IT 11 015
Examensarbete 15 hp
Mars 2011

The effect of using a trailing persistent array to embed logic programming into a functional language

Nicklas Nordenmark

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

The effect of using a trailing persistent array to embed logic programming into a functional language

Nicklas Nordenmark

Logic programming is an important paradigm because of its declarative nature – a programmer declares values and facts and then the program executes by inferring their consequences via backtracking search and unification.
There are many situations where logic programming allows elegant solutions that are difficult to emulate in other paradigms, such as implementing type inference or solving problems that require backtracking search.
Unfortunately it is generally not feasible for a language to be purely based on logic – search spaces are often large or infinite and greater control is required, normally via constructs that move closer to other non-logical paradigms.
An attractive approach that has been attempted by for example Felleisen [12] and Seres & Spivey [16] is to embed logic programming into a host language with rich control constructs such as a functional language.
This report describes a new technique for implementing such an embedding that improves on previous embeddings by concealing trailing and reversion with the help of a persistent array data structure proposed by Baker [6]. This structure was recently used in a domain similar to ours with backtracking by Conchon & Filliatre [8].

Handledare: Rowan Davies
Ämnesgranskare: Konstantinos Sagonas
Examinator: Anders Jansson
IT 11 015
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

1.1 Background

Logic programming [14] is originally based on proof search. To achieve this, logic programming languages come pre-packaged with inbuilt features, such as backtracking and logic variables with unification, that make them easy to use for dealing with many kinds of problems.

This logical foundation alone is often insufficient for programming in practice, and additional constructs are added like those in other paradigms, as well as constructs to control proof search like the *cut*(!) of *Prolog* [3].

An alternative solution is to embed a logic programming language into another language as per the work of *Felleisen* [12], *Haynes* [13], *Seres & Spivey* [16] and the systems *Schelog* [5] and *Racklog* [4]. This has the advantage that the logical parts can stay close to the logical foundation, while providing other ways to control programs via the host language.

1.2 Problem definition

An issue with this approach is providing a relatively direct and efficient embedding that also integrates well with the host language and provides easy ways to program common control patterns like cut. A variety of different embeddings have been proposed, with the most successful being based on embeddings into functional languages that use success continuations with effects, failure continuations, both kinds of continuations or lazy lists/monads with backtracking.

In designing such embeddings extra care must be taken to make sure that the values of logic variables are restored when backtracking. This issue is analyzed in depth in section 3 of a Scheme embedding paper by *Haynes* [13].

1.3 Motivation

Using imperative reference cells for logic variables and success continuations is both efficient and easy to implement. This has been done by for example *Elliott & Pfenning* [11] and the design leads to a structure similar to the *Warren Abstract Machine* [17], which is the basis of many optimized logic programming implementations. However, a clear downside with this style becomes apparent when trying to implement control constructs like *cut*. Basically, the programmer is required to manually mark choice points by calling a "trail" function is required.

The recently suggested *trailing persistent arrays* [8] seem to be a suitable data structure for a logic embedding like this. The embedding can be quite direct without performance issues.

The technique is related to the technique "trailing" in the implementation of logic programming languages, but it is hidden by the embedding via the persistent array.

In particular this allows control constructs, like exceptions and continuations, from the host language to be used without the need for the programmer to mark choice points and without the possibility of variables incorrectly retaining values.

1.4 Related research

Embeddings and interpreters like those described in the problem definition have been investigated by many researchers. *Felleisen's* work in Scheme [12] is probably one of the earliest publications. There are many examples of implementations that follow along the same lines: an interpreter in Standard ML by *Elliott & Pfenning* [11] an embedding in Scheme by *Haynes* [13] and an embedding in Haskell by *Seres & Spivey* [16]. There are also two related embeddings in Scheme dialects; *Schelog* [5] in Scheme and *Racklog* [4] in Racket.

1.5 Approach

In the coming section we'll look at how the embedding was developed and move on to explaining the syntax in section 3. Finally some benchmarking results will be presented in section 4 and future work and conclusions will be discussed in section 5.

2 Development of the Embedding

OCaml was chosen as the host language due to its excellent performance, widespread popularity and rich functional foundation. Although the ideas and concepts introduced are general and can be applied to most functional programming languages that have mutable arrays, higher-order functions and exceptions. The development process was iterative as small improvements were made with each new version of the embedding.

2.1 Initial version

The starting point version was an imperative embedding written in F# by my project supervisor *Davies* [10] based on success continuations and imperative substitutions, i.e. logical variables represented as reference cells. This follows the work of *Elliott and Pfenning* [11] which in turn uses the ideas of success continuations, due to *Carlsson* [7].

This embedding has a major limitation: It is unsound if control constructs like exceptions are used in the host language. As a result it is not possible to implement control constructs like cut without significantly complicating the embedding by requiring each choice point to be marked by the programmer (similar to *Elliott and Pfenning* [11], except they propose an interpreter, not an embedding).

2.2 Imperative OCaml version

The first step was basically a port from the F# version to an OCaml version. Thanks to the similarities in syntax between F# and OCaml, this porting was very straight forward with a few modifications in syntax where necessary. The type used for terms was as follows:

```
type term = Var of term option ref | Con of (con * term list) | Int of int
type subs = unit
```

2.3 Functional version using Maps

This version meant changing the way substitutions worked. At this point logical variables basically pointed to their corresponding (integer) keys in the map data structure.

```
type term = Var of int | Con of (con * term list) | Int of int
type subs = (term) IntMap.t;;
```

2.4 Functional version using a Persistent Array

This step meant a transition from the map to the trailing persistent array. This was a simple process as the array interface is very similar to that of the map.

```
type term = Var of int | Con of (con * term list) | Int of int
type termOpt = term option;;
type subs = (termOpt data) Batteries.ref;;
```

2.5 Introducing "garbage collection"

The previous version did not re-use variables and thus the arrays grew to enormous lengths (4,5 million elements in the 10 queens problem). By keeping track of the number of allocated variables so far in the initial position of the array, the length could be reduced to about 310 elements. This also resulted in a major gain in performance.

2.6 Resizable Persistent Array

The previous version required us to know the array length needed for each individual problem. So, the persistent array was modified to automatically double its size whenever accessed out of bounds. This implementation was fairly straight forward and made use of the exception handling of OCaml.

The helper function that returns a new array double the size of t with elements copied over:

```
let rec doubledArr t =
  incr doubles;
  let arrLen = length t in
```

```
    (* Consider only items in range of the old array t *)
  let func i = if i < arrLen then get t i else None in
    (if arrLen > !size then size := arrLen);
    init (arrLen*2) func

and get t i =
  (* is i out of bounds? *)
  try
    match !t with
      | Array a ->
          a.(i)
      | Diff _ ->
          reroot t;
          begin match !t with Array a -> a.(i) | Diff _ -> assert false end
  with
      (* If we try to access out of bounds, we won't find anything useful,
          just return None *)
      Invalid_argument msg -> None

let rec set t i v =
  try
    reroot t;
    match !t with
       | Array a as n ->
           let old = a.(i) in
               if old == v then
                  t
               else begin
                  a.(i) <- v;
                  let res = ref n in
                      t := Diff (i, old, res);
                      res
              end
      | Diff _ ->
           assert false
  with
      Invalid_argument msg ->
           t := !(doubledArr t);
           set t i v
```

2.7 Performance tweaking

A few tweaks were implemented in order to improve the performance a little bit further in an attempt to match the performance of the imperative version. Firstly the persistent array was turned into a semi-persistent array according to the work of *Conchon & Filliâtre* [9], i.e. backtracking to earlier versions is supposed but not revisiting later versions. The out of bounds check for accessing the array was changed from using exceptions to a basic conditional check of the array length and the `-unsafe` was used for compilation (which turns off bounds checking for arrays and strings). Finally, the number of variables currently allocated is no longer stored in the array but in a pair along with the array. All in all this resulted in a performance gain of about 40% compared to the resizable array version in section 2.6.

2.8 "Semi functional" version

We denote this a "semi functional" version since it's a mixture of logic programming style with backtracking and ordinary functional code such as pattern match-

ing and list operations. In order to further optimize our embedding we can chose to divert from our strict logic programming style and use functional style code where applicable and match or even beat the performance of some Prolog compilers. This proved to give a performance that was almost 10 times better than YAP.

3 Syntax

One of the goals with this embedding is to try and keep the embedded programs structurally similar to Prolog programs with only some minimal "syntactic noise". The syntax is best explained using comparisons with common Prolog problems.

3.1 Member

```
member(X, [X | Tail]).
member(X, [Head | Tail]) :- member(X, Tail).
```

VS

```
1 let rec member args k s =
2     let x, head, tail, s = newV3(s) in
3
4     unify2 args  (x, (x ^| tail))  <| k <| s;
5     unify2 args  (x, (head ^| tail))  -|  member(x, tail)  <| k <| s
```

The first two lines are just "syntactic noise" compared to the Prolog version. They are very straight forward to write as they're always quite similar in all programs; only differing in function and variable names. In this case, **newV3** allocates 3 new variables.

The first line in the embedded version defines the **member** function that takes the Prolog arguments as **args**, followed by the success continuation **k** and substitution structure **s**, which links logic variables to values. The rest of the lines are basically ported from the Prolog version. **unify2** attempts to unify **args** with a pair of 2 terms, creating a logic programming computation which accepts a success continuation and a substitution. Furthermore, additional logic programming computations can act as success continuations via function composition (`-|`) as is demonstrated by the recursive call to **member** on line 5. This composition corresponds to a logical *and*. After such a sequence of compositions, the success continuation (**k**) for the entire call to **member** is given (via `<|`). As the logic programming computations succeed substitutions are passed to the respective continuations. For starters, the empty substitution **s** is given as input to the top level logic programming computation via `<| s`. Logical *or* is implemented by imperative sequences via `;`.

3.2 The cut(!) construct

Once a satisfactory behavior of the persistent array was in place, the *cut* functionality of Prolog could be implemented. The *cut* operator disables backtracking from marked points in your programs. In its essence, this can be seen as making a choice at one point and stand by that when backtracking. This was implemented with the help of exceptions and was very straight forward thanks to the persistent array (not having to specify a depth to return to explicitly as *Carlsson* [7]). This only affects the individual programs that use cut and not the embedding in general. In order to ensure we're catching the correct exception we define the exception to take a unit reference cell and declare the reference cell every time we use cut.

```
1  exception CutOff of (unit ref);;
2
3  let cut here = fun kk ss -> kk ss; raise (CutOff here)
4
5  let cutWrap f =
6    let here = ref () in
7    try
8      f here
9    with
10     CutOff h when h=here -> ()
```

```
(* corresponds to PROLOG
   bb(1).
   bb(2).
   bb(3).
*)
```

```
1  let bb x k sub =
2    x ** !!1 <| k <| sub;
3    x ** !!2 <| k <| sub;
4    x ** !!3 <| k <| sub
```

```
(* corresponds to PROLOG
   sel(X,Y) :- bb(X), bb(Y).
*)
```

```
1  (* Without cut *)
2  let sel x y k sub =
3    bb x   -|   bb y   <| k <| sub;;
```

```
(* corresponds to PROLOG
   sel(X,Y) :- bb(X), !, bb(Y).
*)
```

```
1  (* Including cut *)
2  let selcut x y k sub =
3    cutWrap <| fun here ->
4      bb x   -|   cut here   -|   bb y   <| k <| sub
```

4 Results

An interesting aspect of this project was to see whether a functional embedding (making use of the trailing persistent array) could compete with the imperative counterpart in terms of performance. Below are benchmarking results from the 10 queens problem, a problem involving difference lists and a situation where variable trailing is required. All runtimes are in seconds and the timings were done on a 2.2 GHz 4 GB RAM Macbook.

4.1 N queens

This is a classic problem that involves finding suitable positions for N number of queens on a NxN chessboard so that no single queen can attack another queen. The original Prolog code that the benchmarks are based on is shown below and was taken from [1].

```
% when placing queen in empty list, solution found
queens([]).

% otherwise, for each row
% place a queen in each higher numbered row
% pick one of the possible column positions
% and see if that is a safe position
% if not, fail back and try another column, until
% the columns are all tried, when fail back to
% previous row
queens([ Row/Col | Rest]) :-
            queens(Rest),
            member(Col, [1,2,3,4,5,6,7,8]),
            safe( Row/Col, Rest).

% the empty board is always safe
safe(Anything, []).
% see if it attacks the queen in next row down
safe(Row/Col, [Row1/Col1 | Rest]) :-
            Col =\= Col1,  % same column?
            Col1 - Col =\= Row1 - Row,    % check diagonal
            Col1 - Col =\= Row - Row1,
            safe(Row/Col, Rest). % no attack on next row,
                                 % try the rest of board

% member will pick successive column values
member(X, [X | Tail]).
```

```
member(X, [Head | Tail]) :- member(X, Tail).

% prototype board
board([1/C1, 2/C2, 3/C3, 4/C4, 5/C5, 6/C6, 7/C7, 8/C8]).
```

F# imp.	OCaml imp.	OCaml map	OCaml array
1.80	0.97	25.50	7.85

OCaml GC	OCaml dynamic	Tweaked	Semi functional	YAP
2.87	3.50	2.03	0.03	0.16

Table 10: Execution time (in secs) for the 10 queens problem

# of Queens	11	12	13	14
Semi func	0.12	0.64	3.89	26.1
YAP	0.83	5.08	33.45	4 min

Table 11: Execution time (in secs) for larger N queens boards

Just using OCaml instead of F# for the imperative version (see section 2.2) proved to cut the runtime in half and therefore support our initial statement that OCaml would be a good language choice thanks to its speed. The second OCaml version using the maps (section 2.3) was really only included as a transition version that would ease the switch to using the persistent array in the next version (section 2.4). A comparison between these two versions is still interesting as they both work functionally. The first persistent array version didn't reuse variables and the array size grew to enormous proportions (about 4,500 000 elements).

This was a severe performance issue so a *garbage collecting* version was developed (section 2.5) that reduced the required length to about 310. This tweak made the embedding run at almost three times the speed. Next version added on an automatic resizing feature of the array which resulted in a slight overhead performance wise. The final version (section 2.7) introduced a few performance tweaks and turned out to be about 40% faster than its predecessor. The results from the semi functional version was mostly included as an indication of how fast the embedding can be if necessary by introducing functional programming style code (such as pattern matching and list operations) where applicable.

4.2 Difference lists

This particular example involves bidirectional flow via unification in order to support the standard technique of difference lists, and is based on an example due to *Pfenning* [15].

		# Iterations			
		1	10	100	1000
List Length	10	0.007	0.009	0.022	0.12
	100	0.010	0.025	0.16	1.46
	1000	0.034	0.26	2.46	24.5
	10000	0.5	4.96	50	10min

Table 12: Execution time (in secs) for difference lists OCaml (section 2.7)

		# Iterations			
		1	10	100	1000
List Length	10	< 0.001	< 0.001	< 0.001	0.005
	100	< 0.001	0.001	0.004	0.027
	1000	0.001	0.003	0.028	0.245
	10000	0.003	0.028	0.289	2.92

Table 13: Execution time (in secs) for difference lists YAP Prolog

Clearly, YAP optimizes the difference lists code despite the bidirectional flow. The YAP code runs almost 200 times faster than the OCaml code for long lists.

4.3 Trailing

This benchmark includes creating 100 choice points and trailing 100 variables according to benchmark # 13. in [2].

	# Iterations			
	1000	10000	50000	100000
OCaml	0.058	0.48	2.34	4.67
YAP	0.145	1.65	9.10	17.6

Table 14: Execution time (in secs) for trailing of variables

4.4 SEND MORE MONEY (Version 1)

This benchmark deals with the classical problem of finding possible values for the different characters so that SEND+MORE=MONEY holds.

OCaml	YAP
12.8	1.07

Table 15: Execution time (in secs) for the SEND MORE MONEY problem (Version 1)

When solving this problem in this way (see Appendix, SEND MORE MONEY (Version 1)) we are forced to use a lot of arithmetic and the embedding really suffers from this as we have to convert back and forth between embedding integers (Terms) and OCaml integers in order to do the computations.

4.5 SEND MORE MONEY (Version 2)

This implementation of the SEND MORE MONEY problem uses only simple unifications and comparisons compared to the old version which does a lot of variable comparisons that require us to convert from the term type used in the embedding to regular OCaml integers.

OCaml	YAP
0.07	0.03

Table 16: Execution time (in secs) for the SEND MORE MONEY problem (Version 2)

This implementation proved to be a major increase in performance. Especially for the OCaml version since we no longer were forced to convert back and forth between embedded integers and OCaml integers in order to do all computations that were necessary in Version 1.

5 Future Work & Conclusions

5.1 Future work

Add additional benchmarks

Adding even more benchmarks to find out in what types of problems this embedding performs well and also what areas could be improved further.

Add library functions

More library functions (similar to **list2term** and **term2list**) could be added to support the implementation of semi-functional programs where needed.

Additional infix operators

The embedding could really benefit from having more infix operators for various functions. For example an infix operator to replace the `diff` function in the SEND MORE MONEY program could make the embedding programs even more similar to actual Prolog code.

Syntax

The syntax could be taken even closer to regular Prolog syntax by making use of the very powerful Camlp4/Camlp5 syntax extensions. Also, regular OCaml lists (alternatively functions taking OCaml lists and returning lists of the embedding) could be used instead of the embedding construction – resulting in cleaner code.

Bridging namespace

In its simplest form, this embedding only prints the solutions found, returning `unit`. An apparent improvement over this would be to either return a structure of solutions explicitly from the functions or alternatively, to store the solutions globally in a reference cell. This way, we can access the solutions functionally if further manipulations of the values found are desired.

User request for more solutions

Currently the user is always presented with all solutions when making a query; this could be modified to be closer to Prolog's style of "more solutions on demand". By storing the solutions in a lazy list, we could reduce the unnecessary work done for unwanted solutions. A slight limitation of the embedding is that it can't handle problems that have an unlimited number of solutions in its current state.

Porting script

It is very tedious to port large Prolog programs and a script to automate certain parts of this conversion could really come in handy.

5.2 Conclusions

The implementation of control constructs (namely the *cut* (!) operator) proved to be very straight forward thanks to the trailing persistent array. The need to mark choice points was eliminated and the restoring of values of the logic variables could be abstracted by the array structure while it was used functionally in an intuitive manner. Performance-wise, the tweaked OCaml functional version (section 2.7) almost matched the performance of the original imperative embedding written in F#. If deemed necessary it is also possible to optimize the individual programs to run in a semi-functional manner (introducing functional programming concepts such as pattern matching and lists along with the logic programming concepts backtracking etc) - increasing the performance dramatically. This optimization proved to outperform YAP by almost a factor of 10 for the larger N queens boards.

Working with the embedding

It became clearer and clearer as more Prolog programs were ported that the syntax really needed improvements. The longer Prolog programs took a long time to port just because the syntax was really difficult to work it - especially when it came to inbuilt Prolog constructions (like the exp $\rightarrow$ exp; construct and the "N is N-1" construct). It also became very clear that the embedding really struggles with performance (as well as readability, see Version 1 and 2 of the SEND MORE MONEY program in the Appendix) if a lot of arithmetic was involved - as this required us to explicitly convert embedded integers (Terms in the embedding) to regular OCaml integers in order to compute values and do comparisons. I also realized that in order to become a complete embedding, a lot of Prolog built-in predicates (such as var and nonvar) need to be implemented.

6 Appendix

6.1 N Queens

Functional

```
let rec membOpt (x, poses) k sub =
  if (poses = nil) then
    ()
  else begin
    let head, tail, s = newV2(sub) in
      unify poses (head ^| tail)
 (fun s -> unify x head k s; membOpt (x, tail) k s) <| s
  end

let rec safeOpt (row, col, arg3) k sub =
  unify arg3 (nil) <| k <| sub;
  let row1, col1, rest, s = newV3(sub) in
    unify arg3 ((row1/col1) ^| rest) -|
 begin fun kk s ->
   let ns tm = n s tm in
     if
       (ns col) = (ns col1) ||
       (ns col1) - (ns col) = (ns row1) - (ns row) ||
       (ns col1) - (ns col) = (ns row) - (ns row1) then
       ()
     else
       kk s
 end
      -| safeOpt(row, col, rest) <| k <| s

let rec queens args k sub =
  unify args (nil) <| k <| sub;
  let (row, col, rest, s) = newV3(sub) in
    unify args ((row/col) ^| rest) -| queens(rest)
                                   -| membOpt(col, positions)
                                   -| safeOpt(row, col, rest) <| k <| s
```

Semi functional

```
(* An efficient version of queens that avoids unification. *)
let rec safe3 (row,col,rest) k s =
  match rest with
    | [] -> k s
    | (row1,vcol1,col1)::tail ->
   if col = col1 ||
      col1 - col = row1 - row ||
      col1 - col = row - row1 then
         ()
   else safe3(row, col, tail) k s


let rec queens3 args k s =
  match args with
    | [] -> k s
    | (Con (div, [Int row; Var vCol])) :: rest ->
        (queens3(rest) <| fun s ->
            positions2 |> List.iter (fun (Int c) ->
                safe3(row, c, s) k ((row,vCol,c)::s) ) )
   <| s

(* This wraps the efficient queens3 so that it externally acts
   just like the original.
```

```
   I.e., it calls its continuation with the same substitutions.  *)
let queens3wrapped args k s =
  let argsList = term2list (doSubst s args) in
  let addCol kk (_,vCol, col) ss = addSubst vCol (!!col) kk ss in
  queens3 argsList (fun sList -> (List.fold_left addCol  k sList) s) []
```

6.2 SEND MORE MONEY

Version 1

```
let rec assign_digits args k s =
  let (_List, _D, _Ds, _NewList, s) = newV4 s in
    unify2 args (nil, _List) <|k<|s ;
    unify2 args (_D ^| _Ds, _List) -|
    select(_D, _List, _NewList) -|
    assign_digits(_Ds, _NewList) <|k<|s

let _Digits = (!!0 ^| !!1 ^| !!2 ^| !!3 ^| !!4 ^| !!5 ^| !!6 ^| !!7 ^| !!8 ^| !!9 ^| nil);;

let send_more () =
  let (_S, _E, _N, _D, _M, s) = newV5(initArr()) in
  let (_O, _R, _Y, s) = newV3 s in
  let _X =  (_S ^| _E ^| _N ^| _D ^| _M ^| _O ^| _R ^| _Y ^| nil) in
  assign_digits(_X, _Digits) -|
  (fun k s -> if n s _M > 0 && n s _S > 0 then k s) -|
  (fun k s -> if 1000*^(n s _S) ++ 100*^(n s _E) ++ 10*^(n s _N) ++ (n s _D) ++
                 1000*^(n s _M) ++ 100*^(n s _O) ++ 10*^(n s _R) ++ (n s _E) =
                 10000*^(n s _M) ++ 1000*^(n s _O) ++ 100*^(n s _N) ++ 10*^(n s _E) ++
                 (n s _Y)
              then (Printf.printf "%s\n" (toString (doSubst s _X)); printLetters _X s)
  ) <|(kSols [] false)<|s;;
```

Version 2

```
let digit n k s =
  unify n !!0 <|k<|s;
  (* ... through to ... *)
  unify n !!9 <|k<|s;;

let leftdigit n k s =
  unify n !!1 <|k<|s;
  (* ... through to ... *)
  unify n !!9 <|k<|s;;

let sumdigit args k s =
  cutWrap <| fun here ->
    let _C, _A, _B, _S, _D, s = newV5(s) in
    let _X, s  = newV(s) in
     (* _S and _D are outputs, must be unified, not just compared *)
      unify5 args (_C,_A,_B,_S,_D) -|
 _D ** !!0 -|
 _X %<< (_C +^ _A +^ _B) -|
 (fun k s ->
    if n s _X < 10 then _S ** _X -| cut here <|k<|s) <|k<|s;
      unify5 args (_C,_A,_B,_S,_D) -|
      _X %<< (_C +^ _A +^ _B) -| _S %<< (_X -^ !!10) -|
   _D ** !!1 <|k<|s;;


let diff _X _Y k s =
  if n s _X <> n s _Y then k s;;
```

```
let send k s =
  let (_S, _E, _N, _D, _M, s) = newV5(initArr()) in
  let (_O, _R, _Y, s) = newV3 s in
  let (_C1, _C2, _C3, s) = newV3 s in
  let _X =  (_S ^| _E ^| _N ^| _D ^| _M ^| _O ^| _R ^| _Y ^| nil) in
    (* digit(D), digit(E), D=\=E, *)
    digit _D -| digit _E -| diff _D _E -|
    (* sumdigit2(0, D, E, Y, C1), *)
    sumdigit(!!0,_D,_E,_Y,_C1) -|
    (* digit(N), N=\=Y, N=\=E, N=\=D *)
    digit _N -| diff _N _Y -| diff _N _E -| diff _N _D -|
    (* digit(R), R=\=N, R=\=Y, R=\=E, R=\=D *)
    digit _R -| diff _R _N -| diff _R _Y -| diff _R _E -| diff _R _D -|
    (* sumdigit2(C1,N, R, E, C2), *)
    sumdigit(_C1,_N,_R,_E,_C2) -|
    (* digit(O), O=\=R, O=\=N, O=\=Y, O=\=E, O=\=D *)
    digit _O -| diff _O _R -| diff _O _N -| diff _O _Y -|
         diff _O _E -| diff _O _D -|
    (* sumdigit2(C2, E, O, N, C3) *)
    sumdigit(_C2, _E, _O, _N, _C3) -|
    (* leftdigit(S), S=\=O, S=\=R, S=\=N, S=\=Y, S=\=E, S=\=D *)
    leftdigit _S -| diff _S _O -| diff _S _R -| diff _S _N -|
              diff _S _Y -| diff _S _E -| diff _S _D -|
    (* leftdigit(M), M=\=S, M=\=O, M=\=R, M=\=N, M=\=Y, M=\=E, M=\=D *)
    leftdigit _M -| diff _M _S -| diff _M _O -| diff _M _R -|
                    diff _M _N -| diff _M _Y -| diff _M _E -|
                    diff _M _D -|
    (* sumdigit(C3,S,M,O,M) *)
    sumdigit(_C3,_S,_M,_O,_M) -|
      (fun k s -> Printf.printf "Values: %s\n" (toString (doSubst s _X)); k s) <|k<|s;;
```

References

[1] NQUEENS in Prolog. http://www.cse.scu.edu/~rdaniels/html/courses/Coen171/NQProlog.htm.

[2] Prolog Benchmarks. http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/pereira.txt.

[3] Prolog Tutorial. http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html.

[4] Racklog, Logic Programming embedding in Racket. http://docs.racket-lang.org/racklog/.

[5] Schelog, Logic Programming embedding in Scheme. http://www.ccs.neu.edu/home/dorai/schelog/schelog.html.

[6] Henry G. Baker, Jr. Shallow Binding Makes Functional Arrays Fast. *ACM SIGPLAN Notices*, 26(8):145–147, August 1991.

[7] M. Carlsson. On Implementing Prolog in Functional Programming. *New Generation Computing*, 2:347–359, 1984.

[8] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In Claudio V. Russo and Derek Dreyer, editors, *Proceedings of the*

ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007, pages 37–46. ACM, 2007.

[9] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 322–336, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Rowan Davies. Logic Programming embedding in F#, lecture slides about Logic Programming. `http://undergraduate.csse.uwa.edu.au/units/CITS3242/`.

[11] Conal Elliott and Frank Pfenning. A Semi-Functional Implementation of a Higher-Order Logic Programming Language. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325. MIT Press, 1991.

[12] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report TR 182, Indiana University Computer Science Department, October 1985.

[13] Christopher T. Haynes. Logic continuations. *The Journal of Logic Programming*, 4(2):157 – 176, 1987.

[14] Frank Pfenning. Lecture Notes, 15-819K Logic Programming. `http://www.cs.cmu.edu/~fp/courses/lp/`, 2006.

[15] Frank Pfenning. Lecture Notes, Difference Lists. `http://www.cs.cmu.edu/~fp/courses/lp/lectures/11-diff.pdf`, 2006.

[16] Silvija Seres and Michael Spivey. Embedding Prolog in Haskell. In *Department of Computer Science, University of Utrecht*, 1999.

[17] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report TN309, SRI International, 1983.