CS421 Unit Project: Prolog in OCaml

Haoyu Wang   Xiaohong Chen

December 13, 2016

1 Overview

This project fulfills our extra-credit workload in CS421 – Programming Languages and Compilers. Among all other projects listed in Elsa's website, we found this one the most interesting and meaningful.

We aim for a Prolog implementation in OCaml. The Prolog that we implemented in this project is a fragment of the real Prolog language, so we refer it to SimProlog. SimProlog is able to reason with facts and rules about (potentially nested) Prolog terms, but do not support reasoning with data (such as integers and booleans) and data structures (such as lists and trees).

2 Implementation

The SimProlog implementation contains four important aspects.

1. A lexer (`lexer.mll`)
2. A parser (`parser.mly`)
3. Main functionality of SimProlog (`main.ml`) including
   - Abstract data structures
   - Unification and substitution
   - Backtracking
4. An interactive interface (`simp.ml`)

We fully covered all aspects that were proposed in our original proposal (which we enclosed in Appendix for reference). In addition, we provide an interactive interface for users to teach SimProlog engine facts and rules and ask her questions.

2.1 Backtracking

We find it necessary to provide more explanation in our backtracking algorithm that uses unification and finds all solutions for a given inquiry.

```
1  let rec solve (qs: term list)
2                (rs: rule list)
3                (rs_togo: rule list)
4                (s: substitution)
5                (sols: substitution list)
6                (k: substitution list -> 'a) =
7    match qs with
8  | [] -> k(s::sols)
9  | q1::remq -> (match rs_togo with
10 |             | [] -> k(sols)
11               | r1::remr ->
12                 let fresh_r = fresh_rule r1 in
13                  (match unify [(q1, fst fresh_r)] with
14                 | None -> solve qs rs remr s sols k
15                 | Some sigma ->
16                   solve (lift_subst_term_list sigma (remq @ (snd fresh_r)))
17                          rs
18                          rs
19                         (subst_compose sigma s)
20                          sols
21                         (fun sols' -> solve qs rs remr s sols' k)))
```

We enclose our words as in-line comments below.

line 3 In the effort to solve the first question in `qs`, we try rules that we have not tried yet in `rs_togo`, a subset of `rs`.

line 4 We record the current substitution, and hope it would become a new solution.

line 5 All solutions that we have found so far.

line 8 If no more question to solve, then pass the current solution together with the ones that we have found `s::sols` to the continuation `k`.

line 10 If there is some question `q1` to solve but no rule to apply (in `rs_togo`), then we fail to find a new solution, and can only pass `sols` to the continuation.

line 14 If the first rule cannot resolve the first question, we try the rest of rules by calling `solve` recursively over the same problem but with `rs_togo` replaced by `remr`.

line 16 Otherwise we resolve the first question and replace it by (if there is any) newly generated questions to the question list, before we recursively call `solve` with a new continuation `k'` that whenever it receives the solution `sol'` for the sub-problem, calls `solve` to restore the previous state, pretty much the same as line 14 except that `sols` are updated with `sols'`.

3 Examples

3.1 Ordering reasoning

A finite lattice (see below for an example) can be represented by a number of facts (of the form $a < b$) and the reflexivity and transitivity rules.

```
a < b < c < d < e
    |
    b < x < y < z < w
            |
            y < m < n < p < q
```

For all twelve $<$'s we define twelve rules (facts) as follows.

```
true .
bleq(a, b) .
bleq(b, c) .
bleq(c, d) .
bleq(d, e) .
bleq(b, x) .
bleq(x, y) .
bleq(y, z) .
bleq(z, w) .
bleq(y, m) .
bleq(m, n) .
bleq(n, p) .
bleq(p, q) .
```

And two additional (conditional) rules for reflexivity and transitivity.

```
leq(X, X) :- true.
leq(X, Y) :- bleq(X, Z) , leq(Z, Y) .
```

Then we can ask SimProlog

```
?- leq(x, What) .
```

It shall answer

```
"I found 8 solutions."
{What -> x}
```

```
{What -> y}
{What -> z}
{What -> w}
{What -> m}
{What -> n}
{What -> p}
{What -> q}
```

3.2 Peano arithmetic reasoning

SimProlog is able to reason about Peano arithmetic system, including a successor function and a predicate `addeq(x,y,z)` which holds iff $x + y = z$.

```
true .
addeq(X, zero, X) :- true .
addeq(X, s(Y), s(Z)) :- addeq(X, Y, Z) .
addeq(zero, Y, Y) :- true .
addeq(s(X), Y, s(Z)) :- addeq(X, Y, Z) .
```

One important thing when one using SimProlog is to make sure his/her rules guarantee termination. SimProlog does not check termination, and it falls into the pitfall of infinite loops if rules are badly designed.

```
?- addeq(s(zero), s(s(zero)), What) .
"OK. I am thinking ..."
"I found 1 solution."
{What -> s(s(s(zero)))}

?- addeq(What, s(zero), s(s(s(zero)))) .
"OK. I am thinking ..."
"I found 1 solution."
{What -> s(s(zero))}
```

Appendix

Original proposal

```
We propose to implement a simple version of Prolog in OCaml.
A program of such a simple version of Prolog language that
we concern in this unit project, as SimProlog we call it ,
consists of the following components.

(1) A number of *declarations* that declare predicates.
(2) A number of *base clauses* that state fact.
(3) One inquiry where free variables are allowed to show up.

A sample of such a SimProlog language looks the follows.
```

```
----------- SimProlog Program Sample Begins -----------

% This is a simple SimProlog program.
% Everything following '%' is a comment.

mother(X, Y) :- parent(X, Y), femail(X)
parent(john, bill)
parent(jane, bill)
femail(jane)
| ?- mother(M, bill)

----------- SimProlog Program Sample Ends -----------

Our unit project will consist of five basic building blocks,
which are divided into either syntax category or semantics
category.  Each of us will be responsible for one category.
The five basic building blocks are (where [xc] stands for
Xiaohong and [wh] stands for Haoyu):
(1)[xc] A grammar for SimProlog language.
(2)[xc] A lexer that consumes streams of characters and recognizes
tokens.
(3)[xc] A parser that consumes streams of tokens and build
ASTs.
(4)[wh] An evaluator that takes ASTs, understands predicate
declarations and facts, and answers the inquiry.  This may
include:  a unification process and a backtracking mechanism.
```

Code

File `lexer.mll`

```
{ open Parser  (* The type token is defined in parser.mli *)
  exception Eof
}

let digit       = ['0'-'9']
let lcase       = ['a'-'z']
let ucase       = ['A'-'Z']
let wchar       = digit | lcase | ucase | '_'
let lword       = lcase wchar*
let uword       = ucase wchar*
let space       = [' ' '\t' '\n']

rule token = parse
| space         { token lexbuf } (* skip over whitespace *)
| ":-"          { CDASH }
```

```
| "?-"          { QDASH }
| '('           { LPAREN }
| ')'           { RPAREN }
| '.'           { DOT }
| ','           { COMMA }
| lword as w    { LWORD w }
| uword as w    { UWORD w }
| eof                   { raise Eof }
```

File `parser.mly`

```
%{ open Main %}

%token <string> LWORD
%token <string> UWORD
%token COMMA CDASH QDASH LPAREN RPAREN DOT

%start main
%type <Main.command> main
%%

main:
  | rule DOT                         { $1 }
  | inquery DOT                      { $1 }

rule:
  | term                             { Rule($1, []) }
  | term CDASH term_list             { Rule($1, $3) }

inquery:
  | QDASH term                       { Inquery $2 }

term:
  | lword                            { ConstTerm $1 }
  | uword                            { VarTerm $1 }
  | lword LPAREN RPAREN              { ComplexTerm($1, []) }
  | lword LPAREN term_list RPAREN    { ComplexTerm($1, $3) }

term_list:
  | term                             { [$1] }
  | term COMMA term_list             { $1::$3 }

lword:
  | LWORD                            { Const $1 }

uword:
```

```
  | UWORD                               { Var $1 }
```

File `main.ml` For unification, see line 60. For backtracking, see line 136.

```
1 open Pervasives
2 open List
3
4 (* Abstract Data Structures *)
5 type constant = Const of string
6 type variable = Var of string
7 type term = ConstTerm of constant
8           | VarTerm of variable
9           | ComplexTerm of constant * term list
10 type rule = term * term list
11 type command = Rule of term * term list
12              | Inquery of term
13              | ClearComm
14              | ShowComm
15
16 (* Unification and Substitution *)
17 type substitution = (variable * term) list
18 let identity_subst : substitution = []
19
20 (* helper functions *)
21 let rec occur (varname: string) (t: term) : bool =
22   match t with
23 | ConstTerm _ -> false
24 | VarTerm (Var s) -> varname = s
25 | ComplexTerm (Const s, ts) -> exists (occur varname) ts
26 ;;
27
28 let rec subst_fun (subst: substitution) (varname: string) =
29   match subst with
30 | [] ->
31   VarTerm (Var varname)
32 | (Var varname', t) :: substs ->
33   if varname' = varname then t else subst_fun substs varname
34 ;;
35
36 let rec lift_subst_term (subst: substitution) (t: term) =
37   match t with
38 | ConstTerm (Const s) -> t
39 | VarTerm (Var varname) -> subst_fun subst varname
40 | ComplexTerm (Const s, ts) ->
41   ComplexTerm (Const s, map (lift_subst_term subst) ts)
42 ;;
```

```

let lift_subst_term_list subst (ts:term list) = map (lift_subst_term subst) ts;;

(* Cited from MPs and MLs. *)
let subst_compose (s2: substitution) (s1: substitution) : substitution =
  (filter (fun (tv,_) -> not(mem_assoc tv s1)) s2) @
  (map (fun (tv,residue) -> (tv, lift_subst_term s2 residue)) s1)

(* Pick the subset of substitution @s that contains variables in @vars. *)
let rec pick_subst (s: substitution) (vars: variable list) : substitution =
  match s with
| [] -> []
| (v, t) :: rem ->
  if mem v vars then (v, t) :: pick_subst rem vars
  else pick_subst rem vars

(* Unification *)
let rec unify (eqlst: (term * term) list) =
let rec addNewEqs ls1 ls2 acc =
  match ls1, ls2 with
| [], [] -> Some acc
| (t1 :: tl1), (t2 :: tl2) -> addNewEqs tl1 tl2 ((t1, t2) :: acc)
| _ -> None
in match eqlst with
       | [] -> Some([])
       | (s,t) :: eqs -> if s = t then unify eqs
         else (match (s, t) with
       | (ComplexTerm ((Const c1), t1), ComplexTerm ((Const c2), t2)) ->
         if c1 = c2 then (match (addNewEqs t1 t2 eqs) with
         | None -> None
         | Some l -> unify l)
         else None
       | (ComplexTerm (c, t), VarTerm (Var v)) ->
         unify ((VarTerm (Var v), ComplexTerm (c, t)) :: eqs)
       | (ConstTerm (Const s), VarTerm (Var v)) ->
         unify ((VarTerm (Var v), ConstTerm (Const s)) :: eqs)
       | (VarTerm (Var v), t) -> if (occur v t) then None
                                 else let eqs' =
          map (fun (t1, t2) ->
          (lift_subst_term [(Var v, t)] t1, lift_subst_term [(Var v, t)] t2))
          eqs
                                 in (match (unify eqs') with
                                   | None -> None
                                   | Some phi ->
          Some (subst_compose [(Var v, lift_subst_term phi t)] phi))
                                   | _ -> None)
```

```
;;


(* Backtracking *)

(* generating fresh variables and rules *)

let (fresh, reset) =
  let nxt = ref 0 in
  let f () = (let r = Var("$" ^ string_of_int !nxt)
  in let _ = nxt := !nxt + 1 in r) in
  let r () = nxt := 0 in (f, r)

(* de-duplicate a list *)
let rec ddup l =
  match l with
| [] -> []
| x::xs -> let dxs = ddup xs in
  if mem x dxs then dxs else x::dxs

let rec collect_variables_in_term (t: term) : variable list=
  match t
with  ConstTerm _ -> []
|  VarTerm v -> [v]
|  ComplexTerm(c, ts) -> collect_variables_in_term_list (ts: term list)
and collect_variables_in_term_list (ts: term list) : variable list =
  match ts
with  [] -> []
|  t::ts -> ddup ((collect_variables_in_term t) @
                  (collect_variables_in_term_list ts))

(* generating a fresh instance of an implicit quantified rule by
replacing all variables by fresh variables *)
(* this prevents incorrect variable capturing *)
let rec fresh_rule (t,ts) : rule =
  let bound_variables = collect_variables_in_term_list (t::ts) in
  let subst = map (fun (v:variable) -> (v, VarTerm(fresh())))
                  bound_variables in
  (lift_subst_term subst t, lift_subst_term_list subst ts)


(* find all solutions for a list of inquery @qs using backtracking.
* @rs_togo is the list of rules that haven't been tried to resolve
* the first inquery. @s is the substitution on-the-fly so fat and
* @sols contains all found solutions.
* @k is the continuation: yes, I use CPS style.
```

```
*)
let rec solve (qs: term list)
              (rs: rule list)
              (rs_togo: rule list)
              (s: substitution)
              (sols: substitution list)
              (k: substitution list -> 'a) =
  match qs with
| [] -> k(s::sols) (* no more question to solve: done and @s is the last solution. *)
| q1::remq -> (match rs_togo with
  | [] -> k(sols)  (* no more rules to go: done and abandon @s. *)
  | r1::remr -> let fresh_r = fresh_rule r1 in
                (match unify [(q1, fst fresh_r)] with
    (* cannot use the first rule_to_go: try the rest rules_to_go. *)
    | None -> solve qs rs remr s sols k
    (* solve the new sub-problems, and after that ... *)
    | Some sigma -> solve (lift_subst_term_list sigma (remq @ (snd fresh_r)))
                          rs
                          rs
                          (subst_compose sigma s)
                          sols
                          (fun nsols -> solve qs rs remr s nsols k)))
 (* collect solutions and continue solving the current problem *)
```