Experiments with GHC prototypes

Gérard Huet

ICOT

May 24th, 1988

Abstract

This technical note presents some experiments done by the author during a two-week period at ICOT. The purpose of the experiments was to understand semantic issues concerning GHC, through the design of interpreters realizing various operational semantics of this language. These interpreters have been written in CAML, the version of ML developed at INRIA in the Formel project.

1 An introduction to GHC

GHC was first introduced by K. Ueda in his PhD thesis at University of Tokyo[2]. The language was proposed as a combination of logic programming ideas with imperative programming, well-suited to parallel execution. The basic idea is to replace PROLOG call by unification into a simpler call by matching, and to avoid backtrack. That is, when a clause is chosen (one says that it *commits*), no further choice of a clause pertaining to the corresponding goal is permitted.

A clause commits when its "guard" is successfully executed with the assignment given by pattern-matching the current goal. The guard is an initial list of literals of the clause, separated by a bar from the body proper. For instance, the guarded clause:

$$p(X, S(Y)) :- Y > 0 \quad | \quad q(X)$$

will commit on goal $p(a, S(S(0)))$, with corresponding subgoal $q(a)$ (We use the convention of representing variables with identifiers starting with an upper-case letter). This clause will neither commit on goal $p(a, Z)$, since it does not match its head (although it is unifiable with it), nor will it commit on the goal $p(a, S(0))$, since the corresponding instance of the guard $0 > 0$ is not satisfied.

The first problem that arises is that so far this mecanism permits only functional calls, without even the possibility of returning a result. This possibility is given in GHC by providing a special equality predicate, with $M = N$ meaning that terms M and N must be unified.

The second problem, since we want to avoid backtracking, is that the state (i.e. current goal stack) should not be affected until a clause commits. This is insured for the head, since we only do pattern matching, not full unification. For the literals of the guard, we have a problem, since they must be solved without side-effect. This problem is specially acute when a predicate of the guard is user-defined by clauses. Information concerning protected variables must be carried along the computation. We shall in the rest of this study limit ourselves to the case where only special evaluable predicates occur in the guards (this is usually referred to as *Flat GHC*).

We shall distinguish between two kinds of special predicates: the *meta* predicates, which concern the term structure of the data (for instance, the unification predicate =), and the *evaluable* predicates, which concern base data types such as integers.

2 Pure GHC

We first look at a purely logical version of GHC, without predefined data-types. The only special meta-predicate is =, which stands for unification. The first remark is that a clause with only equations in the guard may be replaced by a clause without guard, since the unifications can be effected at compile-time without changing the semantics. We thus get a very simple "Flat GHC without guards". The equations in the body of clauses serve a dual purpose: They check that certain constraints are satisfied, and they generate output. No backtrack is allowed (rule of commitment), and thus we have a simpler unique-environment language, similar to deterministic PROLOG. However, we relax the left-to-right evaluation order of goals usual in sequential PROLOG implementations by allowing AND parallelism. The resulting language seems well suited to "constraint programming". It is not complete for Horn sentences, since with the program

$$p(X) : -X = 0.$$
$$p(X) : -X = 1.$$

the goal $?P(Y), Y = 1$ will fail.

We now give the complete CAML code of a Pure GHC, or DHC, interpreter.

2.1 Abstract syntax

2.1.1 Terms

```
type term = Var of int
          | Term of string & term list;;
```

We generalize the list iterator `it_list` to terms as a general term traversing functional. The last argument `v` tells what to do to variables.

```
let termtrav f g start v  = travrec
  where rec travrec = function
     Term(oper,sons) -> f(oper,list_it (g o travrec) sons start)
  |  Var(n) -> v(n);;
```

For instance, we can program the usual term traversals algorithms as:

```
let preorder = termtrav (prefix ::) append [] (singleton o string_of_num)
and postorder = termtrav post append [] (singleton o string_of_num)
    where post(x,y) = y@[x];;
```

or we may copy a term by:

```
let copy = termtrav Term cons [] Var;;
```

Here we compute the set of variables of a term by:

```
let vars = termtrav snd union [] singleton;;
```

2.1.2 Substitutions

We shall represent substitutions by association lists: $[(var1, term1); ...; (varN, termN)]$.

```
type subst == (int & term) list;;
```

We use the maximum sharing primitives. That is, we use structure copying, but with maximum sharing of the common subterms.

```
(* look up variable in substitution, raises Identity if not found *)
let look_up x l = assoc x l ? raise Identity;;
```

Similarly, subst raises exception Identity when the term is not affected.

```
(* subst : subst -> term -> term *)
let subst sigma = subst_sigma where rec subst_sigma = function
    Var(n)       -> look_up n sigma
  | Term(f,sons) -> Term(f,map_share subst_sigma sons);;
let substitute sigma = share (subst sigma);;
```

Composition of two substitutions.

```
(* compsubst : subst -> subst -> subst *)
let compsubst subst1 subst2 =
    (map (fun (v,t) -> (v,substitute subst1 t)) subst2) @ subst1;;
```

2.1.3 Pattern-matching

```
exception FAIL;;
```

Matching returns the matching substitution, or raises `FAIL`.

```
let matching = matchrec []
  where rec matchrec subst = function
     (Var v,M) -> (try let N = look_up v subst
                       in if M=N then subst else raise FAIL
                   with Identity -> (v,M)::subst)
   | (Term(op1,sons1),Term(op2,sons2)) ->
  if op1 = op2 then it_pairlist matchrec subst (sons1,sons2)
  else raise FAIL
   | _ -> raise FAIL;;
```

2.1.4 Unification

We use ordinary unification, with occur check.

```
let occur_check ((v,M) as pair) = (v_free M; [pair])
 where rec v_free = function
    Var(n)       -> if n=v then raise FAIL
  | Term(_,sons) -> do_list v_free sons;;
```

```
(* unify : term & term -> subst *)
let rec unify =
 function (Var n1,term2) -> if Var n1 = term2 then []
    else occur_check (n1,term2)
        | (term1,Var n2) -> occur_check (n2,term1)
        | (Term(op1,sons1),Term(op2,sons2)) ->
  if op1 = op2 then
     (it_pairlist unifylist [] (sons1,sons2)
      where unifylist s (t1,t2) =
            compsubst (unify(substitute s t1,substitute s t2)) s)
  else raise FAIL;;

type solution = SUBST of subst
              | NONE;;

let unif pair = try SUBST(unify pair) with FAIL -> NONE;;
```

2.2 Concrete syntax

Here we give some service procedures that compile variable names into integer indexes, and that provide the reverse pretty-printing operations. This section may be skipped, the understanding of the interpreter being independent of these technical parsing/unparsing details.

2.2.1 Parsing

```
(* The names of variables are remembered in a dictionary, here an alist *)
type dict == (string & num) list;;

type map == num list -> term & num list
and  maps == num list -> term list & num list;;

(* collect : maps -> map -> maps *)
let collect (f:maps) (g:map) l0 = let t1,l1 = g l0
                                  in let ts,l = f l1 in (t1::ts),l;;

let mk_terml ((sl,(fl:map list)),(s,(f:map))) = (sl @ s),(f::fl);;

let mk_term (oper,sl,(fl:map list)) = sl,
(fun l0 -> let tl,l = it_list collect (fun l -> [],l) fl l0 in
Term(oper,tl),l);;

(* Variable names start with upper-case letters *)
let var_name string = let n=ascii_code string in n>64 & n<91;;

let Vari n = Var(int_of_num n);;

(* const_or_var : (string -> string list & map) *)
```

```
let const_or_var name = if var_name name then [name],fun (n::l)->Vari(n),l
         else mk_term(name,[],[]);;

let GENSYM = ref 0;;

(* dont'care variables *)
let dontcare () = GENSYM:=!GENSYM+1;
                  ["_" ^ string_of_num !GENSYM],fun (n::l)->Vari(n),l;;

(* numerical constants *)
let mk_num n = mk_term(string_of_num n,[],[]);;

(* Special for lists *)
(* Operators cons and nil are reserved *)

(* map -> map -> map *)
let collect1 (f:map) (g:map) l0 =
   let t1,l1 = f l0 in let t2,l2 = g l1 in Term("cons",[t2;t1]),l2;;

let mk_list ((sl,fl),(s',f')) = s'@sl,
  (fun l0 -> it_list collect1 f' fl l0);;

let mk_infix (oper,t1,t2) = mk_term(oper,mk_terml(mk_terml(([],[]),t1),t2));;

let empty_list = mk_term("nil",[],[]);;

(* map_to_num ["x";"y";"x"] = ([1; 2; 1],["y",2; "x",1],2) *)
let map_to_num l =
let search s (l,(ass,n as pair)) = ((assoc s ass)::l,pair)
? let n'=n+1 in n'::l,(s,n')::ass,n'
in list_it search l ([],[],0);;

(* A goal is a list of terms preceded by a question mark *)
(* It is represented as a clause Answer(x1,...,xn) <- goal *)
let mk_goal (sl,fl) = let il,D,n = map_to_num sl
    in let lt,[] = it_list collect (fun l -> [],l) fl il
       in (Term("Answer",map (Vari o snd) D),lt,n),D;;
```

2.2.2 Unparsing

```
(* New variables are printed as X1, X2, ... *)
let gensymn n = "X" ^ string_of_num n;;
let gensym i = gensymn (num_of_int i);;

let INFIXES = ref ["="];;
```

```
let unparse D M = let Dop = inverse_assoc D in
  let print_var n = print_string (assoc n Dop ? (gensymn n) (* for failures *))
  in  unparserec M;print_newline()
  where rec unparserec = function
Var(n) -> print_var n
     |  Term("nil",[]) -> print_string "[]"
     |  Term("cons",[t;lt]) -> let rec print_tlist = function
              Var(n) -> print_string "|"; print_var n
            | Term("nil",[]) -> ()
            | Term("cons",[t;lt']) -> print_string ","; unparserec t;
                                      print_tlist lt'
            | _ -> failwith "WRONG LIST TERM" in
                         print_string "["; unparserec t; print_tlist lt;
                         print_string "]"
     |  Term(oper,sons) -> if mem oper !INFIXES then
                             (let [t1;t2]=sons in
                              (unparserec t1; print_string oper;
                               unparserec t2)
                             ? failwith ("Infix " ^ oper ^ " wrong arity"))
                           else
                             (print_string oper;
                              match sons with
              []        -> ()
            | (t :: lt) -> print_string "(";
                  unparserec t;
                  do_list (fun t -> print_string ",";unparserec t) lt;
                          print_string ")");;

(* For printing the answer *)
let unparse_answer D (Term("Answer",lt)) =
 let newvars = (subtract (list_it (union o vars) lt []) (map (int_of_num o snd) D)) in
 let newD = union D (num_map (fun n i -> (gensymn n,num_of_int i)) newvars)
 in (let unparseD = unparse newD in
     map (fun ((name,_),term) -> print_string name;print_string " = ";
                           unparseD term;print_newline())
         (combine(D,lt)));();;
```

2.2.3 Clauses

First, let us define the abstract syntax of clauses.

A (definite) clause is composed of a term (its conclusion), a list of terms (its hypotheses), and an integer (the number of variables appearing in all the terms). These variables are assumed to be bound together at the level of the clause.

```
type clause == term & (term list) & num;;
```

Nest, we extend the concrete syntax algorithms. Parsing returns a clause and its variables

dictionary.

```
type concrete_clause == clause & dict;;

let mk_clause ((s,f),(sl,fl)) = let (il,dl,n) = map_to_num (s @ sl)
   in let (h,il') = f il
    in let (t,[]) = it_list collect (fun l -> [],l) fl il'
       in (h,t,n),dl;;

let unparse_clause ((conc,hyps,n),D) =
 let U = unparse D in
  let U' x = (print_string ";";U x) in
(U conc;match hyps with
   [] -> ()
 | (h::t) -> (print_string "<-";U h;map U' t;print_newline()));;
```

2.3 The DHC Interpreter

2.3.1 Accessing programs

Programs, i.e. lists of clauses, are stored in an association list. This is rather naive, we should use a hash-code instead of an association list.

```
type program == clause list
and programs == (string & program) list;;

let PROGRAM = ref ([]:programs);;

let program pred = assoc pred !PROGRAM;;
```

We keep during the computation a renaming index, used to generate new names.

```
let INDEX = ref 0;;
```

2.3.2 Executing programs

We now explain how to try and execute a goal on the relevant program.

```
type execution = SUBGOALS of term list
               | SUSPEND;;

let execute goal = exec where rec exec = function
   []                   -> SUSPEND
 | (head,body,k)::rest -> try
     let sigma = matching (head,goal)
     (* now we rename the remaining variables of the matching clause *)
     (* prim_reci is primitive recursion on int *)
     in let (sigma',n') = prim_reci complete_sigma (sigma,!INDEX) (int_of_num k)
                where complete_sigma ((sigma1,n1) as pair) p =
```

```
                    if mem_assoc p sigma1 then pair
                    else let n2=n1+1 in ((p,Vari n2)::sigma1,n2)
            in (INDEX:=n'; SUBGOALS(share_map (subst sigma') body))
                       with FAIL -> exec rest;;
type result =
   DEADLOCK of term list
 | SOLUTION of term
 | FAILURE  of term&term;;
```

2.3.3 The interpreter loop

We now explain the main loop of our Pure GHC interpreter. We call this formalism DHC, for Deterministic Horn Clauses, since it may be seen as a smart way of executing deterministically Horn Clauses.

The arguments of loop are respectively:

- current is the list of current goal terms
- susp is the list of suspended goal terms
- answer keeps the answer

```
let rec loop answer = loop_goals
  where rec loop_goals (current,susp) = match current with
    [] -> (* we have no more current goals *)
          (match susp with
               [] -> (* no suspended clause waiting *) SOLUTION(answer)
             | _  -> (* deadlock situation *) DEADLOCK(susp))
  | g::rest -> match g with
        Term("=",[t1;t2]) -> (let pair = (t1,t2) in match unif pair with
                 NONE -> FAILURE(pair)
               | SUBST(sigma) -> (let sub = subst sigma in
let wake (awake,asleep) goal = try (((sub goal)::awake),asleep)
                                   with Identity -> (awake,goal::asleep)
(* We wake up a suspended goal whenever the substitution changes it        *)
(* We could try and have execute return information about needed variables *)
in let goals' = it_list wake (share_map sub rest,[]) susp
   in loop (share sub answer) goals'))
      (* Other case : non equality subgoal *)
      | Term(pred,_) -> loop_goals (match execute g (program pred) with
                 SUBGOALS(body) -> (body@rest,susp)
               | SUSPEND        -> (rest,g::susp));;
```

2.3.4 The interpreter top-level

```
let DHC ((ans,goals,index),D) = INDEX:=index;
  match loop ans (goals,[]) with
     SOLUTION(answer) -> (message "Solution found";
```

```
                        unparse_answer D answer)
   | DEADLOCK(susp)   -> (message "Deadlock detected with suspended goals:";
                        do_list (unparse D) susp;print_newline())
   | FAILURE(t,t')    -> (message "Constraint not satisfiable:";
                        unparse D t;
                        message " cannot be unified with ";
                        unparse D t');;
```

Note that we do not really do AND parallelism, but rather a deterministic pseudo parallelism biased left to right. But it is intuitively complete for deterministic programs.

2.3.5 Program management

```
(* Entering programs *)
(* Second argument is complete list of concrete clauses pertaining to
   given predicate, given in reverse order *)
let enter_program name conc_clauses =
if mem_assoc name !PROGRAM
   then failwith ("Program " ^ name ^ " already exists; use Forget")
else (check [] conc_clauses; message ("Program " ^ name ^ " entered"))
     where rec check body = function
       []        -> PROGRAM:=(name,body)::!PROGRAM
     | (cl,_)::rest -> let (Term(pred,_),_) = cl in
                       if pred=name then check (cl::body) rest
                       else failwith ("Some clause does not pertain to " ^ name);;

(* To remove a program *)
let Forget name = PROGRAM:=except_assoc name !PROGRAM;;

(* To remove all programs *)
let Reset () = PROGRAM:=[];;
```

2.4 The parser

2.4.1 The CAML-Yacc file

Here is the Yacc parser file dhc.mly:

```
%mlescape
%token IDENT NUM
%left '='
%%
dhc
   : IDENT '=' program              {enter_program $1 $3}
        | goal                      {DHC $1}
        ;
program : clause '.'                {[$1]}
        | program clause '.'        {$2::$1}
```

```
        ;
clause  : cl                            {mk_clause($1)}
        ;
goal    : '?' tail                      {mk_goal($2)}
        | ':' '-' tail '.'              {mk_goal($3)}
        ;
cl      : term ':' '-' tail             {$1,$4}
        | term                          {$1,([],[])}
        ;
tail    : term                          {mk_terml(([],[]),$1)}
        | tail ',' term                 {mk_terml($1,$3)}
        ;
term    : IDENT                         {const_or_var($1)}
        | '_'                           {dontcare()}
        | NUM                           {mk_num($1)}
        | IDENT terms                   {mk_term($1,$2)}
        | term '=' term                 {mk_infix("=",$1,$3)}
        | '(' term ')'                  {$2}
        | '[' list ']'                  {mk_list $2}
        ;
list    :                               {([],[]),empty_list}
        | nelist '|' term               {$1,$3}
        | nelist                        {$1,empty_list}
        ;
nelist  : term                          {mk_terml(([],[]),$1)}
        | nelist ',' term               {mk_terml($1,$3)}
        ;
terms   : '(' t_list term ')'           {mk_terml($2,$3)}
        ;
t_list  :                               {[],[]}
        | t_list term ','               {mk_terml($1,$2)}
        ;
%%
```

2.4.2 User interface

```
<<p = p(X,Y):-q(X),r(Y).>> constructs the concrete clause CC, with
CC=(clause,dict),
with clause=(head,body,arity),
where head=(Term ("p",[Var #2; Var #1]),
      body=[Term ("q",[Var #2]); Term ("r",[Var #1])],
      arity=2,
and dict=["X",2; "Y",1].
It then calls enter_program "p" clause,
which updates PROGRAM.
```

```
<<?p(a,X)>> constructs the concrete (goal) clause :
(Term ("Answer",[Var 1]),[Term ("p",[Term ("a",[]); Var 1])],1),["X",1]
and calls the interpreter DHC on it.
```

2.5 Examples

2.5.1 Toy examples

```
<<append =
append([],X,V) :- V=X.
append([U|X],Y,[U|Z]) :- append(X,Y,Z).>>;;

<<?append(X,Y,[a,b])>>;;
Deadlock detected with suspended goals:
append(X,Y,[a,b])

Forget "append";;

(* The right definition of append *)
<<append =
append([],X,V) :- V=X.
append([U|X],Y,V) :- append(X,Y,Z) , V=[U|Z].>>;;

<<?append([a,b],[c,d],X)>>;;

Solution found
X = [a,b,c,d]

<<p = p(0,Y) :- q(Y).
      p(s(X),0).
      p(s(X),s(Y)) :- p(X,Y).
>>;;
<<q = q(Y) :- Y=s(Z).
>>;;

<<?p(X,0),q(X)>>;;
Solution found
X = s(X1)
```

Remark: left-to-right deterministic PROLOG would fail in this last example.

2.5.2 Conjecture of Collatz

Conjecture of Collatz : This program stops for every N.

```
while N >= 1 do if even(N) then N := N/2
                           else N := (3N+1)/2
```

This example is due to Ph. Devienne.

```
<<classify = classify(0,X) :-  X = even.
             classify(s(0),X) :-  X = odd.
             classify(s(s(N)),X) :-  classify(N,X).>>;;

<<divide =   divide(0,D) :-  D = 0.
             divide(s(s(N)),D) :-  divide(N,D1) , D = s(D1).>>;;

<<mul =      mul(0,M) :-  M = s(0) .
             mul(s(N),M) :-  mul(N,M1) , M = s(s(s(M1))). >>;;

<<while =    while(s(0),List,X) :-  List = [s(0)] .
             while(N,List,even) :-  divide(N,D), classify(D,X),
                                    List = [N|List1], while(D,List1,X) .
             while(N,List,odd)  :-  mul(N,M), divide(M,D) , classify(D,X),
                                    List = [N|List1], while(D,List1,X) .>>;;

<<collatz =  collatz(N,List) :-  classify(N,X) , while(N,List,X).>>;;

(*
<<?collatz(s(s(s(0))),TRACE)>>;;
Solution found
TRACE = [s(s(s(0))),                          3
         s(s(s(s(s(0))))),                    5
         s(s(s(s(s(s(s(s(0))))))),            8
         s(s(s(s(0)))),                       4
         s(s(0)),                             2
         s(0)]                                1
```

2.5.3 Digital circuits test program

```
<<circuit =
circuit(V1,V2,V3,V4) :- and(V1,V2,X),or(X,V2,Y),or(Y,V4,V3),xor(Y,X,V4).>>;;

<<and = and(1,X,Y) :- X=Y.
        and(X,1,Y) :- X=Y.
        and(0,X,Y) :- Y=0.
        and(X,0,Y) :- Y=0.
        and(X,Y,1) :- X=1,Y=1.>>;;

<<or =  or(1,X,Y) :- Y=1.
        or(X,1,Y) :- Y=1.
        or(0,X,Y) :- X=Y.
        or(X,0,Y) :- X=Y.
        or(X,Y,0) :- X=0,Y=0.>>;;
```

```
<<xor = xor(0,X,Y) :- X=Y.
        xor(X,0,Y) :- X=Y.
        xor(X,Y,0) :- X=Y.
        xor(1,X,Y) :- diff(X,Y).
        xor(X,1,Y) :- diff(X,Y).
        xor(X,Y,1) :- diff(X,Y).>>;;

<<diff = diff(0,X) :- X=1.
         diff(1,X) :- X=0.
         diff(X,0) :- X=1.
         diff(X,1) :- X=0.>>;;

<<?circuit(1,X,1,0)>>;;
Solution found
X = 1

<<?circuit(0,0,X,1)>>;;
Constraint not satisfiable:
0
 cannot be unified with
1

<<?circuit(0,0,X,Y)>>;;
Solution found
X = 0
Y = 0

<<?circuit(1,1,X,Y)>>;;
Solution found
X = 1
Y = 0
```

3 Introducing arithmetic guards

We now move to a fuller GHC, with a predefined data-type of arithmetic. We use CAML arithmetic, with numbers including arbitrary precision integers and rationals, as well as floating point. We provide the special evaluable operators $+, -, *, /$, integer constants, and the evaluable predicates $\#, >, <, =:=$. Finally, the predicate $:=$ evaluates its right hand operand to a number representation, and then unifies it with its left hand operand.

We need an extension of the syntax, as follows.

3.1 Syntax extension

3.1.1 Abstract syntax

A guarded clause is composed of a term (its head), a list of terms (its guard), a list of terms (its body), and an integer (the number of variables appearing in all the terms).

```
type gclause == term & (term list) & (term list) & num;;
```

3.1.2 Concrete syntax

```
type concrete_gclause == gclause & dict;;

let mk_clause ((s,f),(gl,fgl),(sl,fl)) = let (il,dl,n) = map_to_num (s @ gl @ sl)
   in let (h,il') = f il
    in let (g,il'') = it_list collect (fun l -> [],l) fgl il'
     in let (t,[]) = it_list collect (fun l -> [],l) fl il''
       in (h,g,t,n),dl;;

INFIXES := ["=";":=";"#";"=:=";"<";">";"+";"-";"*";"/"];;

let unparse_clause ((conc,guard,hyps,n),D) =
 let U = unparse D in
  let U' x = (print_string ";";U x) in
        (U conc;print_string "<-";
         match guard with
           [] -> ()
         | (h::t) -> (U h;map U' t;print_string"|");
         match hyps with
           [] -> ()
         | (h::t) -> (U h;map U' t;print_newline()));;
```

3.2 The GHC Interpreter

3.2.1 Programs accessing

```
type program == gclause list
and programs == (string & program) list;;

let PROGRAM = ref ([]:programs);;

let program pred = assoc pred !PROGRAM
                   ? failwith ("Undefined predicate: " ^ pred);;

let INDEX = ref 0;; (* renaming counter *)
```

3.2.2 Guard evaluation

```
(* arithmetic evaluation *)
```

```
let rec evaluate t = eval 0 t where rec eval n = function
    Term("0",[]) -> n
  | Term("s",[t]) -> eval (n+1) t
  | Term("+",[t1;t2]) -> let n1 = evaluate t1 in eval (n+n1) t2
  | Term("-",[t1;t2]) -> let n2 = evaluate t2 in eval (n-n2) t1
  | Term("*",[t1;t2]) -> let n1 = evaluate t1 and n2 = evaluate t2 in n+(n1*n2)
  | Term("/",[t1;t2]) -> let n1 = evaluate t1 and n2 = evaluate t2 in n+(n1/n2)
  | Term(s,[]) -> (n+num_of_string s) ? failwith "Illegal arithmetic operand"
  | Term(_) -> failwith "Illegal arithmetic operand"
  | Var(_) -> raise FAIL (* non ground term *);;

(* check guard *)
let check (Term(oper,[t1;t2])) =
    if mem oper [">";"#";"=:=";"<"] then
       let m=evaluate t1 and n=evaluate t2 in
       if (case oper of
           ">" -> m>n
         | "<" -> m<n
         | "#" -> not(m=n)
         | "=:=" -> m=n
          ) then () else raise FAIL
    else failwith ("Illegal operator " ^ oper ^ " in guard");;

type arith =
   NUM of num
 | NONUM;;

let eval t = try NUM(evaluate t) with FAIL -> NONUM;;
```

3.2.3 Program execution

```
type execution = SUBGOALS of term list
               | SUSPEND;;

let execute goal = exec where rec exec = function
   []                          -> SUSPEND
 | (head,guard,body,k)::rest -> try (
      let sigma = matching (head,goal)
      in (* First we check the guard *)
         (* Caution: we assume that all guard variables occur in the head *)
         (do_list (check o (substitute sigma)) guard;
          (* now we rename the remaining variables of the matching gclause *)
          (* prim_reci is primitive recursion on int *)
          let (sigma',n') = prim_reci complete_sigma (sigma,!INDEX) (int_of_num k)
                 where complete_sigma ((sigma1,n1) as pair) p =
                       if mem_assoc p sigma1 then pair
```

```
                    else let n2 = n1+1 in ((p,Vari n2)::sigma1,n2)
        in (INDEX:=n';SUBGOALS(share_map (subst sigma') body))))
                            (* failure may come from match or check *)
                            with FAIL -> exec rest;;
```

3.2.4 The GHC Interpreter loop

```
type result =
   DEADLOCK of term list
 | SOLUTION of term
 | FAILURE  of term&term;;

let wake sub (awake,asleep) goal = try (((sub goal)::awake),asleep)
                                   with Identity -> (awake,goal::asleep);;
(* We wake up a suspended goal whenever the substitution changes it        *)
(* We could try and have execute return information about needed variables *)

(* The GHC loop *)
(* Same notations as for DHC *)
let rec loop answer = loop_goals
  where rec loop_goals (current,susp) = match current with
    [] -> (* we have no more current goals *)
          (match susp with
               [] -> (* no suspended gclause waiting *) SOLUTION(answer)
             | _  -> (* deadlock situation *) DEADLOCK(susp))
  | g::rest -> match g with
        Term("=",[t1;t2]) -> (let pair = (t1,t2) in match unif pair with
                  NONE -> FAILURE(pair)
                | SUBST(sigma) -> (let sub = subst sigma in
              let goals' = it_list (wake sub) (share_map sub rest,[]) susp
              in loop (share sub answer) goals'))
      | Term(":=",[t1;t2]) -> (match eval t2 with
            NUM(num) -> (let t'2=Term(string_of_num num,[]) in
                         let pair = (t1,t'2) in match unif pair with
                  NONE -> FAILURE(pair)
                | SUBST(sigma) -> (let sub = subst sigma in
              let goals' = it_list (wake sub) (share_map sub rest,[]) susp
              in loop (share sub answer) goals'))
          | NONUM -> loop_goals (rest,g::susp)
                     (* t2 is not evaluable yet, g is suspended *))
        (* Other case : non equality subgoal *)
      | Term(pred,_) -> loop_goals (match execute g (program pred) with
                SUBGOALS(body) -> (body@rest,susp)
              | SUSPEND        -> (rest,g::susp));;
```

3.2.5 The GHC interpreter top-level

```
let GHC ((ans,goals,index),D) = INDEX:=index;
  match loop ans (goals,[]) with
     SOLUTION(answer) -> (message "Solution found";
                          unparse_answer D answer)
   | DEADLOCK(susp)   -> (message "Deadlock detected with suspended goals:";
                          do_list (unparse D) susp;print_newline())
   | FAILURE(t,t')    -> (message "Constraint not satisfiable:";
                          unparse D t;
                          message " cannot be unified with ";
                          unparse D t');;
```

3.2.6 Programs management

```
(* Entering programs *)
(* Second argument is complete list of concrete gclauses pertaining to
   given predicate, given in reverse order *)
let enter_program name conc_gclauses =
if mem_assoc name !PROGRAM
   then failwith ("Program " ^ name ^ " already exists; use Forget")
else (check [] conc_gclauses; message ("Program " ^ name ^ " entered"))
     where rec check body = function
       []         -> PROGRAM:=(name,body)::!PROGRAM
     | (cl,_)::rest -> let (Term(pred,_),_) = cl in
                       if pred=name then check (cl::body) rest
                       else failwith ("Some clause does not pertain to " ^ name);;

(* To remove a program *)
let Forget name = PROGRAM:=except_assoc name !PROGRAM;();;

(* To remove all programs *)
let Reset () = PROGRAM:=[];;
```

3.3 The GHC parser

```
/* true =:= := :- */
%token TRUE EQUAL ASSIGN ARROW
%token IDENT NUM
%right ASSIGN
%right EQUAL
%right '='
%right '>'
%right '<'
%right '#'
%right '+'
%right '-'
```

```
%right '*'
%right '/'
%%
command : IDENT '=' program {enter_program $1 $3}
| goal {GHC $1}
;
program : clause '.' {[$1]}
| program clause '.' {$2::$1}
;
clause  : cl   {mk_clause($1)}
;
goal : '?' tail  {mk_goal($2)}
| ARROW tail '.' {mk_goal($2)}
;
cl : term ARROW guard '|' tail {$1,$3,$5}
;
tail : term {mk_terml(([],[]),$1)}
| tail ',' term  {mk_terml($1,$3)}
| TRUE {([],[])}
;
term : IDENT {const_or_var($1)}
        | '_'                           {dontcare()}
| NUM {mk_num($1)}
| IDENT terms {mk_term($1,$2)}
| term '=' ASSIGN term {mk_infix("=:=",$1,$4)}
| term ASSIGN term {mk_infix(":=",$1,$3)}
| term '=' term {mk_infix("=",$1,$3)}
| term '#' term {mk_infix("#",$1,$3)}
| term '>' term {mk_infix(">",$1,$3)}
| term '<' term {mk_infix("<",$1,$3)}
| term '+' term {mk_infix("+",$1,$3)}
| term '-' term {mk_infix("-",$1,$3)}
| term '*' term {mk_infix("*",$1,$3)}
| term '/' term {mk_infix("/",$1,$3)}
| '(' term ')' {$2}
        | '[' list ']'                  {mk_list $2}
;
guard : TRUE {([],[])}
| term {mk_terml(([],[]),$1)}
| guard ',' term {mk_terml($1,$3)}
;
list : {([],[]),empty_list}
| nelist '|' term  {$1,$3}
| nelist '|' '_'  {$1,dontcare()}
| nelist   {$1,empty_list}
;
```

```
nelist : term {mk_terml(([],[]),$1)}
| nelist ',' term  {mk_terml($1,$3)}
;
terms : '(' t_list term ')' {mk_terml($2,$3)}
;
t_list : {[],[]}
| t_list term ',' {mk_terml($1,$2)}
;
%%
```

3.4 Examples

3.4.1 Trivial examples

The list of N first integers is easily defined with:

```
<<ints= ints(N,L) :- N>0   | L=[N|L1], N1:=(N-1), ints(N1,L1).
        ints(N,L) :- N=:=0 | L=[]. >>;;

<<?ints(10,L)>>;;
Solution found
L = [10,9,8,7,6,5,4,3,2,1]
```

3.4.2 Hamming problem: the eager version

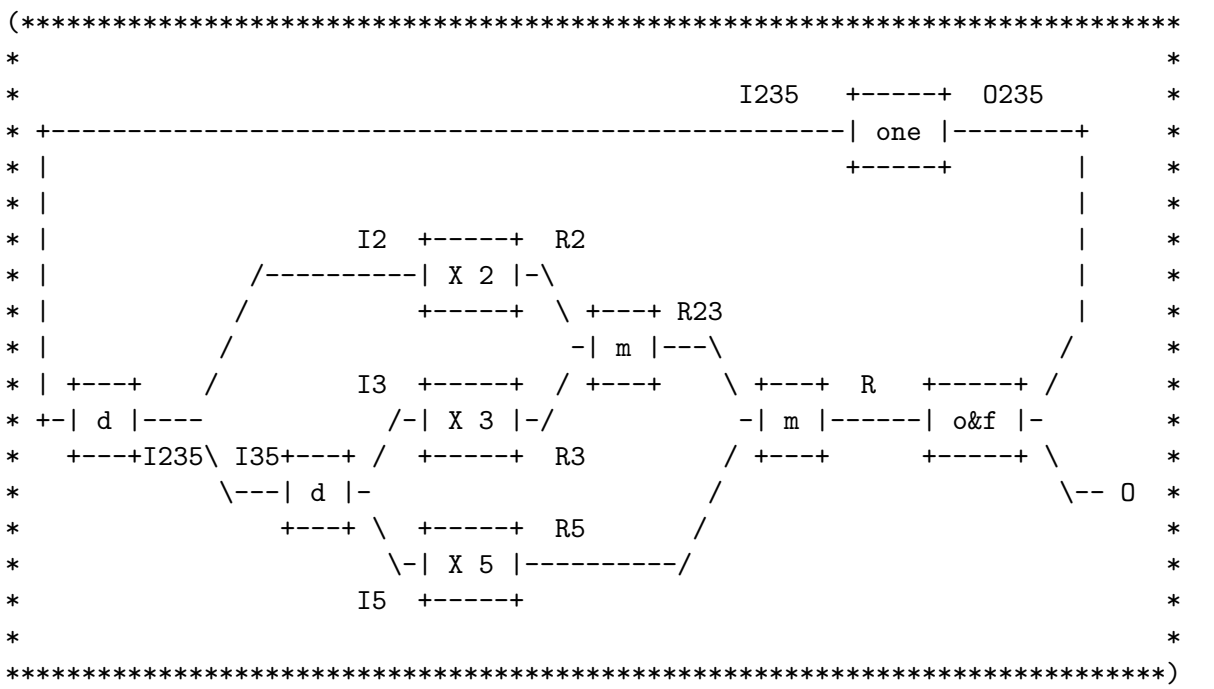
We are looking for the ascending sequence R of integers of the form $2^I * 3^J * 5^K$. The first first is an eager solver. The following GHC program is adapted from K. Ueda.

```
<<mults = mults(N,R) :- true |
              timeslist(2,N,[1|R],R2),
              timeslist(3,N,[1|R],R3),
              timeslist(5,N,[1|R],R5),
              merge(R2,R3,R23),
              merge(R23,R5,R).>>;;

<<timeslist =
timeslist(U,N,[V|X1],Y) :- U*V < N | W:=U*V, Y=[W|Y1], timeslist(U,N,X1,Y1).
timeslist(U,N,[V|X1],Y) :- U*V=:=N | W:=U*V, Y=[W|Y1], timeslist(U,N,X1,Y1).
timeslist(U,N,[V|_], Y) :- U*V > N | Y=[].>>;;

<<merge =
merge([U|X1],[V|Y1],Z) :- U < V | Z=[U|Z1], merge(X1,    [V|Y1],Z1).
merge([U|X1],[V|Y1],Z) :- U > V | Z=[V|Z1], merge([U|X1],Y1,    Z1).
merge([U|X1],[V|Y1],Z) :- U=:=V | Z=[V|Z1], merge(X1,    Y1,    Z1).
merge([],    Y,     Z) :- true  | Z=Y.
merge(X,     [],    Z) :- true  | Z=X.>>;;

(* Generate the sequence of Hamming numbers <= N *)
```

```
<<test =
test(N,R) :- true | mults(N,R).>>;;

<<?test(25,R)>>;;
Solution found
R = [2,3,4,5,6,8,9,10,12,15,16,18,20,24,25]
```

3.4.3 Hamming problem: lazy version

Again, we adapt a GHC program by K. Ueda. Note the similarity of such examples with the CSP-like examples of Kahn and MacQueen[1].




```
<<mults=
mults(O) :- true |
            timeslist(2,I2,R2),
            timeslist(3,I3,R3),
            timeslist(5,I5,R5),
            merge(R2,R3,R5,R),
            out_and_feedback(R,O,O235),
            one(O235,I235), distr(I235,I2,I3,I5).>>;;

<<out_and_feedback =
out_and_feedback(I,O,F) :- true | I=O, of1(I,F).>>;;

<<of1 =
of1(I,[A|F1]) :- true | I=[A|I1], of1(I1,F1).
of1(_,[]    ) :- true | true.>>;;
```

```
<<timeslist =
timeslist(M,I,[B|O1]) :- true | I=[A|I1], B:=M*A, timeslist(M,I1,O1).
timeslist(_,I,[]    ) :- true | I=[].>>;;

<<one =
one(I,[A|O1]) :- true | A=1, I=O1.
one(I,[]    ) :- true | I=[].>>;;

<<merge =
merge(R2,R3,R5,R) :- true | merge'(R2,R3,R23), merge'(R23,R5,R).>>;;

<<merge' =
merge'(Ix,Iy,[A|O1]) :- true | Ix=[Ax|Ix1], Iy=[Ay|Iy1],
                               merge3(Ax,Ay,A,Ix1,Iy1,O1).
merge'(Ix,Iy,[]    ) :- true | Ix=[], Iy=[].>>;;

<<merge2 =
merge2(Ax,Ix1,Iy,[A|O1]) :- true | Iy=[Ay|Iy1], merge3(Ax,Ay,A,Ix1,Iy1,O1).
merge2(_ ,Ix1,Iy,[]    ) :- true | Ix1=[], Iy=[].>>;;

<<merge3 =
merge3(Ax,Ay,A,Ix1,Iy1,O1) :- Ax < Ay | A=Ax, merge2(Ay,Iy1,Ix1,O1).
merge3(Ax,Ay,A,Ix1,Iy1,O1) :- Ax > Ay | A=Ay, merge2(Ax,Ix1,Iy1,O1).
merge3(Ax,Ay,A,Ix1,Iy1,O1) :- Ax=:=Ay | A=Ax, merge'(Ix1,Iy1,O1).>>;;

<<distr =
distr(I235,I2,I3,I5):- true | distr'(I235,I2,I35), distr'(I35,I3,I5).>>;;

<<distr' =
distr'(I,Ox,Oy) :- true | dist(I,I,Ox,Oy,0).>>;;

<<dist =
dist(Ix,Iy,[A|Ox1],Oy,D) :- true | Ix=[A|Ix1], D1:=D+1, dist(Ix1,Iy,Ox1,Oy,D1).
dist(Ix,Iy,Ox,[A|Oy1],D) :- true | Iy=[A|Iy1], D1:=D-1, dist(Ix,Iy1,Ox,Oy1,D1).
dist(Ix,_,[],[],D) :- D>0 | Ix=[].
dist(Ix,_,[],[],D) :- D=:=0 | Ix=[].
dist(_,Iy,[],[],D) :- D<0 | Iy=[].>>;;

<<list =
list(N,L) :- N>0 | L=[_|L1], N1:=N-1, list(N1,L1).
list(0,L) :- true | L=[].>>;;

(* Generate first N numbers of Hamming sequence *)
<<test =
test(N,R) :- true | list(N,R), mults(R).>>;;
```

```
<<?test(15,R)>>;;
Solution found
R = [2,3,4,5,6,8,9,10,12,15,16,18,20,24,25]
```

This program is much slower than the eager version: it takes 4.23s on a SUN 3/280, compared to 1.31s for the eager version.

3.4.4 A hardware simulator

This logic simulator is due to Y. Noda, modified by K. Ueda.

```
(* Structure of a 3-bit adder *)
<<add_3 =
add_3(Inx0,Inx1,Inx2,Iny0,Iny1,Iny2,Sum0,Sum1,Sum2,Carry,Ground) :- true |
    fadd(Inx0,Iny0,Ground,Carry0,Sum0),
    fadd(Inx1,Iny1,Carry0,Carry1,Sum1),
    fadd(Inx2,Iny2,Carry1,Carry, Sum2).
>>;;
<<fadd =
fadd(Inx,Iny,Cin,Carry,Sum) :- true |
    inv(Inx,M1),
    inv(M2, M3),
    mpx(Inx,M1, Cin,M2),
    mpx(Inx,Iny,M2, Carry),
    mpx(M2, M3, Iny,Sum).
>>;;
<<mpx =
mpx(Inx,Iny,Ctr,Out) :- true |
    nand_2in(M2, M3,M4),
    nand_2in(Ctr,M1,M5),
    nand_2in(M4, M5,M6),
    inv(Ctr,M2),
    inv(Inx,M3),
    inv(Iny,M1),
    inv(M6,Out).
>>;;

(* Behavior of gates *)
<<nand_2in =
nand_2in(Is0,Is1,Os) :- true | gate_2in(nand,Is0,Is1,Os).
>>;;

<<delay_time =
delay_time(nand,D) :- true | D:=1.
delay_time(inv,D)  :- true | D:=1.
>>;;
```

```
<<evaluator2 =
evaluator2(nand,0,_,X) :- true | X=1.
evaluator2(nand,_,0,X) :- true | X=1.
evaluator2(nand,1,1,X) :- true | X=0.
evaluator2(nand,_,_,X) :- true | X=x.
>>;;

<<inv =
inv(Is,Os) :- true | gate_1in(inv,Is,Os).
>>;;

<<evaluator1 =
evaluator1(inv,1,X) :- true | X=0.
evaluator1(inv,0,X) :- true | X=1.
evaluator1(inv,x,X) :- true | X=x.
>>;;

(* Execution control *)
<<gate_2in =
gate_2in(Type,I0s,I1s,Os) :- true |
    delay_time(Type,D), ini_out(D,Os,Os1),
    gate_2(Type,I0s,I1s,Os1).
>>;;
<<gate_2 =
gate_2(Type,[I0|I0s],[I1|I1s],Os) :- true |
    evaluator2(Type,I0,I1,O), Os=[O|Os1],
    gate_2(Type,I0s,I1s,Os1).
gate_2(_,   [],      _,       Os) :- true | Os=[].
gate_2(_,   _,       [],      Os) :- true | Os=[].
>>;;

<<gate_1in =
gate_1in(Type,I1s,Os) :- true |
    delay_time(Type,D), ini_out(D,Os,Os1),
    gate_1(Type,I1s,Os1).
>>;;
<<gate_1 =
gate_1(Type,[I1|I1s],Os) :- true |
    evaluator1(Type,I1,O), Os=[O|Os1],
    gate_1(Type,I1s,Os1).
gate_1(_,   [],      Os) :- true | Os=[].
>>;;

<<ini_out =
ini_out(D,Os,Os2) :- D>0  | D1:=D-1, Os=[x|Os1], ini_out(D1,Os1,Os2).
```

```
ini_out(0,Os,Os2) :- true | Os=Os2.
>>;;

(* The top-level *)
<<add3 =
add3(End,P1,P2,P3,P4) :- true |
    timer(End,Ts),
    ground(End,Ground),
    data_in6(Inx0,Inx1,Inx2,Iny0,Iny1,Iny2,Ts),
    add_3(Inx0,Inx1,Inx2,Iny0,Iny1,Iny2,Sum0,Sum1,Sum2,Carry,Ground),
    probe(Sum0,Ts,P1),
    probe(Sum1,Ts,P2),
    probe(Sum2,Ts,P3),
    probe(Carry,Ts,P4).
>>;;


(* Definition of input signals to the circuit *)
<<input_data =
input_data(inx0,X) :- true | X=[p(0,0),p(20,1)].
input_data(inx1,X) :- true | X=[p(0,0),p(20,1)].
input_data(inx2,X) :- true | X=[p(0,0),p(20,1)].
input_data(iny0,X) :- true | X=[p(0,0),p(20,1)].
input_data(iny1,X) :- true | X=[p(0,0),p(20,1)].
input_data(iny2,X) :- true | X=[p(0,0),p(20,1)].
>>;;


(* Clock generator *)
<<timer =
timer(End,Ts) :- true | timer2(End,0,Ts).
>>;;
<<timer2 =
timer2(End,Now,Ts) :- End>Now   |
    Ts=[Now|Ts1], Next:=Now+1, timer2(End,Next,Ts1).
timer2(End,Now,Ts) :- End=:=Now | Ts=[].
>>;;

(* Supplier of input signals *)
<<data_in6 =
data_in6(Inx0,Inx1,Inx2,Iny0,Iny1,Iny2,Ts) :- true |
    input_fetch(inx0,Inx0,Ts),
    input_fetch(inx1,Inx1,Ts),
    input_fetch(inx2,Inx2,Ts),
    input_fetch(iny0,Iny0,Ts),
    input_fetch(iny1,Iny1,Ts),
```

```
    input_fetch(iny2,Iny2,Ts).
>>;;
<<input_fetch =
input_fetch(Name,In,Ts) :- true |
    input_data(Name,Ss),
    fetch1(In,Ss,Ts).
>>;;
<<fetch1 =
fetch1(Is,Ss,Ts)   :- true | fetch2(Is,Ss,Ts,x).
>>;;

<<fetch2 =
fetch2(Is,[p(T1,X1)|Ss],Ts,X) :- true | fetch3(Is,Ss,Ts,X,T1,  X1).
fetch2(Is,[],           Ts,X) :- true | fetch3(Is,Ss,Ts,X,9999,_ ).
>>;;

<<fetch3 =
fetch3(Is,Ss,[T|Ts],X,T1,X1) :- T < T1 | Is=[X|Is1], fetch3(Is1,Ss,Ts,X,T1,X1).
fetch3(Is,Ss,[T|Ts],_,T1,X1) :- T=:=T1 | fetch2(Is,Ss,[T|Ts],X1).
fetch3(Is,_, [],    _,_, _ ) :- true   | Is=[].
>>;;


(* Ground signal generator *)
<<ground =
ground(End,Gs) :- End > 0 | Gs=[0|Gs1], End1:=End-1, ground(End1,Gs1).
ground(End,Gs) :- End=:=0 | Gs=[].
>>;;

(* Observer of signal streams *)
(* <<?probe([x,x,x,0,0,1,1,x,0,0,0,0],[1,2,3,4,5,6,7,8,9,10,11,12],S)>>;;
computes S = [p(4,0),p(6,1),p(8,x),p(9,0)]  *)

<<probe =
probe(Os,Ts,S) :- true | probe1(Os,Ts,S,x).
>>;;

<<probe1 =
probe1([0|Os],[_|Ts],S,0) :- true | probe1(Os,Ts,S,0).
probe1([0|Os],[T|Ts],S,_) :- true | S=[p(T,0)|S1], probe1(Os,Ts,S1,0).
probe1(_,     [],    S,_) :- true | S=[].
>>;;

<<?add3(35,Sum0,Sum1,Sum2,Carry)>>;;
Solution found
Sum0 = [p(8,0),p(24,1),p(29,0)]
```

```
Sum1 = [p(16,0),p(24,1),p(29,0),p(32,1)]

Sum2 = [p(25,1),p(29,0),p(32,1)]

Carry = [p(24,1)]
```

Notes

The implemented syntax is poorer than ICOT's GHC. For instance, <= and >= are not provided. Predicates are assumed to have a fixed arity. This involves some renaming if variable arity is used. If you give two different arities to a predicate, this may provoque the failure: `Evaluation Failed: combine`. Finally, pairs must be explicitated with an operator.

4 Conclusion

These experiments provide us with two interpreters: one for a simplified GHC without guards, called DHC, the other for a fuller version of GHC admitting guards with arithmetic predicates. These CAML programs are very concise and express clean mathematical definitions, since they are completely applicative, except for the use of exceptions. They can thus serve as reference definitions of operational semantics of the corresponding subsets of GHC.

On the practical side, they are efficient enough to treat non-trivial benchmarks. They are very easily modifiable, for instance to trace the computations, and gather statistics about GHC's behaviour. They may thus serve as a basis for experiments with debugging facilities, and other program analysis tasks. It took the author less than 2 weeks to write and debug these programs, which shows that CAML is a powerful environment for fast prototyping in software engineering and artificial intelligence.

This very short study of GHC suggests many research problems. For instance, it seems that we should be able to profit of the execution of GHC with pattern-matching instead of unification in order to use some work in the domain of term rewriting systems. It seems that numerous GHC programs are deterministic in a strong sense. When the heads of the various clauses pertaining to a predicate are non-overlapping, in the sense of being non mutually unifiable, we know that only one clause may apply to a given goal. When the heads are linear, i.e. without repeated occurrences of variables, we may think to try and check the condition of strong sequentiality defined by Huet and Lévy. The technique of Laville may be useful for transforming a GHC program with overlapping clauses to one which is free of overlaps, permitting the previous test. Strongly sequential GHC programs may be compiled very efficiently, since the relevant clause may be accessed directly using a dictionary structure or "pattern-matching dag".

Conversely, we may define for GHC programs, an appropriate notion of "sufficient completeness". This would involve a minimum of typing, since we would check the completeness of arguments according to all constructors of a type, but this typing could be inferred automatically from simple declarations of data types constructors. For instance, the list data type could be declared similarly to ML, as:

```
type list = [] | [x:'a|y:list]
```

Now, if P is a unary predicate, the heads $P([])$, $P([X])$ and $P([X, Y|Z])$ form a sufficiently complete set over lists. We may try and use such notions to give sufficient conditions for deadlock avoidance of certain goals.

GHC programs are better adapted to potential parallel execution than PROLOG programs, since no backtracking data-structure is needed. However, multiples occurrences of some variables in distinct goals present a potential problem. It seems that many GHC programs are descriptions of data-flow processes, where there is a clear distinction between input and output. It seems that annotations for variables (similar to the explicit annotations of Concurrent Prolog) could be automatically synthesized in many situations. These indications could then be used by the GHC system in order to have a finer analysis of how to wake-up suspended goals. That is, we would wake-up only the goals which were blocked when waiting on some input variable, which has become instanciated in the current cycle. It seems that these notions should naturally suggest a "lazy GHC".

The treatment of equality goals is similar to resolution with a clause $x = x$. This is the only case where we use unification instead of matching. Maybe we could generalize this to allow two kinds of programs: some would admit side-effects through unification like in standard PROLOG (and then = would be such a program), and some would use the matching/suspension paradigm. We would have thus a hybrid of PROLOG and GHC.

These few remarks are very hasty, and come from a naive GHC beginner, and many of the suggested problems have probably been looked at intensively by the GHC specialists. Thus partial or complete solutions or refutals may be already well-known. We hope in any case that this discussion might provide some ground for future collaborative work on these topics.

References

[1] G. Kahn, D. Mac Queen. "Coroutines and networks of parallel processes." Rapport Laboria 202, IRIA (Nov. 1976).

[2] K. Ueda. *Guarded Horn Clauses*, Doctoral thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo (1986).

[3] K. Ueda. *Guarded Horn Clauses.* In *Concurrent Prolog: Collected Papers*, Vol. 1, Ed. E. Shapiro, MIT Press, 1987.

[4] K. Ueda. *Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard.* Programming of Future Generation Computers, Eds. M. Nivat and K. Fuchi, North-Holland, to appear.

[5] K. Ueda. *Introduction to Guarded Horn Clauses.* ICOT Tech. Report TR-209, Institute for New Generation Computer Technology, Tokyo, 1986.

[6] K. Ueda. *On the Operational Semantics of Guarded Horn Clauses.* ICOT Tech. Memorandum TM-160, Institute for New Generation Computer Technology, Tokyo, 1986.

[7] K. Ueda. *GHC Compiler User's Guide.* DEC-10 Prolog Version 1.10, Dec. 24th, 1987.