Contents

1 A generic Prolog compiler

The current Prolog implementation in MBase is not very efficient, it is a non-destructive, pure functional interpreter. This compiler is supposed to provide a more efficient and flexible implementation with easy–to–write drop–in backends.
Compiler idea is simple: Prolog source is translated into an intermediate language which explicitly introduces new variables, explicitly marks choice points and explicitly unifies values. There are two call varieties: `call_with` which calls a Prolog function and passes a remaining continuation to it and a simpler `tailcall` which does not pass any continuation. Only variables and constants are allowed as call arguments, any structured call arguments must be lifted as new variable definitions.
The trickiest part is the `choice` construction which allocates a choice point and makes a list of continuations (choice variants).
Any backend should implement a translation from this intermediate language into something which can be compiled or interpreted. A default backend translates this code into raw MBase featuring explicit continuations, whith a handful of simple runtime function library.
It's a backend's responsibility to provide implementations of `cut/0` and `call/1`.

1.1 Prolog source AST

Source language is simpler than the original MBase Prolog, therefore we'd need a new definition:

```
ast prolog {
  term = term(ident:id, .*term:ts)
       | var(ident:id)
       // Extensions
       | number(number:v)
       | string(string:v)
       | weakset(.*term:ts)
       // Externally defined variable
       // Must be elimnated by a pre-compilation pass
       | evar(ident:id)
       ;
  clause = clause(term:hd, .*term:tl)
       |  query(.*term:tl) // headless clause
       |  dynamic(term:t) // external dynamic entry, database query, etc.
       ;
  clauses is (.*clause:cs);
}
```

1.2 Compiler helper functions

Environment is pretty simple in this compiler. It contains only the locally scoped definitons, in form of a list of pairs: `name:type`, where `type` can either be `'var'` or `'arg'`. We do not keep a track of any global declarations, hoping that the backend or some post–processing passes will take care of it (e.g., it should be possible to do some inlining and compile–time specialisation).

```
% "Look up a variable in the environment"
function prolog_check_env(env,id)
  do loop(l = env)
    match l with
      (xid:v):tl ↦ if(xid===id) v else loop(tl)
    | else ↦ ∅
```

We have to extract the free variables in order to initialise them explicitly before any unification attempts. In addition to the explicit free variables there will be a list of new implicit variables introduced for the high order "call" arguments.

```
% "Get a list of free variables in a term within a
   given environment"
function prolog_freevars(env, t)
 collector(add,get) {
   iter:prolog(term:t) {
```

```
    deep term {
      var ↦ {
        chk = prolog_check_env(env, id);
        if (not(chk)) add(id)
      } | else ↦ ∅}};
    return unifiq(get()) }
```

A key idea of this compiler is that all the calls are made with trivial arguments only (constants and variables). It means that a structure argument have to be lifted and explicitly unified with a newly allocated variable. A separate optimisation pass (in a backend) can mark such trivial unifications explicitly.

```
% "Compile a call argument; Returns [nenv; intros; arg]"
function prolog_compile_arg(env, t)
  collector(add, get) {
    v = visit:prolog(term:t) {
      deep term {
        term ↦ 'new'(id,@ts)
      | number ↦ 'const'(v)
      | string ↦ 'const'(v)
      | var ↦ {
          chk = prolog_check_env(env, id);
          match chk with
            'arg' ↦ 'arg'(id)
          | 'var' ↦ 'var'(id)
          | ∅ ↦ { add(id); 'var'(id) }
        }}};
    ins = get();
    nenv = (map i in ins do i:'var')⊕env;
    return [nenv; ins; v]}
```

```
% "Compile a single call with possibly high order arguments"
function prolog_lift_call(env, r)
  collector(add, get) {
    nins = prolog_freevars(env, r);[a]
    v = visit:prolog(term:r) {
      once term {
        term ↦ {
          newargs = map a in ts do {
            visit:prolog(term:a) {
              once term {
                var ↦ node
              | number ↦ 'const'(v)
              | string ↦ 'const'(v)
              | term ↦ {
                  nm = gensym();
                  <[x_;x_;cnode]> = prolog_compile_arg(env, node);
                  add([nm;cnode]);
                  return 'var'(nm)
                }}}};
          return mk:node(ts=newargs)
        }
      | var ↦ ccerror('PROLOG_WRONG_CLAUSE'(r))
      }};
    r = get();
    ins = nins ⊕ map [nm;x] in r do nm;[b]
    nenv = (map i in ins do i:'var')⊕env;
    <[x;id;@args]> = v;
    return [nenv; ins; r; id; args]}
```

[a] New variables must be allocated before the call
[b] List of free variables and new variables

```
% "Build a chain of nested unifications"
function prolog_unify_chain(ins, inner)
  do loop(i=ins)
    match i with
      [nm;vl]:tl ↦ 'unify'('var'(nm), vl, loop(tl))
    | else ↦ inner
```

The list of rules in a function "body" is compiled as a sequence of nested function calls. The last term is a tail call (i.e., there is no continuation). If any explicit or implicit free variables occured inside a term, they're lifted into an explicit `'intros'` node and unified before committing a call.

```
% "Compile a prolog function body"
function prolog_compile_body(env0, rules)
  do loop(env = env0, rs = rules)
    match rs with
      [r] ↦ {
       <[nenv; ins; uns; id; args]> = prolog_lift_call(env, r);
        inner = 'tailcall'(id,@args);
        if (ins) 'intros'(ins, prolog_unify_chain(uns, inner)) else inner
      }
    | r:tl ↦ {
       <[nenv; ins; uns; id; args]> = prolog_lift_call(env, r);
        inner = 'call_with'([id;@args], loop(nenv, tl));
        if (ins) 'intros'(ins, prolog_unify_chain(uns, inner)) else inner
      }
    | else ↦ 'proceed'()
```

A clause head introduces function "arguments". High order arguments are simply unified with the argument variable slots, which are expected to be allocated outside of a call (see the flat call rules above).

```
% "Compile clause head"
function prolog_compile_head(aterms, args, tl)
  do aloop(a = aterms, n = args, env=map a in args do a:'arg') {
     match [a;n] with
       [hd:tl; nhd:ntl] ↦ {
         <[nenv;ins;v]> = prolog_compile_arg(env, hd);
          body = 'unify'('arg'(nhd), v, aloop(tl, ntl, nenv));
          if (ins) 'intros'(ins, body) else body
       }
     | else ↦ prolog_compile_body(env, tl)}
```

Some simple helper functions:

```
function prolog_term_arity(t) {<[x;id;@rest]> = t; length(rest) }
function prolog_term_args(t) {<[x;id;@rest]> = t; rest }
function prolog_term_head(t) {<[x;id;@rest]> = t; id }
```

1.3 Compiler top level

Queries are pretty much a backend business, so here we'll provide only the most basic support. Namely, transforming a query into a special function called 'query', with all the free variables lifted into arguments, preserving their names. Backend must detect this name and behave appropriately. A possible behaviour is following: detect 'query' function in a parsed stream, slice everything above it and perform the compilation (may be giving a new unique name to the query function), extract a list of argument names, allocate the variable slots, store them with their corresponding names into a variables environment and then call the query function. Dismiss results with a backend's failure function to get more alternative answers. Use the variables environment for pretty–printing the results.

```
% "Sanitise queries into special predicates,
   expect only ONE query"
function prolog_sanitise_queries(cs, qname, qargsref)
```

```
  visit:prolog(clauses:cs) {
    once clause {
      query ↦ {
        fv = unifiq(foldl(λ(fv, t) {
              prolog_freevars(∅,t)⊕fv
            }, ∅, tl));
        qargsref := fv;
        'clause'('term'(qname,@map f in fv do 'var'(f)),@tl)
      }
    | else ↦ node}}
```

Once clauses are prepared for compilation, their arguments are given consequent simple names and heads and bodies are compiled. Multiple clauses for the same predicate are going into a single `'choose'` node.

```
% "Compile a single clause (should go into 'choose' sequence)"
function prolog_compile_clause(c)
  visit:prolog(clause:c) {
    once clause { clause ↦ {
      nargs = prolog_term_arity(hd);
      args = map i in [0..nargs] do %Sm<<('__A',i);[a]
      aterms = prolog_term_args(hd);
      id = prolog_term_head(hd);
      return (id:args) : prolog_compile_head(aterms, args, tl)
    }}}
```

[a] Arguments are named as $\_\_A_i$, just for convenience.

Multiple clauses are compiled into a single prolog function with a 'choose' group inside. Interpretation of such node is up to backend - e.g., it can be compiled into a list of stored continuations.

```
% "Compile group of the same predicates"
function prolog_compile_clauses(cs)
{  cls = map c in cs do prolog_compile_clause(c);
   body = (match cls with
             [a:one] ↦ one
           | hd:more ↦ 'choose'(@map (a:cx) in cls do cx));
  <id:args> = caar(cls);
   return 'define_prolog_function'(id,args,body)}
```

Dynamic entries are expected to be served by the backend:

```
% "Compile a list of dynamic entries"
function prolog_compile_dynamics(ts)
  map t in ts do {
    visit:prolog(term:t) {
      deep term {
        term ↦ 'define_prolog_dynamic_function'(id, ts)
      | var ↦ id
      | else ↦ ccerror('ARG'(node))
      }}}
```

Predicates in the source stream can be given with an arbitrary order, so we have to group the same predicates together and then compile.

```
% "Group the predicates together for compilation"
function prolog_compile(cs)
collector(add_dyn, get_dyn) {
  groups = mkhash();
  add(id, v) = {
    prev = ohashget(groups,id);
    ohashput(groups, id, v : prev)
  };
```

```
  iter:prolog(clauses:cs) { once clause {
      clause ↦ add(prolog_term_head(hd), node)
    | dynamic ↦ add_dyn(t)
  }};
  dyns = prolog_compile_dynamics(get_dyn());
  return dyns ⊕
    hashmap(
      λ(id, gr) { prolog_compile_clauses(reverse(gr)) },
      groups)
}
```

1.4 Compiled backend AST

Now, the formal specification for the language generated by the above compiler is following:

```
ast pbackend {
  top = define_prolog_function(ident:id, *ident:args, expr:body)
      | define_prolog_dynamic_function(ident:id, *ident:args)
      ;
  tops is (.*top:ts);
  expr = intros(*ident:vars, expr:body)
       | unify(ctor:a, ctor:b, expr:tr)
       | choose(.*expr:es)
       | call_with(calltgt:t, expr:cnt)
       | tailcall(.calltgt:t)
       | proceed()
       ;
  calltgt is (ident:fn, .*callarg:a);
  callarg = var(ident:id) | arg(ident:id) | const(any:v);
  ctor = new(ident:id, .*ctor:args)
       | const(any:v)
       | var(ident:id)
       | arg(ident:id)
       ;
}
```

2 Parser frontend

```
parser prologlex (pfront) {
  @@CapLetter ⇐ [A-Z];
  @@LLetter ⇐ [a-z];
  @tkvarident ⇐ [CapLetter] [IdentRest]*;
  @tkconstident ⇐ [LLetter] [IdentRest]*;
  varident ⇐ [tkvarident]:v ⇒ {ctoken=ident} $sval(v);
  constident ⇐ [tkconstident]:v ⇒ {ctoken=keyword} $sval(v);
}
```

```
parser prolog (prologlex) {
  !!Spaces;
  [lexical:] ⇐ [lexical] ⇒ {ctoken = lexic};
  [keyword:] ⇐ [keyword] ![IdentRest] ⇒ {ctoken=keyword};

  prolog ⇐ slist<[plgclause]>:cs [Spaces]* ⇒ cs;


  &plgclause_start; &plgclause_end; &plgfact_start; &plgfact_end;
```

```
  plgclause "Prolog clause"
        ⇐
          // syntax extensions:
          { dynamic [plgterm]:t "." ⇒ dynamic(t) }
        / [plgclause_start]
          // proper Prolog:
        / { [plgterm]:t ":-" cslist<[plgterm],","|>:bs "." ⇒
              clause(t,@bs) }
        / { [plgterm]:t "." ⇒ clause(t) }
        / { "?" cslist<[plgterm],","|>:ts "." ⇒ query(@ts) }
        / [plgclause_end]
        ;
  plgterm "Prolog binary term"
      ⇐ { [plgfact]:l "=" [plgterm]:r ⇒
            term(‘equals,l,r) }
      / [plgfact]
      ;
```

Non-binary terms:

```
  plgfact "Prolog term"
      ⇐
        [plgfact_start]
      / { [constident]:id "(" cslist<[plgterm],","|>:args ")" ⇒
            term(id,@args) }
      / { [constident]:id ⇒ term(id) }
      / { [varident]:id ⇒ var(id) }
      / { "$" [varident]:id ⇒ evar(id) }
      / { "[" [plgterm]:hd "|" [plgterm]:tl "]" ⇒
            term(‘cons,hd,tl) }
      / { "[" [plglist]:ls "]" ⇒ ls }
      / { "[" "]" ⇒ term(‘nil) }
      / { "!" ⇒ term(‘cut) }
      / { [number]:n ⇒ number(n) }
      / { [string]:s ⇒ string(s) }
      / [plgfact_end]
      ;
```

Inner list representation:

```
  plglist "Prolog list node"
      ⇐ { [plgterm]:hd "," [plglist]:tl ⇒
            term(‘cons, hd, tl) }
      / { [plgterm]:hd ⇒ term(‘cons, hd, term(‘nil)) }
      ;
}
```

Terms produced by this parser will lack arity information, so we'll need this simple pass to fix it:

```
function prolog_parse_fix_arity(t)
  visit:prolog(term:t) { deep term {
      term ↦ mk:node(id=%Sm<<(id,"/", length(ts)))
    | else ↦ node }}
function prolog_parse_fix_arity_clause(c)
  visit:prolog(clause:c) {
```

```
    once term : ∀ prolog_parse_fix_arity(node) }
```

2.1 Pretty–printer

```
parser prolog_strip (prologlex) {
  prolog_strip ⇐ [constident]:id [rest]? [Spaces]* ⇒ id;
  @@rest ⇐ "/" [0-9]+;
}

% "Remove '/N' suffix from prolog term id"
function prolog_strip_id(id)
  parse %S<<(id) as prolog_strip
```

```
% "Pretty-print a prolog term into a string"
function prolog_term_to_string(t)
  visit:prolog(term:t) {
    deep term {
       term ↦ (
         match id with
           'cons/2' ↦ %S<<("[",car(ts),"|",cadr(ts),"]")
         | 'nil/0' ↦ "[]"
         | 'cut/0' ↦ "!"
         | else ↦
             if (ts)
               %S<<(prolog_strip_id(id),"(",
                    strinterleave(ts,", "),
                  ")")
             else prolog_strip_id(id))
     | number ↦ %S<<(v)
     | string ↦ v
     | weakset ↦ %S<<("weakset(",strinterleave(ts,", "),")")
     | var ↦ %S<<(id)}}
```

```
% "Pretty-print a prolog clause or a query into a string"
function prolog_clause_to_string(cl)
  visit:prolog(clause:cl) {
     deep clause { clause ↦
                   if(tl) %S<<(hd, " :- ",
                     strinterleave(tl, ", "), ".")
                   else %S<<(hd, ".")
                 | query ↦ %S<<("?: ",
                     strinterleave(tl, ", "), ".")};
     once term : ∀ prolog_term_to_string(node);
  }
```

3 A reference (inefficient) backend implementation

```
function prolog_backend_dynamic_wrapper(id, args)
{
  return ['_prolog_get_dynamic_function'(
              'quote'(%Sm<<("_prolog_function_",id)));
          'prolog_environment';'prolog_continuation';@args]
}

function prolog_codegen(b)
```

```
visit:pbackend(top:b) {
  deep ctor {
    new ↦ 'prolog_alloc_struct'('prolog_environment',
                          'quote'(id),'list'(@args))
  | const ↦ 'prolog_const'(v)
  | var ↦ id
  | arg ↦ id
  };
  deep expr {
    intros ↦ 'let'(map v in vars do
                  [v;'prolog_alloc_cell'('prolog_environment')],
                body)
  | unify ↦ 'if_prolog_unify'(a,b,tr)
  | choose ↦
     'prolog_choice_point'('prolog_environment',
                      'prolog_continuation',
                      'list'(@map e in es do
                           'fun'(['prolog_environment';
                                  'prolog_continuation'],
                                 e)))
  | call_with ↦ t('fun'(['prolog_environment'],cnt))
  | tailcall ↦ t('prolog_continuation')
  | proceed ↦ '_prolog_call_wrapper'('prolog_continuation'('prolog_environment'))
  };
  deep calltgt: λ(next) {
    match fn with
      'equals/2' ↦ {
        match next with
          'fun'(x, cnt) ↦
             'if_prolog_unify'(@a,cnt)
        | else ↦ 'if_prolog_unify'(@a, 'prolog_continuation'('prolog_environment'))}
    | else ↦
       [ '_prolog_call_function'; %Sm<<("_prolog_function_", fn);
         'prolog_environment'; next ;@a ]
  };
  deep callarg {
    var ↦ id
  | arg ↦ id
  | const ↦ 'prolog_const'(v)
  };
  deep top {
    define_prolog_function ↦
      '_prolog_toplevl_function'(%Sm<<("_prolog_function_",id),
                ['prolog_environment';
                 'prolog_continuation';@args],
                body)
  | define_prolog_dynamic_function ↦
      '_prolog_toplevl_function'(%Sm<<("_prolog_function_",id),
                ['prolog_environment';
                 'prolog_continuation';@args],
              prolog_backend_dynamic_wrapper(id,args))
  }}
```

3.1 Backend functions implementation

```
function %_prolog_call_function_cps(v)
  match v with
   [id;fnm;@fargs] ↦
     if(shashget(getfuncenv(), fnm))
         #'(cons 'CPS (fun () (,fnm ,@fargs)))
     else #'(let ((ff (ohashget *PROLOGENV* ',fnm)))
              (cons 'CPS (fun () (ff ,@fargs))))
```

```
define PROLOG_GLOBENV = mkref(mkhash())

function %_prolog_get_dynamic_function(id)
  ohashget(^PROLOG_GLOBENV, id)

function %_prolog_dynamic_toplevl_function(args0) {
 <[_;nm;args;@rest]> = args0;
 #'(letrec ((,nm (fun ,args (begin ,@rest))))
     (ohashput (deref PROLOG_GLOBENV) (quote ,nm) ,nm))}

function prolog_dynamic(code)
 %read-compile-eval(
   #'(with-macros ((_prolog_toplevl_function _prolog_dynamic_toplevl_function))
       ,code))

function %_prolog_toplevl_function_cps(v)
 match v with
  [id;nm;args;@body] ↦ #'(ohashput *PROLOGENV* ',nm (fun ,args ,@body))
function %_prolog_failure_cps(v)
 match v with
  [id;@rest] ↦ #'(cons 'CPS (fun () (prolog_failure_f ,@rest)))
function %_prolog_call_wrapper_cps(v)
 match v with
  [id;c] ↦ #'(cons 'CPS (fun () ,c))
```

```
#(function _prolog_minicps (tmpenv v)
  (let* ((code
          '(lambda (*PROLOGENV*)
             (with-macros ((_prolog_call_function _prolog_call_function_cps)
                           (prolog_failure _prolog_failure_cps)
                           (_prolog_call_wrapper _prolog_call_wrapper_cps)
                           (_prolog_toplevl_function _prolog_toplevl_function_cps))
               ,v)))
         (ccode (read-int-eval code))
         (ecode (ccode tmpenv)))
    ecode))
```

```
#(macro prolog_failure rest '(prolog_failure_f ,@rest))
#(macro _prolog_call_function rest rest)
#(macro _prolog_call_wrapper (rest) rest)
#(macro _prolog_toplevl_function (nm args . rest)
  '(recfunction ,nm ,args ,@rest))
```

```
function prolog_remember_cell(env, cell) {
  chp = prolog_last_chpoint(env);
  if(chp && ^chp) {
    <[env;prev;cont;cnts;vars]> = ^chp;
    vars := cell : ^vars;
  }
}
function prolog_alloc_cell(env) noconst(cons('var',∅))
function prolog_alloc_struct(env, id, args) { return ['str';id;@args] }
function prolog_const(c) cons('const',c)
function prolog_weak(v) cons('weak',[v])
function prolog_empty_weak() noconst(cons('weak',∅))
function prolog_save_chpoint(env, c) { %set-cdr!(env, c)}
function prolog_last_chpoint(env) { cdr(env) }
```

```
function prolog_deref(v) // Fall through constraint
  match v with
    'ref':x ↦ prolog_deref(x)
  | 'constraint':x:c ↦ prolog_deref(x)
  | else ↦ v
function prolog_deref_c(v) // Stop at constraint
  match v with
    'ref':x ↦ prolog_deref_c(x)
  | else ↦ v
function prolog_set_failguard(env, fg) ohashput(car(env), ' failguard ', fg)
function prolog_fail_guard(env) ohashget(car(env), ' failguard ')

function prolog_failure_f(env) {
   failguard = prolog_fail_guard(env);
   if (failguard) failguard(env);
   chp =  deref(prolog_last_chpoint(env));
   if(chp && ^chp) {
     <[env;prev;cont;cnts;vars]> = chp;
      iter v in ^vars do {
         %set-car!(v,'var');%set-cdr!(v,∅);
      }; vars := ∅;
      prolog_next_choice_point(prolog_last_chpoint(env))
   }
}
function prolog_query_wrapper(v) {
  match v with
    'CPS':n ↦ prolog_query_wrapper(n())
  | else ↦ v
}
function prolog_next_choice_point(chp)
{
  <[env;prev;cont;cnts;vars]> = deref(chp);
   if(cnts) {
     chp := [env;prev;cont;cdr(cnts);vars];
     (car(cnts))(env, cont);
   } else if (prev) { prolog_save_chpoint(env, prev); prolog_failure_f(env); }
   else ∅
}
function prolog_choice_point(env, cont, cnts) {
  prevchp = prolog_last_chpoint(env);
  chp = mkref([env;prevchp;cont;cnts;mkref(∅)]);
  prolog_save_chpoint(env, chp);
  prolog_next_choice_point(chp);
}
function prolog_set_var(env, dst, src) {
   prolog_remember_cell(env,dst);
   %set-car!(dst,'ref'); %set-cdr!(dst, src);
}
function prolog_unify_structs(env, a, b) {
  <[str;ida;@argsa]> = a;
  <[str;idb;@argsb]> = b;
  if (ida === idb) {
    do loop (aa = argsa, ab = argsb) {
      match [aa;ab] with
        [(hd:tl);(hdb:tlb)] ↦
            if (prolog_unify_inner(env, hd, hdb)) loop(tl, tlb)
            else ∅
      | else ↦ true
    }
  } else ∅
}
function prolog_fuse_weaklings(a, b) {
  //TODO!
```

```
  return a⊕b}

function prolog_merge_weaklings(env, a, b) {
  bodya = cdr(a); bodyb= cdr(b);
  nw = cons('weak',prolog_fuse_weaklings(bodya, bodyb));
  %set-car!(a, 'ref');
  %set-car!(b, 'ref');
  %set-cdr!(a, nw);
  %set-cdr!(b, nw);
}

function prolog_is_var(a) match a with 'var':x ↦ true | else ↦ ∅
function prolog_is_struct(a) match a with 'str':x ↦ true | else ↦ ∅
function prolog_is_const(a)  match a with 'const':x ↦ true | else ↦ ∅
function prolog_is_weak(a) match a with 'weak':x ↦ true | else ↦ ∅
function prolog_unify_inner(env, a, b) {
   da = prolog_deref(a); db = prolog_deref(b);
   if (da === db) true
   else if (prolog_is_var(da)) {prolog_set_var(env, da, db);true}
   else if (prolog_is_var(db)) {prolog_set_var(env, db, da);true}
   else if (prolog_is_struct(da) && prolog_is_struct(db)) {
     prolog_unify_structs(env, da,db)
   } else if (prolog_is_const(da) && prolog_is_const(db)) {
     cdr(da) == cdr(db)
   } else if (prolog_is_weak(da) && prolog_is_weak(db)) {
     prolog_merge_weaklings(env, da, db); // fuse both into one
     return true // weak entries always unify
   } else ∅
}
function prolog_unify(env, a, b) {
  ret = prolog_unify_inner(env, a,b);
  return ret;}

macro if_prolog_unify(a, b, body) {
  #'(if (prolog_unify prolog_environment ,a ,b)
        ,body
        (prolog_failure prolog_environment))}

% "Translate the backend internal term representation
   into the source AST form"
function prolog_decode_result_inner(varenv, v, vis) {
  dv = prolog_deref(v);
 <[vish;doneh;visa]> = vis;
  chk = ohashget(vish, dv);
  if(chk) {
    match chk with
      'none'() ↦ {
          chk1 = ohashget(varenv, dv);
          if (chk1) {
            ohashput(vish, dv, 'var'(chk1));
            return 'var'(chk1)
          } else symbols(newref0) {
             newref = %Sm<<('UB_', newref0);
             ohashput(vish, dv, 'var'(newref));
             ohashput(varenv, dv, newref);
             visa(newref, dv);
             return 'var'(newref)}}
    | 'var'(x) ↦ return chk
  } else {
   ohashput(vish, dv, #'(none));
   ret = {
    match dv with
      str(id,@args) ↦ 'term'(id,@map a in args do
```

```
                                prolog_decode_result_inner(varenv, a, vis))
    | 'const':v ↦ if(%string?(v)) 'string'(v) else 'number'(v)
    | 'var':x ↦ { chk = ohashget(varenv, dv);
          if (chk) 'var'(chk) else {
             nm = gensym(); ohashput(varenv, dv, nm);
             return 'var'(nm)}
        }
    | 'weak':l ↦ 'weakset'(@map x in l do prolog_decode_result_inner(varenv, x, vis))
    | else ↦ ∅ // error?
  };
  ohashput(doneh, dv, ret);
  return ret}}
function prolog_decode_result(varenv, v, addback) {
  vish = mkhash();doneh = mkhash();visrev=mkhash();
  visa(nm,v) = { ohashput(visrev, nm, v) };
  ret1 = prolog_decode_result_inner(varenv, v, [vish;doneh;visa]);
  backrefs = hashmap(λ(k,v)[k;v], visrev);
  if(backrefs)
    iter [nm;v] in backrefs do {
      d = ohashget(doneh, v);
      addback(nm, d)};
  return ret1;
}
function prolog_default_continuation(env) env
```

3.2 Prolog core functions

```
function %_prolog_function_cut/0 (env, cnt) {
    chp =  deref(prolog_last_chpoint(env));
    if(chp) {
      <[env;prev;cont;cnts;vars]> = chp;
       if (prev) {
         <[penv;pprev;pcont;pcnts;pvars]> = deref(prev);
          pvars := ^vars ⊕ ^pvars;
       };
       prolog_save_chpoint(env, prev);
    };
    cnt(env)}
```

```
function %_prolog_function_call/1 (env, cnt, term) {
   dt = prolog_deref(term);
   if (not(prolog_is_struct(dt))) {
      prolog_failure(env)
   } else {
     <[str;id;@args]> = dt;
      nm = %Sm<<("_prolog_function_", id);
      fn = shashget(getfuncenv(), nm);
      if (fn) {
        apply(fn, [env; cnt; @args]);
      } else prolog_failure(env)}}
```

```
function %_prolog_make_isfun (op)
 λ(env, cnt, dst, a, b) {
  da = prolog_deref(a); db = prolog_deref(b);
  dd = prolog_deref(dst);
  if (prolog_is_const(da) && prolog_is_const(db) && prolog_is_var(dd)) {
    ca = cdr(da); cb = cdr(db);
    if (prolog_unify(env, dst, prolog_const(op(ca, cb))))
      cnt(env)
```

```
    else prolog_failure(env)
  } else prolog_failure(env);}
```

```
define %_prolog_function_isadd/3 =
  %_prolog_make_isfun (λ(a,b) a+b)
define %_prolog_function_issub/3 =
  %_prolog_make_isfun (λ(a,b) a-b)
define %_prolog_function_ismul/3 =
  %_prolog_make_isfun (λ(a,b) a*b)
define %_prolog_function_isdiv/3 =
  %_prolog_make_isfun (λ(a,b) a/b)
```

```
function %_prolog_make_cmp (op)
  λ(env, cnt, a, b) {
    da = prolog_deref(a); db = prolog_deref(b);
    if (prolog_is_const(da) && prolog_is_const(db)) {
      ca = cdr(da); cb = cdr(db);
      if (op(ca, cb)) cnt(env) else prolog_failure(env)
    } else prolog_failure(env)}
```

```
define %_prolog_function_gr/2 =
  %_prolog_make_cmp (λ(a, b) a>b)
define %_prolog_function_lt/2 =
  %_prolog_make_cmp (λ(a, b) a<b)
define %_prolog_function_ge/2 =
  %_prolog_make_cmp (λ(a, b) a>=b)
define %_prolog_function_le/2 =
  %_prolog_make_cmp (λ(a, b) a<=b)
```

```
function %_prolog_function_fail/0 (env, cnt)
  prolog_failure(env)
```

```
function %_prolog_function_raise/2 (env, cnt, msg, src)
{ dm = prolog_deref(msg); ds = prolog_deref(src);
  ds1 = prolog_term_to_string(prolog_decode_result(mkhash(), ds, λ(k,v)∅));
  ccerror('prolog_message'(dm, ds, ds1))}
```

```
function %_prolog_function_debug/1 (env, cnt, v) {
  writeline(v);
  cnt(env);
}
```

```
function %_prolog_function_print/1 (env, cnt, v) {
  println(prolog_term_to_string(prolog_decode_result(mkhash(), v, λ(k,v)∅)));
  cnt(env);
}
```

```
function %_prolog_function_weak/2 (env, cnt, dst, v) {
  dd = prolog_deref(dst);
  if (prolog_is_var(dd)) {
    if(prolog_unify(env, dst, prolog_weak(v)))
      cnt(env)
    else prolog_failure(env)
  } else prolog_failure(env)}
```

```
function %_prolog_function_weak/1 (env, cnt, dst) {
  dd = prolog_deref(dst);
  if (prolog_is_var(dd)) {
    if(prolog_unify(env, dst, prolog_empty_weak()))
      cnt(env)
    else prolog_failure(env)
  } else prolog_failure(env)}
```

```
function list_to_prolog(env, lst) {
   do loop (l = lst) {
     match l with
       hd: tl ↦ {
         ltl = loop(tl);
         prolog_alloc_struct(env, 'cons/2', [hd; ltl])}
     | ∅ ↦ prolog_alloc_struct(env, 'nil/0', ∅)}}
```

```
function %_prolog_function_weak_to_list/2(env, cnt, wn, dst) {
  d1 = prolog_deref(wn);
  d2 = prolog_deref(dst);
  match d1 with
    weak(@lst) ↦ {
      l1 = list_to_prolog(env, lst);
      prolog_unify(env, dst, l1);
      cnt(env)}
  | else ↦ prolog_failure(env)}
```

```
function prolog_to_list(env, l0) {
   do loop(l = l0) {
     x = prolog_deref(l);
     match x with
       str('cons/2', hd0, tl0) ↦ hd0 : loop(tl0)
     | str('nil/0') ↦ ∅
     | else ↦ ccerror('PROLOG_TO_LIST')}}
```

```
function %_prolog_function_weak_reset/2(env, cnt, wn, src) {
  d1 = prolog_deref(wn);
  d2 = prolog_deref(src);
  match d2 with
    weak(@x) ↦ {
      nw = cons('weak', prolog_to_list(env, d1));
      %set-car!(d2, 'ref');
      %set-cdr!(d2, nw);
      cnt(env)}
  | else ↦ prolog_failure(env)}
```

3.3 Backend-specific query compilation

```
function prolog_backend_driver_generic(parsed, exec, getexec, failure, outp) {
  flush(c, queryp, prev) = symbols(qname) if(c) {
    cc = reverse(c);
    app0 = swbenchmark0("plg parse", map a in cc do prolog_parse_fix_arity_clause(a));
    qstat = mkref(∅);
    app = swbenchmark0("plg sanitise", prolog_sanitise_queries(app0, qname, qstat));
    capp = swbenchmark0("plg compile", prolog_compile(app));
    if (shashget(getfuncenv(), 'debug-prolog-compile'))
      writeline('PROLOG_COMPILED'(@capp));
    cgen = swbenchmark0("plg codegen", %map(prolog_codegen, capp));
    if (shashget(getfuncenv(), 'debug-prolog-codegen'))
      writeline('PROLOG_CODEGEN'(@cgen));
    swbenchmark0("plg cgexec",exec('begin'(@cgen)));

    if (queryp) {
      qargs = deref(qstat);
      venv = mkhash();
      vars = map q in qargs do {
        v = noconst('var':∅);
        ohashput(venv, v, q);
        return v};
      qnamescreen = %Sm<<("_prolog_function_",qname);
      envh = mkhash();
      env0 = envh:∅;
      cnt0 = λ(env) env;
      result =
        swbenchmark0("plg exec",prolog_query_wrapper(apply(getexec(qnamescreen), env0:cnt0:vars)));

      if (result) {
        if (outp) {
          do printloop(res = result, resnum = 0) {
            if (res) {
              println(%S<<("Solution N",resnum,":"));
              iter [n;v] in zip(qargs, vars) do {
                nv = prolog_decode_result(venv, v,λ(k,v)∅);
                println(%S<<(n, " = ", prolog_term_to_string(nv)))
              };
              printloop(prolog_query_wrapper(failure(res)), resnum+1)}}
        } else
          do resultloop(res = result, resnum = 0) collector(addb0, getback) {
            if (res) {
              addback(k,v) = addb0([k;v]);
              resmap = swbenchmark0("plg decode", map [n;v] in zip(qargs, vars) do {
                nv = prolog_decode_result(venv, v, addback);
                [n; nv]
              });
              return resnum : (getback()⊕resmap) : λ()
                { resultloop(failure(res), resnum+1) }
            } else return ∅
          }
      } else return ∅} else return prev} else prev;
  do loop(i = parsed, c = ∅, prev = ∅) {
    match i with
      ['query';@x]:rest ↦ {
          nquery = car(i);
          nxt = flush(nquery:c, true, prev); loop(rest, ∅, nxt)}
    | ∅ ↦ flush(c, ∅, prev)
    | hd : tl ↦ loop(tl, hd : c, prev)}}
```

```
function prolog_backend_driver(parsed, exec, outp)
```

```
  prolog_backend_driver_generic(parsed, exec,
      λ(nm) shashget(getfuncenv(), nm), prolog_failure_f, outp)
```

4 PFront integration

```
syntax in top, start (prolog): ' ".prolog" ":" "{" [prolog]:px "}" '
{
  app0 = map a in px do prolog_parse_fix_arity_clause(a);
  capp = prolog_compile(app0);
  cgen = %map(prolog_codegen, capp);
  return 'expr'('lisp'('begin'(@cgen)))
}
```

```
function prolog_sanitise_evars(a)
collector(add, get) {
  ev = mkhash();
  na = visit:prolog(clauses:a) {
    deep term {
      evar ↦ {
        chk = ohashget(ev, id);
        if (chk) ∅ else {ohashput(ev,id,id); add(id)};
        return 'var'(id)
      }
    | else ↦ node
    }
  };
  return na:get()
}
```

```
syntax in expr, start (prolog):
  ' [qident]:nm "<-?" cslist<[plgterm],",">:ts "." '
{
  app0 = [prolog_parse_fix_arity_clause('query'(@ts))];
  qstat = mkref(∅); qname = gensym();
 <app01:evars> = prolog_sanitise_evars(app0);
  app = prolog_sanitise_queries(app01, qname, qstat);
  newvars = %set-substr(^qstat, evars);
  capp = prolog_compile(app);
  cgen = %map(prolog_codegen, capp); // expecting single recfunction
 <['_prolog_toplevl_function'(id,args,body)]> = cgen;
  env0 = gensym();
  lcode =
    ['flatbegin-inside-begin-with-defs';
          ['inblock-def';qname;'fun'(args,body)];
          ['inblock-def';env0;'noconst'('cons'('mkhash'(),'mkref'('nil')))];
         @map v in newvars do {
           ['inblock-def';v;'noconst'('cons'('quote'('var'),'nil'))]
         };
          ['inblock-def';nm;
            ['prolog_loop_results';
                [qname;env0; 'prolog_default_continuation';@(^qstat)];
                ^qstat]]];
  return 'lisp'(lcode)}
```

```
function prolog_loop_results_fun(res,qargs,venv)
  do loop(r = res, i = 0) {
    if (r) {
      nargs = map [nv;@v] in qargs do {
```

```
      [nv; prolog_decode_result(venv, v, λ(k,v)∅)]};
    return (i:venv):nargs:λ() {
      loop(prolog_failure(res), i+1)}
  } else ∅}
```

```
#(macro prolog_loop_results (res args)
  (with-syms (venv qargs)
    `(let ((,venv (mkhash))
           (,qargs (list ,@(foreach-map (v args)
                             `(cons (quote ,v) ,v)))))
       ,@(foreach-map (v args)
           `(ohashput ,venv ,v (quote ,v)))
       (prolog_loop_results_fun ,res ,qargs ,venv))))
```

5 Prolog core library

```
.prolog: {
  //Equality
  X=X.

  // Logic
  and(A,B) :- call(A), call(B).
  or(A,B) :- call(A).
  or(A,B) :- call(B).

  failwith(M, T) :- call(T).
  failwith(M, T) :- raise(M, T).

  // Lists
  append([],L,L).
  append([H|T],L,[H|A]) :- append(T,L,A).
  treclength([],N,N).
  treclength([H|T],L,N) :- isadd(NN,1,N), treclength(T,L,NN).
  length([],0).
  length([H|T], L) :- treclength(T,L,1).

  revert([], []).
  revert([H|L], R) :- revert(L, RL), append(RL,[H],R).

  // Infamous Prolog negation
  negate(X) :- call(X),!,fail.
  negate(X).
  // Sets (list-based)
  in(E, []) :- fail.
  in(E, [E|X]) :- !.
  in(E, [X|Y]) :- in(E,Y).


  setsubelt([E|T], E, T) :- !.
  setsubelt([], E, []).
  setsubelt([H|T], E, [H|R1]) :- setsubelt(T,E,R1).
  setsub(A, [], A).
  setsub(A, [H|T], R) :- setsubelt(A, H, X), setsub(X, T, R).
  setadd(A, B, AB) :- setsub(A, B, X), append(X, B, AB).

  setxor(A, B, XAB) :- setsub(A, B, AB), setsub(B, A, BA),
                       append(AB, BA, XAB).
  setand(A, B, AAB) :- setadd(A, B, AB), setxor(A,B, XAB),
                       setsub(AB, XAB, AAB).
```

```
    unifiqinner([], R, R).
    unifiqinner([H|T], Prev,R) :- in(H,Prev),!,unifiqinner(T,Prev,R).
    unifiqinner([H|T], Prev,R) :- unifiqinner(T,[H|Prev],R).

    unifiq([], []) :- !.
    unifiq(L, R) :- unifiqinner(L,[],R),!.

    /// /// /// Does not work yet:
    /// perms(L, [H|T]) :- setsubelt(L,H,R), perms(R,T).
    /// perms([], []).

    // Assoc lists
    find(Key, [[Key, Value]|Rest], Value) :- !.
    find(Key, [X|Rest], Value) :- find(Key, Rest, Value).

    amod(K, V, [], []) :- !.
    amod(Key, Value, [[Key, OldValue]|Rest], [[Key, Value]|Rest]) :- !.
    amod(Key, Value, [X|Rest], [X|R]) :- amod(Key, Value, Rest, R).

    //Peano
    natural_number(o).
    natural_number(s(N)) :- natural_number(N).

    natural_add(o, N, N).
    natural_add(s(A), B, s(C)) :- natural_add(A,B,C).

    natural_mul(o, N, o).
    natural_mul(s(N), M, P) :-
        natural_mul(N, M, K),
        natural_add(K, M, P).

    natural_gr(s(N),o).
    natural_gr(s(A),s(B)) :- natural_gr(A,B).

    natural_lt(o,s(N)).
    natural_lt(s(A),s(B)) :- natural_lt(A,B).

    natural_max(A,A,A).
    natural_max(A,B,A) :- natural_gr(A,B).
    natural_max(A,B,B) :- natural_gr(B,A).

    natural_min(A,A,A).
    natural_min(A,B,B) :- natural_gr(A,B).
    natural_min(A,B,A) :- natural_gr(B,A).

    natural_to_num(o,0).
    natural_to_num(s(N), I) :- natural_to_num(N, I1), isadd(I, I1, 1).

    num_to_natural(0,o).
    num_to_natural(I, s(N)) :- issub(I1, I, 1), num_to_natural(I1, N).

}
```