Higer-order Prolog

Matteo Capelletti

July 25, 2010

Abstract

Implementation of a basic Prolog compiler enabling higher order programming.

1 Terms

```
data Term = V Int | T String
      | Abs Int Term | App Term Term
     deriving Eq
```

Terms are basically λ-terms. These are used for predicates, within clauses, and for basic terms, as arguments of predicates. This is useful for higher order programming, where predicates are taken as argument of predicates in other programs. For instance, in the following definition of `map`, `f` is a variable over a predicate defined in another program.

```
map f [] [].
map f (y : ys) (z : zs) :- f y z, map x ys zs.
```

Given the reversible character of Prolog programs we can, to some extent, be able to call the program with `f` uninstantiated, and infer a program (or more, by backtracking) instantiating it, as we will see. For ease of reference we state:

```
type Pred = Term
type Lambda = Term
```

A clause has a head predicate and a list of predicates in the body.

```
data Clause = Pred :- [Pred]
     deriving (Eq,Show)

type Prog = [Clause]
```

For instance the `map` program is expressed (with still some sugaring) as:

```
mapping =
     [
      map f [] [] :- [],
      map f (y : ys) (z : zs) :- [f y z, map x ys zs]
     ]
```

2 Unification

The unification algorithm is Martelli and Montanari's algorithm.

```
unify :: Term -> Term -> Maybe (Term -> Term)
unify t1 t2 = unify' [(t1,t2)] id

unify' :: [(Term,Term)] -> (Term -> Term) -> Maybe (Term -> Term)
unify' [] f = Just f
unify' ((T n,T m):ps) f
    | n==m = unify' ps f
    | otherwise = Nothing
unify' ((App t1 t2,App u1 u2):ps) f =
    unify' ((t1,u1):(t2,u2):ps) f
unify' ((x@(V _),y):ps) f =
       if x'==y'
       then unify' ps f
       else case (x',y') of
             (V _,_) -> unify' ps (sub x' y' . f)
             (_,V _) -> unify' ps (sub y' x' . f)
             (_,_) -> unify' ((x',y'):ps) f
        where
         x' = f x
         y' = f y
unify' ((x,y@(V _)):ps) f =
       if x'==y'
       then unify' ps f
       else case (x',y') of
             (V _,_) -> unify' ps (sub x' y' . f)
             (_,V _) -> unify' ps (sub y' x' . f)
             (_,_) -> unify' ((x',y'):ps) f
        where
         x' = f x
         y' = f y
unify' _ _ = Nothing
```

3 Lambda conversion

Abstraction and application are written as follows.

Abstraction:	`[x,y,z]^t`	$\lambda xyz.t$
Application:	`t $ [u,v,w]`	$(((t\ u)\ v)\ w)$

```
infixr 4 ^
(^) :: [Lambda] -> Lambda -> Lambda
[] ^ t = t
V i:xs ^ t = Abs i (xs ^ t)
```

```
infixl 3 $
($) :: Lambda -> [Lambda] -> Lambda
t $ [] = t
t $ (t':ts) = App t t' $ ts
```

The following conversion procedure does not account for *variable capture*. If this raises problem, one may change the substitution procedure.

```
convert :: Lambda -> Lambda
convert t = convert' t []
   where
    convert' :: Lambda -> [Lambda] -> Lambda
    convert' (Abs i t) [] = Abs i (convert t)
    convert' (Abs i t) (t':ts) = convert' (sub i t' t) ts
            where
             sub :: Int -> Lambda -> Lambda -> Lambda
             sub x _ t@(T _) = t
             sub x t v@(V y)
                    | x==y = t
                    | otherwise = v
             sub x t (App t1 t2) = App (sub x t t1) (sub x t t2)
             sub x t (Abs i t') = Abs i (sub x t t')
    convert' (App t1 t2) ts = convert' t1 (convert t2:ts)
    convert' t ts = t $ ts
```

4 Search procedure

The search procedure consists of two main function. Firstly we define the procedure for looking up suitable clauses from a program, for a given goal. We have the following parameters: `v` is an integer for refreshing the variables in program clauses; `g` is the current goal `g`; clauses of the program have been labeled with the list of variables `vs` occurring in them, for simplifying the refreshing of indices.

```
type InstGoals = (Subs,[Pred])

scan :: Int -> Pred -> Prog' -> (Int, [InstGoals])
scan v _ [] = (v,[])
scan v g ((vs,p :- ps):prog)
    | otherwise =
        let (v',p' :- ps') = refresh (v,vs,p :- ps)
            (v'',gs) = scan v' g prog
        in case unify g p' of
            Just f' -> (v'',(f',ps'):gs)
            Nothing -> (v'',gs)
```

Function `scan` returns a new refresh index and a pair `(f,ps)`, where `f` is the substitution resulting from the unification of `g`, with the head of the rule `p :- ps`, for each clause in the program. By immediately generating all alternatives, we implement *backtracking*.

The evaluation function takes the refresh index `v`, a list of pairs `(f,ps)` where `f` is the substitution obtained in deriving the goals `ps` and a program `prog`. The function `eval'` takes initially `[(id,goals)]`. The elements accumulated in the list `as` are alternatives. This implements backtracking. We work at the same time on two kinds of lists, the *and*-list `ps` and the *or*-list `as`.

```
eval' :: Int -> ORs -> Prog' -> [Subs]
eval' v [] prog = []
eval' v ((f,[]):as)  prog =
    f:eval' v as prog
eval' v ((f,q:qs):as) prog =
    eval' v' (os'++as) prog
     where
      q' = convert q
      (v',os) = scan v q' prog
      os' = [ (f'.f,map f' (gs ++ qs)) | (f',gs) <- os ]
```

We discuss the program clause by clause. Firstly the termination, when the list of lists of goals is empty and we have found all solutions, if any.

```
eval' v [] prog = []
```

If there are no more goals in the *and*-list, we have succeeded and produce as a result the corresponding substitution. We proceed with the alternatives in the *or*-list, implementing backtracking.

```
eval' v ((f,[]):as)  prog =
    f:eval' v as prog
```

Generation of new goals takes place in the last clause. We have a substitution `f` and process the head `q` of `q:qs`, the goals that need to be satisfied (*and*-list). Here conversion of `q` takes place, in case it is not a normal form lambda term. Then we `scan` for suitable rules. We may find several, so that the result of `scan` is a list `os` of alternatives (*or*-list). Now we have the *and*-list `qs` of remaining goals, and the *or*-list `os` of alternatives of the form `(f',gs)`, where `f'` is the substitution resulting from the unification of `q` and some rule's head, and `gs` are the new goals from the rule's body. The new *or* consists of `(f'.f,map f' (gs ++ qs))`, where `map f' (gs ++ qs)` is the new *and*-list and `f'.f` is the resulting substitution.

```
eval' v ((f,q:qs):as) prog =
    eval' v' (os'++as) prog
     where
      q' = convert q
```

```
    (v',os) = scan v q' prog
    os' = [ (f'.f,map f' (gs ++ qs)) | (f',gs) <- os ]
```

The rule we are applying with the results of `scan` is: $(A \vee B) \wedge C \Rightarrow (A \wedge C) \vee (B \wedge C)$.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{q : qs}{q \wedge qs}\ \mathit{and}\text{-list} \qquad \dfrac{}{os}\ \texttt{scan}}{os \wedge qs}\ q := os}{[o_1, \ldots, o_n] \wedge qs}\ os = o_1, \ldots, o_n}{(o_1 \vee \ldots \vee o_n) \wedge qs}\ \mathit{or}\text{-list}}{(o_1 \wedge qs) \vee \ldots \vee (o_n \wedge qs)}\ \text{Distr}}{(o_1^1 \wedge \ldots \wedge o_1^m \wedge qs) \vee \ldots \vee (o_n^1 \wedge \ldots \wedge o_n^l \wedge qs)}\ o_i = o_i^1 \wedge \ldots \wedge o_i^m}{(o_1^1 \wedge \ldots \wedge o_1^m \wedge q_1 \wedge \ldots \wedge q_k) \vee \ldots \vee (o_n^1 \wedge \ldots \wedge o_n^l \wedge q_1 \wedge \ldots \wedge q_k)}\ qs = q_1, \ldots, q_k}{[o_1^1, \ldots, o_1^m, q_1, \ldots, q_k] \vee \ldots \vee [o_n^1, \ldots, o_n^l, q_1, \ldots, q_k]}\ \mathit{and}\text{-list}}{[[o_1^1, \ldots, o_1^m, q_1, \ldots, q_k], \ldots, [o_n^1, \ldots, o_n^l, q_1, \ldots, q_k]]}\ \mathit{or}\text{-list}
$$

Invocation happens through the function `eval` that rerurns all variable instantiations, that is all the backtracking results.

```
eval :: [Pred] -> Prog -> [[(Term, Pred)]]
eval query prog =
    [ [ (V v, f (V v)) |
            v <- nub (freeVars query) ] | f <- fs ]
            where
             prog' = preProc prog
             i = top (top (0,query),prog)
             g = (id,query)
             fs = eval' i [g] prog'
```

5 Basic function

5.1 Booleans

```
true = T "T"
false = T "F"

not x y = T "not" $ [x,y]
and x y z = T "and" $ [x,y,z]
or x y z = T "or" $ [x,y,z]

bool =
    [
     not true false :- [],
```

```
     not false true :- [],

     and true x x :- [],
     and false x false :- [],

     or false x x :- [],
     or true x true :- []
    ]
```

5.2 Natural numbers

```
zero = T "0"
succ x = T "s" $ [x]

num 0 = zero
num (i+1) = T "s" $ [num i]

add x y z = T "add" $ [x,y,z]
mul x y z = T "mul" $ [x,y,z]
exp x y z = T "exp" $ [x,y,z]

addition =
    [
     add zero x x :- [],
     add (succ x) y (succ z) :- [add x y z]
    ]

multiplication =
    addition ++
    [
      mul zero x zero :- [],
      mul (succ y) x z :- [mul y x w, add x w z]
    ]

exponentiation =
    multiplication ++
    [
      exp zero x (succ zero) :- [],
      exp (succ x) y z :- [exp x y w, mul y w z]
    ]

*Run> eval [add x y (num 5)] addition
[[(x,0),(y,5)],[(x,1),(y,4)],[(x,2),(y,3)],[(x,3),(y,2)],[(x,4),(y,1)],[(x,5),(y,0)]]
*Run> eval [mul (num 3) (num 5) x] multiplication
[[(x,15)]]
*Run> eval [mul x (num 3) (num 15)] multiplication
```

```
[[(x,5)]^CInterrupted.
*Run> eval [mul x y (num 15)] multiplication
[[(x,1),(y,15)]^CInterrupted.
```

5.3 Orders

```
gt = T "GT"
lt = T "LT"
eq = T "EQ"
cmp x y z = T "cmp" $ [x,y,z]
x << y = T "<" $ [x,y]
x >> y = T ">" $ [x,y]
x >=< y = T "==" $ [x,y]
x =<< y = T "=<" $ [x,y]
x >>= y = T ">=" $ [x,y]

comparison =
    bool ++
    [
     cmp zero zero eq :- [],
     cmp zero (succ x) lt :- [],
     cmp (succ x) zero gt :- [],
     cmp (succ x) (succ y) z :- [cmp x y z],

     x << y :- [cmp x y lt]],

     x >> y :- [cmp x y gt],

     x >=< y :- [cmp x y eq],

     x =<< y :- [cmp x y lt],
     x =<< y :- [cmp x y eq],

     x >>= y :- [cmp x y gt],
     x >>= y :- [cmp x y eq]
    ]
```

5.4 Lists

```
empty = T "[]"

infixr 4 !
h ! t = T ":" $ [h,t]

mem y x = T "mem" $ [y,x]
app x y z = T "app" $ [x,y,z]
```

```
rev x y = T "rev" $ [x,y]
rev' x y z = T "rev'" $ [x,y,z]

infix 7 !!
z !! (x,y) = T "!!" $ [x,y,z]
lg x y = T "lg" $ [x,y]

membership =
    [
      mem x (x ! y) :- [],
      mem x (z ! y) :- [mem x y]
    ]

concatenation =
    [
      app empty x x :- [],
      app (x ! y) z (x ! w) :- [app y z w]
    ]

reversal =
       [
        rev x y :- [rev' x empty y],
        rev' empty x x :- [],
        rev' (x ! y) w z :- [rev' y (x ! w) z]
       ]

at =
    [
     x !! (x ! y,zero) :- [],
     z !! (x ! y,succ w) :- [z !! (y,w)]
    ]

length =
    [
     lg empty zero :- [],
     lg (x ! y) (succ z) :- [lg y z]
    ]
```

5.5 Higer-order programming

```
infix 3 =:=
x =:= y = T "=:=" $ [x,y]

equality =
    [
     x =:= x :- []
```

```
    ]

refl x y z = T "refl" $ [x,y,z]
symm x y z = T "symm" $ [x,y,z]
tran x y z = T "tran" $ [x,y,z]
rel x y z w = T "rel" $ [x,y,z,w]

relations =
    equality ++
    [
     refl x y z :- [y =:= z, x $ [y,z]],
     symm x y z :- [x $ [y,z], x $ [z,y]],
     tran x y z :- [x $ [y,z]],
     tran x y z :- [x $ [y,y'], tran x y' z],
     rel x y z (T "refl") :- [refl x y z],
     rel x y z (T "symm") :- [symm x y z],
     rel x y z (T "tran") :- [tran x y z]
    ]

map x y z = T "map" $ [x,y,z]

mapping =
        [
         map x empty empty :- [],
         map x (y ! z) (y' ! z') :- [x $ [y,y'], map x z z']
        ]

*Run> eval [map (T "add" $ [num 2]) (rangeList 3 7) x] (mapping ++ addition)
[[(x,[5,6,7,8,9])]]
*Run> eval [map (T "add" $ [num 2]) (rangeList 3 7) (rangeList 5 9)] (mapping ++
                                                                      addition)
[[]]
*Run> eval [map x (rangeList 3 7) (rangeList 5 9)] (mapping ++ addition)
[[(x,(add 2))]]
*Run> eval [map x (rangeList 3 7) y] (mapping ++ addition)
[[(x,(add 0)),(y,[3,4,5,6,7])],
 [(x,(add 1)),(y,[4,5,6,7,8])],
 [(x,(add 2)),(y,[5,6,7,8,9])],
 [(x,(add 3)),(y,[6,7,8,9,10])],
 [(x,(add 4)),(y,[7,8,9,10,11])],
 [(x,(add 5)),(y,[8,9,10,11,12])],...^CInterrupted.
*Run> eval [map x y (rangeList 3 7)] (mapping ++ addition)
[[(x,(add 0)),(y,[3,4,5,6,7])],
 [(x,(add 1)),(y,[2,3,4,5,6])],
 [(x,(add 2)),(y,[1,2,3,4,5])],
 [(x,(add 3)),(y,[0,1,2,3,4])]]
```

```
foldr x y z w = T "foldr" $ [x,y,z,w]

folding =
    [
     foldr x y empty y :- [],
     foldr x y (z ! z') w' :- [foldr x y z' w, x $ [z,w,w']]
    ]

*Run> eval [foldr (T "app") empty (rangeList 0 3 ! rangeList 4 7 !
            rangeList 8 10 ! empty) (rangeList 0 10)] (folding ++ concatenation)
[[]]
*Run> eval [foldr x empty (rangeList 0 3 ! rangeList 4 7 !
                           rangeList 8 10 ! empty) (rangeList 0 10)] (folding ++ concatena
[[(x,app)]]
*Run> eval [foldr x empty (rangeList 0 3 ! rangeList 4 7 !
            rangeList 8 10 ! empty) y] (folding ++ concatenation)
[[(x,app),(y,[0,1,2,3,4,5,6,7,8,9,10])]]

concat x y = T "concat" $ [x,y]

concatenation' =
    folding ++ concatenation ++
    [
     concat x y :- [foldr (T "app") empty x y]
    ]

*Run> eval [concat (rangeList 3 7 ! rangeList 5 9 !
            rangeList 1 4 ! empty)  y] concatenation'
[[(y,[3,4,5,6,7,5,6,7,8,9,1,2,3,4])]]
*Run> eval [concat x (rangeList 3 7)] concatenation'
[[(x,[[3,4,5,6,7]])],[(x,[[3,4,5,6,7],[]])],[(x,[[3,4,5,6],[7]])],[(x,[[3,4,5],[6,7]])],[(
!4,5,6,7]])],[(x,[[],[3,4,5,6,7]])]^CInterrupted.

sum x y = T "sum" $ [x,y]

summation' =
    folding ++ addition ++
    [
     sum x y :- [foldr (T "add") zero x y]
    ]
```

6 Troublesome

```
filter x y z = T "filter" $ [x,y,z]

filtering =
```

```
    [
     filter x empty empty :- [],
     filter x (y ! z) (y ! z') :- [x $ [y,true], filter x z z'],
     filter x (y ! z) z' :- [x $ [y,false], filter x z z']
    ]

all x y z = T "all" $ [x,y,z]
any x y z = T "any" $ [x,y,z]

quantification =
    bool ++
    [
     all x empty true :- [],
     all x (y ! y') z'' :- [x $ [y,z'], all x y' z, and z z' z'']
     ,
     any x empty false :- [],
     any x (y ! y') z'' :- [x $ [y,z'], any x y' z, or z z' z'']
    ]
```

7 Other programs

7.1 All balanced brackets

```
trees x y = T "trees" $ [x,y]

balancedTrees =
      addition ++
      concatenation ++
      [trees zero empty :- [],
       trees (succ x) y :- [add x' x'' x,
                            trees x' z,
                            trees x'' z',
                            app ((<) ! z) ((>) ! z') y]]

*Run> eval [trees (num 4) x] balancedTrees
[[(x1,[<,>,<,>,<,>,<,>])],
 [(x1,[<,>,<,>,<,<,>,>])],
 [(x1,[<,>,<,<,>,>,<,>])],
 [(x1,[<,>,<,<,>,<,>,>])],
 [(x1,[<,>,<,<,<,>,>,>])],
 [(x1,[<,<,>,>,<,>,<,>])],
 [(x1,[<,<,>,>,<,<,>,>])],
 [(x1,[<,<,>,<,>,>,<,>])],
 [(x1,[<,<,<,>,>,>,<,>])],
 [(x1,[<,<,>,<,>,<,>,>])],
```

```
[(x1,[<,<,>,<,<,>,>,>])],
[(x1,[<,<,<,>,>,<,>,>])],
[(x1,[<,<,<,>,<,>,>,>])],
[(x1,[<,<,<,<,>,>,>,>])]]
```

7.2 Combinator normalization

```
k = T "K"

infixl 5 $$
f $$ g = T "$" $ [f,g]

redComb x y = T "red" $ [x,y]
redComb' x y z = T "red" $ [x,y,z]
compose x y z = T "compose" $ [x,y,z]
constant x = T "constant" $ [x]

p11 = [constant a :- [],
       constant b :- [],
       constant c :- [],
       redComb x y :- [redComb' x empty y],
       redComb' (x $$ y) z w :- [redComb' x (y ! z) w],
       redComb' k (x ! z ! y) w :- [redComb' x y w],
       redComb' s (x ! y ! z ! w) w' :- [redComb' x (z ! y $$ z ! w) w'],
       redComb' x y z :- [constant x, compose x y z],
       compose x empty x :- [],
       compose x (y ! z) w :- [redComb y y', compose (x $$ y') z w]]

bCB = s $$ (k $$ s) $$ k
bCB' = s $$ (k $$ (s $$ bCB)) $$ k
cCB = s $$ (bCB $$ bCB $$ s) $$ (k $$ k)

*Run> eval [redComb (bCB $$ a $$ b $$ c) x] p11
[[(x1,(($ a) (($ b) c)))]]
*Run> eval [redComb (bCB' $$ a $$ b $$ c) x] p11
[[(x1,(($ b) (($ a) c)))]]
*Run> eval [redComb (cCB $$ a $$ b $$ c) x] p11
[[(x1,(($ (($ a) c)) b))]]
```

7.3 NL-Proof-net contraction

```
form a c b = T "form" $ [a,c,b]
neg a = T "-" $ [a]
par = T "&"
times = T "*"
```

```
atom x = T "atom" $ [x]
dual x y = T "dual" $ [x,y]
red x y = T "red" $ [x,y]
prove x y = T "prove" $ [x,y]

p14 = [atom a :- [],
       atom b :- [],
       atom c :- [],
       dual par times :- [],
       dual times par :- [],
       dual x (neg x) :- [atom x],
       dual (neg x) x :- [atom x],
       dual (form x y z) (form z' y' x') :- [dual x x',
                                             dual y y',
                                             dual z z'],
       red x x :- [atom x],
       red (neg x) (neg x) :- [atom x],
       red (form x y z) (form x' y z') :- [red x x', red z z'],
       red (form x par z) y :- [red x (form y times w),
                                red z z',
                                dual w z'],
       red (form x par z) y :- [red x x' ,
                                red z (form w times y),
                                dual w x'],
       prove x y :- [red x z, red y z', dual z z']]

lf 0 = b
lf (i+1) = form a par (form (neg a) times (lf i))
```

7.4 DCG

7.4.1 Balanced brackets

```
(<) = T "<"
(>) = T ">"
sen x y = T "S" $ [x,y]
open x y = T "open" $ [x,y]
closed x y = T "closed" $ [x,y]

bal = [ sen x x :- [],
        sen x y :- [open x x', sen x' z, closed z z', sen z' y ],
        open ((<) ! x) x :- [],
        closed ((>) ! x) x :- []]
```

7.4.2 Stacks

```
senStack x y z = T "Sen" $ [x,y,z]
```

```
aLex x y = T "A" $ [x,y]
bLex x y = T "B" $ [x,y]
cLex x y = T "C" $ [x,y]

repl 0 x = id
repl (i+1) x = (T x !) . (repl i x)

abc i = (repl i "a" . repl i "b" . repl i "c") empty

aNbNcN =
      [ sen x y :- [senStack x y empty],
        senStack x x empty :- [],
        senStack x y z :- [aLex x x', senStack x' y (a ! z) ],
        senStack x y (a ! z) :- [bLex x x', senStack x' x'' z, cLex x'' y ],
        aLex (a ! x) x :- [],
        bLex (b ! x) x :- [],
        cLex (c ! x) x :- []]

dupl xs = (ys . ys) empty
    where
     ys = term xs
     term [] = id
     term (a:as) = (T (a:[]) !) . term as

ww =
      [ sen x y :- [senStack x y empty],
        senStack x x empty :- [],
        senStack x y z :- [aLex x x', senStack x' y (a ! z) ],
        senStack x y z :- [bLex x x', senStack x' y (b ! z) ],
        senStack x y (a ! z) :- [senStack x x' z, aLex x' y],
        senStack x y (b ! z) :- [senStack x x' z, bLex x' y],
        aLex (a ! x) x :- [],
        bLex (b ! x) x :- []]
```

7.5 Categorial grammars

7.5.1 Recognition

```
infixr 3 !>
x !> y = T "\\" $ [x,y]

infixl 3 <!
x <! y = T "/" $ [x,y]

rec x y = T "rec" $ [x,y]
rec' x y z w = T "rec'" $ [x,y,z,w]
lex x y = T "lex" $ [x,y]
```

```
cgRecognition =
      length ++ at ++ comparison ++
      [
        rec x z :- [lg x y, rec' x zero y z],
        rec' w x (succ x) z :- [w' !! (w,x),lex w' z],
        rec' w x y z :- [y >> x', x' >> x,
                         rec' w x x' (z <! w'),
                         rec' w x' y w']
      ] ++ lex1
```

7.5.2 Generation

```
gen x y z = T "gen" $ [x,y,z]
gen' x y z w = T "gen'" $ [x,y,z,w]

cgGen =
     concatenation ++ comparison ++
     [
     gen x y z :- [gen' zero x y z],
     gen' x (succ x) (y ! empty) z :- [lex y z],
     gen' x y z w :- [ y >> x', x' >> x,
                       gen' x x' z' (w <! w'),
                       gen' x' y z'' w',
                       app z' z'' z]
    ] ++ lex1

lex1 =
        [
         lex (<) (s <! s <! a <! s) :- [],
         lex (<) (s <! s <! a) :- [],
         lex (<) (s <! a <! s) :- [],
         lex (<) (s <! a) :- [],
         lex (>) a :- []
        ]
```