, at present, too early to endeavour comments on the success of that work. We hope to use the approach sketched here (which is a minimal change to traditional logic programming) as a yardstick for judging more adventurous models of execution.

## Acknowledgements

We would like to thank Christopher Hogger and John Lloyd for their discussions regarding fairness, and Graham Hutton and Philip Wadler for their comments on an earlier draft of the paper. Richard McPhee gratefully acknowledges the financial support of the Engineering and Physical Sciences Research Council of the United Kingdom.

## References

[AvE82] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[Bac85] J. Backus. From function level semantics to program transformations and optimization. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Mathematical foundations of software development, Vol. 1*, volume 185 of *Lecture Notes in Computer Science*, pages 60–91. Springer–Verlag, 1985.

[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.

[BH93] R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42, 1993.

[BVK95] R. Berghammer and B. Von Karger. Formal derivation of CSP programs from temporal specifications. In *Mathematics of Program Construction*, volume 947 of *Lecture Notes in Computer Science*, pages 180–196. Springer–Verlag, 1995.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures and Languages Europe*. Springer-Verlag, June 1993.

[Han94] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19(20):583–628, 1994.

[HC95] T. Hallgren and M. Carlsson. Programming with Fudgets. In *International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Hug95] J. Hughes. The design of a pretty-printing library. In *International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[Hut92a] G. Hutton. *Between Functions and Relations in Calculating Programs*. Research report, Department of Computer Science, Glasgow University, 1992.

[Hut92b] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[Jon95] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), 1995.

[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.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[LM84] J. L. Lassez and M. J. Maher. Closures and fairness in the semantics of programming logic. *Theoretical Computer Science*, 29(1–2):167–184, 1984.

[McP95] Richard McPhee. Implementing Ruby in λProlog. Available over the internet at URL `http://www.comlab.ox.ac.uk/oucl/users/richard.mcphee/publications.html`, Programming Research Group, University of Oxford, U.K., 1995.

[MNRA92] J. Moreno-Navarro and M. Roderiguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–223, 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.

[Wad85] P. Wadler. How to replace failure by a list of successes. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128. Springer-Verlag, 1985.