Logical Programming and Error Recovery

Thierry Despeyroux

INRIA – Sophia Antipolis
B.P. 93 - 06902 Sophia Antipolis Cedex - France
thierry.despeyroux@sophia.inria.fr

Prolog has a great potential as a high level programming language or as a specification language. However, the Prolog programmer would be happier if it was easier to isolate the pure logical parts of a program from its low level or non logical ones. This is the case in particular if one wants to derive a real tool from a logic program by adding some error recovery. This paper presents a clean way of adding error recovery to a pure prototype. This method has been used for generating Prolog code from a high level specification language that has a built-in error recovery mechanism.

1 Introduction

The Typol specification language [Des83, Des88] is a specialized formalism designed to ease the specification of programming languages. It is integrated in the Centaur system[I.N94, JLG92], a system designed to generate interactive programming environments. From a Typol specification (for the static semantics or the dynamic semantics of an object language) one either generates executable code (type-checkers, interpreters, compilers), or use the specification to build proofs about the language itself or about some properties of some programs. Typol is a "pure" logical system in which the semantics of programming languages are given by means of axioms and inference rules also known as Natural Semantics[Kah87, CDD$^+$85]. In this settle, the execution of a type checker is viewed as building the proof that a program type-check, executing an object program is no more than proving that this program gives a value. Depending on the semantics itself, on the expected performances, on the (logical) goal that one wants to achieve, Typol specifications can be compiled into Lisp [ACG92], attributed grammars [AFZ88], data for a general proof system [Ter95], or compiled into Prolog. The Prolog implementation of Typol may be seen as its natural or standard implementation, even though the deapth-first search strategy of Prolog, that is not part of the semantics of Typol, is inherited.

When giving a semantic specification (let's say when specifying a type-checker, for example), one wants to focus on the main points of interest: the correct cases. The semantic rules describe the logic that will be used to prove that a statement is correct. Everything that cannot be proved with the system of rules is defined as false.

If this is acceptable from a theoretical point of view, it is not reasonnable from a pratical point of view. Who would be satisfied with a type-checker or a compiler which only gives *true* or *false* as output? The minimal information expected by the end-user in case of error is *why did it fail*, and *where*. Furthermore, one would like the process to go on if possible, not stopping at the first error encountered, emitting more error messages if necessary. So two different features must be provided by an error recovery mechanism: localizing errors, and allowing the whole process to continue.

From the specifier point of view, an error mechanism must not pollute nor interfer with the rest of the specification for at least two reasons: the regular semantics rules must stay as pure as possible to be easily understandable as this specification may be used by other tools as provers to check or prove meta-properties. Futhermore, the management of error messages may involve some low level, "dirty", programming.

The solution presented here is implemented in the current Prolog implementation of Typol. Typol programs contain axioms and inference rules that are roughly compiled into Prolog on a rule by rule basis. However some extra parameters are generated mecanically. Futhermore, the Typol objects are typed terms, known also as abstract syntax trees. As Prolog term are untyped some dynaming typing predicates are generated, using Prolog delayed procedures.

To make our presentation clearer, we will present all examples directly in Prolog, taking away all pieces of code generated by Typol that is not used in our examples. Of course, the method can be applied more generally to every logical program, and not only to semantic specifications.

2 A Pure System

Let's focus on a particular example. We want to check a piece of an imperative programming language, namely assignments. Our language will contain constants and typed variables. An assignment is correct iff all variables are declared and if both sides of the assignment have compatible types. Our programs are represented as Prolog terms. The type of a variable must be found in a structure called environment, or symbol table, that is a list of pairs of a variable name and a type. This environment is supposed to be constructed somewhere else (typically when analysing a declarative part of the program). We will allow overloading (i.e. a variable name may be assigned different types). The following (pure) Prolog program implements our type-checker.

```
type_check(Env,assign(V,E),T):- % t1
   type_check(Env,V,T),
   type_check(Env,E,T).
type_check(Env,int(X),int).     % t2
type_check(Env,true,bool).      % t3
type_check(Env,false,bool).     % t4
type_check(Env,var(X),T):-      % t5
   look_up(Env,X,T).
type_check(Env,plus(A,B),T):-   % t6
   type_check(Env,A,T),
   type_check(Env,B,T).

look_up([pair(X,T)|Q],X,T).     % l1
look_up([pair(X1,T1)|Q],X,T):-  % l2
   look_up(Q,X,T).
```

The syntax of our toy language is small enough to be easily understandable. Type checking is defined by induction on the structure of the program. Searching in the environment is done by the predicate `look_up`. Notice that rule `l2` does not contain any negation, allowing overloading: when a variable appears twice or more in the environment, all types will be found by backtracking.

The behaviour of this type-checker is boolean. If the program given as data is correct versus a given environment it succeeds, else it fails.

The execution of a Prolog program may be seen as a proof construction. We suppose in the following that all prolog clauses are named.

Following [Des92] a proof may be recursively defined by:

- G, a goal

- $(A:\ T)$, an axiom, where T is a literal and A its name

- $(R\ P_1 ... P_n)$ where R is a clause name and $P_1 ... P_n$ are proofs, denoting the application of clause R to the list $P_1 ... P_n$.

A proof is complete when it does not contain any goal.

Starting with an initial goal, executing a semantics is building a proof, replacing successively a goal by the application of a rule to subgoals or by an axiom, whenever it is possible. This is known as the resolution principle [Rob65]. Notice that the notion of proof is an abstract notion and does not take the strategy of execution into account.

Notice also that, if we may have proofs of true facts, we cannot build a proof of a negative fact, for exemple that some program is not correctly typed.

For example, with the environment `{x:bool,x:int}` the program `x=5` is proved correct, and here is the proof:

```
(t1(t5(l2(l1:
      look_up([pair(x,int)],x,int))))
   (t2:type_check
         ([pair(x,bool),pair(x,int)],
          int(5),int)))
```

Notice that the type checker of a real language with overloading, like ADA, will check than the statements are not ambiguous, i.e. that there is unicity of the typing proof.

3 Adding Error Recovery

Of course, a boolean type-checker is of no use in real life. This means that we must handle erroneous cases. When writing a type-checker in a functional language, one will use if-then-else constructs. The same behaviour may be achieved in Prolog. The problem is that the execution of a logical program may be non-deterministic, and the initial specification is polluted by new clauses, with perhaps some cuts and some negations. If we want to avoid these negations it may be difficult and bothering to express them explicitly.

If a negation is needed in a specification, the proof of the negation must appear in the proof and non logical predicates as negation as failure must be prohibited. But this requirement is too strong in the case of error recovery.

3.1 Proof with recovery

A good idea seems again to think in terms of proofs. Type-checking an erroneous program results in failing to construct the proof that the program type-checks. To avoid this boolean behavior we can try to build an incomplete proof of the correctness, i.e. a proof with some remaining goals in it. This remaining goals will be considered as error occurrences in the program being type-checked. Doing this is not sufficient as the missing pieces of proof may imply some missing bindings.

The solution now is to replace remaining goals by alternative goals that will mark the proof as incomplete (or erroneous) and bind the variables that are needed to complete the rest of the proof, explaining how to recover from the error.

We can now give a new definition of a proof with or without error recovery. It is recursively defined by:

- G, a goal

- $(A:\ T)$, an axiom, where T is a literal and A its name

- $(R\ P_1 ... P_n)$ where R is a clause name and $P_1 ... P_n$ are proofs

- $[A:\ T]$, where T is a recovery literal and A its name

- $[R\ P_1 ... P_n]$ where R is a recovery clause name and $P_1 ... P_n$ are proofs

The proofs that do not contain any error recovery correspond to the previous definition. As before, a proof is complete when it does not contain any goal.

In the case of a type-checker, a correct program will give a complete proof without recovery, a program containing error will give a complete proof with recovery. In any case, the execution of the type-checker must give a complete proof. If it is not the case, it means either that the type-checker itself is not correct, or that some error recovery rules are missing.

Back to our example, an obvious way to add error recovery to our type checker is to assign an arbitary type to every untyped expression. We would like to be able to say:

```
type_check(Env,assign(V,E),T):-  % t1
   type_check(Env,V,T),
   type_check(Env,E,T).
type_check(Env,int(X),int).      % t2
type_check(Env,true,bool).       % t3
type_check(Env,false,bool).      % t4
type_check(Env,var(X),T):-       % t5
   look_up(Env,X,T).
type_check(Env,plus(A,B),T):-    % t6
   type_check(Env,A,T),
   type_check(Env,B,T).
recovery
  type_check(Env,E,T).           % rt1
     % message: Type error

look_up([pair(X,T)|Q],X,T).      % l1
look_up([pair(X1,T1)|Q],X,T):-   % l2
   look_up(Q,X,T).
```

The keyword "recovery" isolates the regular clauses of a predicate from its error recovery part.

3.2 Accuracy of a proof with recovery

Without more rules to precise when a recovery rule may be applied, proofs with recovery (as well as incomplete proofs) can be build even for correct programs. For example, with the environment `{x:bool,x:int}`, `x=5` may yields to the following proof with error recovery :

```
(rt1:type_check
       ([pair(x,bool),pair(x,int)],
        assign(var(x),int(5)),T))
```

where `T` is a free variable.

The intuitive missing strategic rule is that a recovery rule can be used only when nothing else is possible. The meaning of "nothing else is possible" must be however investigated with care. One want error recovery to be as accurate as possible. For example, in the environment `{x:int}` when type-checking the program `x=y+5` we would like an error message to be attached to `y` (which is not in the environment) and not only on `y+5` (it would be correct, but not accurate enought to our taste). In other words, we want the error recovery rules to be applied as deeper in the proof tree as possible. We can define a notion of accuracy, saying that

```
(t1(t5(l1:look_up
            ([pair(x,int)],x,int)))
   (t6(rt1:type_check
             ([pair(x,int)],y,int))
      (t2:type_check
            ([pair(x,int)],
             int(5),int))))
```

is more accurate than

```
(t1(t5(l1:look_up
            ([pair(x,int)],x,int)))
   (rt1:type_check
          ([pair(x,int)],
           plus(var(y),int(5)),
           int))).
```

Accuracy gives a partial order on proofs with recovery. Given a set of proofs our tactic of execution must reject any proof that is less accurate than another one.

As we have only a partial order, two non comparable proofs with recovery may be of interest. For example, in the environment `{x:int,x:bool,y:bool}` the assignment `x=y+5` yields to two possible error recoveries and thus to two uncomparable proofs :

-when `x` is an integer there is a type error on `y`

```
(t1(t5(l1:look_up([pair(x,int),
                   pair(x,bool),
                   pair(y,bool)],
                  x,int)))
    (t6(rt1:type_check
             ([pair(x,int),
               pair(x,bool),
               pair(y,bool)],
              y,int)))
       (t2:type_check([pair(x,int),
                       pair(x,bool),
                       pair(y,bool)],
                      int(5)),int))
```

-and when x is a boolean the error is on 5

```
(t1(t5(l2(l1:look_up([pair(x,bool),
                      pair(y,bool)],
                     x,bool)))
    (t6(t5(l2(l1:look_up
                 ([pair(y,bool)],
                  y,bool))))
       (rt1:type_check
             ([pair(x,int),
               pair(x,bool),
               pair(y,bool)],
              int(5)),bool))
```

The implementation given below will give all the possibilities on backtracking.

3.3 Incremental development

Having a notion of error recovery in a programming language rather than having to code it at a lower level discharges the user of a lot of control that might, in the case of logical programming, affect the semantics of the program.

Starting with a "pure" program, with a boolean behaviour, we have added only one error recovery rule. So now our program is able to report some type errors. It is now possible to refine this system just by adding more recovery rules. The following program will report more accurate messages:

```
type_check(Env,assign(V,E),T):- % t1
   type_check(Env,V,T),
   type_check(Env,E,T).
type_check(Env,int(X),int).     % t2
type_check(Env,true,bool).      % t3
type_check(Env,false,bool).     % t4
type_check(Env,var(X),T):-      % t5
   look_up(Env,X,T).
type_check(Env,plus(A,B),T):-   % t6
   type_check(Env,A,T),
   type_check(Env,B,T).
recovery
  type_check(Env,plus(A,B),T):- % rt2
     type_check(Env,A,T1),
     type_checl(Env,B,T2).
     % message: T1 and T2 are not
     %   compatible for +
  type_check(Env,E,T).          % rt1
     % message: Type error

look_up([pair(X,T)|Q],X,T).     % l1
look_up([pair(X1,T1)|Q],X,T):-  % l2
   look_up(Q,X,T).
recovery
  look_up(Env,X,T).             % rl1
     % message: The variable X should
     %  be declared with type T.
```

In case of a non declared variable, a specific message may be reported (rule rl1). And in case of a "plus" construct the type mismatch is reported, but in this case subexpressions may be typechecked independantly (rule rt2).

Notice that in the pure part of our program the order of the clauses is not significant (this is the case in Typol that does not contain non-logical features), but the order is important in the recovery part. Error recovery rules are tried in order, and only the first one that lead to a correct subproof may be used. This is why the rule rt1 is placed after rule rt2 in our example.

4 Implementation

This leads us to the following strategy of execution for proving a subgoal. We first present our strategy at the meta-level. Even if there is a large variety of legal proofs with recovery, we want a strategy that searches for the most accurate ones.

To prove a subgoal, the following attempts must be done in order :

1. Try to prove it without using any recovery in the subproof
2. Try to prove it allowing recovery in the subproof
3. Try recovery clauses in order for the current subgoal
4. Fail

To implement this strategy, we have to keep in mind some particular constraints.

1. When a particular solution for a subproof is found, we know that error recovery must not be used at this point, but backtracking must still be possible. This means that a regular cut cannot be used; a soft cut [Smo84, Nai89, Nai95] is needed.
2. If coroutining is used (this is the case in the Prolog implementation of Typol), some care must be taken when some delayed goals appear inside a negation or before a soft cut.
3. The implementation of error recovery must not slow down the regular execution. However, when errors occur, we may think that the time performance problem is of lower interest.

These constraints lead us to the following implementation that is used by the code generator of Typol, and may be used in other contexts. The first implementation was made using MU-Prolog [Nai85, TZ88, Nai86], while the current one is done using ECLiPSe [E.C94].

In some places, there is the possibility of using either a negation or a soft cut. Checking that delayed goals do not appear before a soft cut is slow, at least with some implementations of Prolog, so we have prefered to use a negation in some places to comply with our time performance constraint.

An argument is added to each initial predicate. This argument will be bound to the constant 'recovery' when a recovery rule is used. So it is possible to reject a subproof that uses error recovery, binding this variable to an other constant, 'norecovery', in a call. Each predicate of the form $P(A_1, ..., A_n)$ is replaced by three predicates: $P(A_1, ..., A_n, Rec)$, $P_{reg}(A_1, ..., A_n, Rec)$, and $P_{rec}(A_1, ..., A_n)$. In the following we will omit the parameters A_i. The predicate P is in fact a meta-predicate and follow the same scheme for all predicates that use error recovery :

```
P(Rec):- Preg(norecovery).
P(recovery):-
   snot(Preg(norecovery)),
   Preg(recovery).
P(recovery):- snot(Preg(_)), Prec.
```

For each regular clause of the form `P :- Q.` we generate a clause

```
Preg(Rec):- Q(Rec).
```

For each recovery clause of the form `P :- Q.` we generate a clause

```
Prec:- scall(Q(Rec)), soft_cut.
```

The predicates that do not use error recovery remains unchanged, except that a variable `Rec` is added to the normal parameters to propagate the recovery flag.

The predicates `snot` and `scall` check that there is no new delayed goals at the end of the execution of respectively a `not` and a `call`. If this is the case, an error message is emitted, explaining that the error recovery mechanism may fail because they have been a new delayed goal inside a not or before a soft cut. If this delayed goal appends to fail, the soft cut is executed too early, cutting some recovery rules that should not have been cut. A more complete implementation should wait until all delayed goals have been woken and proved before performing the soft cut, but this is hard to achieve using current Prolog implementations.

The code given above is for the MU-Prolog implementation. The ECLiPSe soft cut needs a slightly different code :

```
Prec:-
   *->(subcall(Q(Rec),D),
       (check_no_delayed(D),
        recall(D));fail.
```

As `subcall` removes the delayed goals that are produced during its execution, these goals must be reactivated after producing an error message. This is performed by `recall`.

A preliminary version of this method has been used for several years by the Typol compiler that generates Prolog code from semantic specifications. This implementation allows the user to enable or disable the error recovery mechanism manually to improve performances in some cases. However, it should be possible to optimize the method after a static analysis of the program, in particular when some predicates do not have recovery rules or when some predicates are declared deterministic.

5 Correctness

We will not give any formal proof of the correctness of the implementation. However we can state some properties of this implementation. In the following we will note P an initial logical program, $\overline{P}$ the same program with some added recovery rules, $P(A)$ and $\overline{P}(R, A)$ some goals.

Completeness If $P(A)$ succeeds with some substitution σ then $\overline{P}(norecovery, A)$ succeds with the same substitution.

Soundness If $\overline{P}(norecovery, A)$ succeeds with some substitution σ then $P(A)$ succeeds with the same substitution.

So, we can say that P and $\overline{P}(norecovery)$ are **equivalent**.

Expressiveness $\overline{P}$ is more expressive than P: if $\overline{P}(R, A)$ succeeds with $R = recovery$ then $P(A)$ fails.

Accuracy Given a goal, if two valid complete proofs with recovery P_1 and P_2 can be build and P_1 is more accurate than P_2, then P_2 will be rejected by the implementation.

6 Conclusion

In this paper we have presented a clean way of handling error recovery in logical programs. Our main concern was to preserve the semantics and the readability of the initial program.

We use two important features not available in all prolog systems and not enought well known, namely the soft cut ('*->' in ECLiPSe, $soft_cut in MU-Prolog), and the access to the list of delayed goals (available in ECLiPSe, and hacked in our MU-Prolog implementation).

The implementation of error recovery is an important feature of the Typol language. It was a crutial tool in making this specification language of interest to produce real programming tools. As the mechanism used is independant of Typol, it could be used by other systems that produce Prolog code, or be implemented in futher implementation of Prolog and could be of great interest to help some prototypes to become real tools.

References

[ACG92] A. Attali, J. Chazarin, and S. Gilette. Incremental evaluation of natural semantics specifications. In *Progrmming Language Implementation and Logic Programming*, LNCS, leuven, Belgium, august 1992.

[AFZ88] I. Attali and P. Franchi-Zannetttaci. Unification-free execution of typol programs by semantics attribute evaluation. In *Fifth International*

Conference Symposium on Logic Programming, Seattle, 1988. MIT Press.

[CDD+85] D. Clément, J. Despeyroux, Th. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Technical Report RR416, I.N.R.I.A., june 1985.

[Des83] Th. Despeyroux. *Spécifications sémantique dans le système Mentor.* PhD thesis, Paris-Sud University, october 1983. in french.

[Des88] Th. Despeyroux. Typol, a formalism to implement natural semantics. Technical Report RT94, I.N.R.I.A., march 1988.

[Des92] J. Despeyroux. Theo: an interactive proof development system. *The Scandinavian Journal on Computer Science and Numerical Analysis (BIT), special issue on 'Programming Logic'*, 32:15–29, 1992.

[E.C94] E.C.R.C. *ECLiPSe 3.4 User Manual*, july 1994.

[I.N94] I.N.R.I.A. *The Centaur 2.0 Manual*, 1994.

[JLG92] I. Jacobs and L.Rideau-Gallot. A centaur tutorial. Technical Report RT140, I.N.R.I.A., july 1992.

[Kah87] G. Kahn. Natural semantics. In *Symp. on Theoretical Aspects of Computer Science*, Passau, Germany, february 1987. also I.N.R.I.A. Research Report 601.

[Nai85] L. Naish. The mu-prolog 3.2 reference manual. Technical Report TR85/11, University of Melbourne, Department of Computer Science, october 1985.

[Nai86] L. Naish. *Negation and Control in Prolog*, volume 238 of *LNCS*. Springer-Verlag, 1986.

[Nai89] Lee Naish. Pruning the search space of Prolog: cut and other primitives. *Proceedings of Australian Joint Artificial Intelligence Conference*, pages 238–253, November 1989.

[Nai95] Lee Naish. Pruning in logic programming. Technical Report 95/16, Department of Computer Science, University of Melbourne, Melbourne, Australia, June 1995.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12-1:32–41, Janvier 1965.

[Smo84] Gert Smolka. Making control and data flow in logic programs explicit. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 311–322, Austin, Texas, August 1984.

[Ter95] D. Terrasse. Encoding natural semantics in coq. In *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95*, volume 936 of *Springer-Verlag LNCS*, pages 230–244, July 1995.

[TZ88] J. A. Thom and J. Zobel. The nu-prolog 1.3 reference manual. Technical Report TR86/10, University of Melbourne, Department of Computer Science, january 1988.