THE CONCEPT OF EXCEPTION HANDLING IN VIP PROLOG

Susi Dulli*), Eva Kühn**)

*) Facolta´ di Statistica
Universita´ di Padova
Italy-35100 Padova
on leave at **)

**) Technical University of Vienna
Argentinierstr. 8/3/9
Austria-1040 Vienna

Abstract
This paper describes a new proposal for an exception handling mechanism for a Prolog Interpreter (the VIP interpreter is a superset of DEC-10 Prolog characterized by module concepts and global variables. It runs the naive reverse on a standard MC68020 microprocessor with 50 KLIPS, on a 16 MHZ MC68000 with15 KLIPS, and on a 8 MHZ Intel80286 with 12 KLIPS).
The paper is organized as follows: a brief introduction addresses the issue of exceptions. Section 1 describes the exception handling mechanisms as they are in the existing implementations of Prolog, Section 2 discusses the central issues in exception handling schemes, while a new proposal together with its implementation for the VIP Prolog is given in Section 3. Examples are in Section 4.

INTRODUCTION

The events or conditions that a program encounters during its execution can be classified as either usual or exceptional. Examples of exceptional conditions include the following: a subprogram discovers that some values of parameters can cause the execution of an illegal division by zero; a storage allocator runs out of storage to allocate; a protocol error is caught during reception of a message on a transmission line. All these are considered anomalous conditions and the ability to deal with them is generally called exception handling. Indeed we may define it as a methodology to obtain reliability which consists of considering a priori every eventual situation of "malfunctioning" and providing tools (mechanisms, constructs etc.) to handle these situations and to recover the program. The language should make it possible to trap undesired events (arithmetic overflows, invalid input, null determinant) and to specify suitable response to such events. It is common use in conventional programming languages, to divide a program into several units in order to obtain a program that fullfilles the demand on structured programming. Generally units handle the usual events and can detect the occurence of anomalous conditions

(exceptions). The occurence of an exception transfers control to an appropriate unit, called an exception handler, that deals with the exception.
Earlier programming languages provide a primitive exception handling (mainly through labels and parameter passing), while recent ones provide ad hoc features. PL/1 was the first language to provide an explicit construct for exception handling, while later languages included them in their design as advanced facilities. Functional languages too, did not ignore this issue [Harp86], [HMQM86]. Recently also some implementations of Prolog [Prolog1], [Prolog2], [MProlog],considered exception handling, but due to the fact that Prolog identifies a different paradigm in the field of programming languages, existing exception handling mechanisms cannot easily be adapted.

Section 1. EXCEPTION HANDLING IN PROLOG

We will here briefly review the exception handling features offered by some implementations of Prolog.

Prolog-1 version 2.0 of 8086 includes error handling primitives to recover programs from some errors. The error handling mechanism consists of a new system state flag which can be set to disable the normal mechanism of printing a message and entering a break state. When this is done, a goal which generates an error fails, but causes the side effect of setting an error number variable, which can in turn be examined as a system state. The user is free to examine the error number, and either to resume execution (ignoring the error, outputting a message and continue), or to abort execution at his discretion.

Prolog-2 also has facilities for handling errors. More exactly Prolog-2:
-provides a default error handler
-allows the user to define errors and to decide how they will be handled
-enables to switch off the error handler completely, wether if it is the one supplied by the system itself or by the user. The error handler, an external module that may be altered or replaced, determines what action is available after the error, presenting a menu of options (abort, break, exit, fail, help, retry, trace).
No matter whether user or system defined, in Prolog-2 there are four kinds of errors (numbered from 0 to 3), depending on their seriousness. The user can also add new errors that the interpreter will recognize, but only within the 0-3 scale. If the error is type 0, 1, or 2 an error-window containing the menu of options appears; of course for a type 3 error (fatal error) the return to the operating system will be forced. When the interpreter detects an error, Prolog-2 does the following: it recognizes the kind of error (0-3), finds the appropriate error-number, passes this information on to the error-handler and invokes the user defined handler or, if any, the default one.

MPROLOG allows the user to handle errors by defining predicates to take over from the ongoing process. More precisely the programmer may define error-handling predicates which catch errors by their type (as usual in any other language), or by protecting a region of the program (by location).
When an error occurs, MPROLOG will execute the predicate that the programmer has specified for the given type of error. Since more than one error-handler is allowed for the same error, MPROLOG will obey the youngest error-handler. The error-handler itself may succeed or fail. If it succeeds, then MPROLOG resumes execution using the predicate that it would have executed had the error not occurred in the first place. Otherwise, i.e. if the error-handler itself fails, MPROLOG resumes execution with the failure of the evaluation causing the error.

The user may request that MPROLOG propagates the error to an outer level of program execution by using the predicate "raise-error", which causes the evaluation of the predicate "error-protect". Note that for those situations where error-handling cannot be treated by error type, MPROLOG propagates the error in any case to the nearest surrounding evaluation of the predicate, "error-protect".

Section 2. GENERAL DISCUSSION ON EXCEPTIONS IN PROLOG

The main issues in exception handling schemas concern:

- how is an exception declared
Not every language that handles exceptions declares them. A declaration is very useful as it defines the scope, that is the validity range, of an exception. Static checks may be provided during compile time.

- how is an exception raised
exceptions may be signalled and handled at two levels:
- in the language (so-called default or predefined or system defined)
- in the program (user defined)

During the execution of the program, if an exception arises, it may be signalled:
- automatically (for predefined exceptions an implicit test is provided by the interpreter or compiler)
-explicitely (for user defined exceptions, an explicit and appropriate construct has been provided by the language)
It may be possible to raise the same exception many times in the same program; it is just a matter of overloading.

- how do we specify the action to be executed after the signalling of an exception (exception handler)
The mechanism which is able to handle the exception is called exception handler. It is to the handler that the control flow is transferred after the signalling of an exception. Generally the scope of the handler depends on the granularity of the language.
In logic programming languages, we may distinguish:
- an implicit approach: There is no real recovery provided, since a system state is entered after the occurence of an exception and the handler can either abort the execution or ignore the exception. This mechanism is provided by Prolog-1 and treats only ´escape´exceptions (see [Good75] for more detail about the term ´escape´).
- an explicit approach: the handler consists of a predicate which is the new goal to take over from the ongoing process. This new goal itself may succeed or fail, providing resumption in any case (like in MProlog) or presenting a menu of options like in Prolog-2. It is worth to notice that while traditional programming languages like Ada, Clu, Chill, consider only ´escape´ exceptions, logical programming languages offer a complete treatment of exceptions. Infact they deal also with ´retry´ [CoDu82a] exceptions (in Prolog-2 there is the possibility to recover from some hardware error) and ´notify´ [CoDu82b] ones (a real recovery is obtained when the bad goal is substituted by a succeeding one and the execution continues with the evaluation of the predicate as if no error occurred).

- how is an exception bound to the handler (dynamically or statically)
The binding is the mechanism that associates the signalling of an exception to the corresponding handler. In logic programming languages we may have

- *a static treatment of exceptions:* this means that no recovery can be provided, because no jump to another place in the proof tree can be performed on resumption. The following possibilities can be considered:
a) abort: causes the abandon of the program current activity, and the return to the toplevel interpreter. The whole proof tree is dropped, no debugging can longer be done.
b) local jump: Continuation through evaluation of the next current goal by execution of the right or left branch of the proof tree. No resumption is possible because no scope has been involved, and the kind of recovery is somehow just ´local´:
b.1.) immediate left: causes normal backtracking
b.2.) immediate right: causes normal continuation

- *a dynamical treatment of exceptions:* this means that there is a real possibility to recover the program since a new scope may be defined marking the exception definition. In other words the state of the database is changed as before, but a jump to the mark can be performed so that the recovery can be done at the beginning of the scope as well as before or after the actual goal. More precisely the following possibilities are available (case a and case b are the same as in the static treatment):
a) abort
b) local jump: (immediate left/ immediate right)
c) global jump: a jump to the place where the mark-goal can be found (on the goalstack) is performed. Continuation means evaluation of the right or left goal besides the mark-goal. Resumption is possible because some scope has been involved, the kind of recovery is ´global´, because the whole proof tree is involved.
c.1) dynamical left: causes backtracking to the mark-goal (including the mark-goal)
c.2.) dynamical right: causes backtracking to the mark-goal (not including the mark-goal) and continuation with the right logical successor of mark-goal

- where does the control flow after an exception is handled
In logical programming languages considering the control flow after the handler has finished his task means to look at which branches of the proof tree shall remain. After the occurrence of an exception the following may occur:
a) abort: There is no succeed or fail, just this command causes the interpreter to abort execution forcing a return to the operating system. It is generally performed for unrecoverable errors and therefore it causes a Prolog session to terminate.
Example: exceeding maximum number of allowable atoms.
b) continuation: executing the next goal at the left or right side of the predicate where exception occurred; it corresponds to replacing the whole subtree, where the exception occurred, by fail or by true respectively.
d) resumption with a recovery of the goal. Since there exists a mark on the exception definition, it is possible to define a scope which is the whole proof tree that dynamically follows. It corresponds to substituting the whole subtree dynamically below the mark by true or fail.

In what follows we regroup the above central issues according to the adopted solutions, identifying in such a way 4 different model-groups. The first three groups serve to motivate the fourth which overcomes the deficiencies of the previous ones combining their advantages.
The new mechanism we propose in the following Section 3 can be considered in the context of the fourth of these model-groups.


```
interpreter :-
    select_step (Focus,Type,Condition),
    knowledge_source(Name,Type,Condition),
    execks (Name,Focus),
    !, interpreter.
```

Un ciclo di questo tipo e' stato utilizzato nel sistema esperto per la fusione dati, sopra citato [5].

Controllo Knowledge-scheduling.

Le caratteristiche di questo tipo di controllo sono:

- un KS opera cambiamenti su una o piu' parti del blackboard
- ogni KS specifica il contributo che puo' offrire rispetto al nuovo stato della soluzione e di conseguenza ne viene selezionato uno
- il controllo sceglie un insieme di oggetti del blackboard come argomenti dell'attivazione del KS selezionato.

Interprete relativo:

```
interpreter :-
    select_source (knowledge_source(Name,Type,Condition)),
    select_object (Name,Focus),
    execks (Name,Focus),
    !, interpreter.
```

E' possibile definire controlli di tipo diverso; potrebbe essere per esempio interessante dotare il sistema di un modulo che gestisca un backtracking intelligente, utilizzando il meccanismo di ereditarieta' tra i mondi. Si osservi che una caratteristica molto potente dell'architettura e' la possibilita' di contenere piu' moduli di controllo da poter attivare in fasi diverse dell'elaborazione.

Deletion of marks is done automatically on backtracking. Exception raising can be done by an extra command. The resumption may refer to the place of marking.

Advantages: System and user defined exceptions can be treated in a uniform way. Exception-definitions remain valid after a query has finished. A list of exceptions can be given by the system. The state of exceptions can be saved. Recovery can be done in a global way, as a jump can be performed everywhere. Marks are automatically backtracked and need no housekeeping.

Disadvantages: The cleaning up of definitions must be done by the user, but an existing definition is no burden, if there exists no mark for it.

This approach seems so promising that we choose it for our proposal.

Section 3. EXCEPTION HANDLING IN VIP

Similary to what happens for MProlog, Prolog-1, Prolog-2, the VIP system [Kral87] too recognizes the importance of the ability to deal with exceptions and requires a mechanism for it. We therefore designed a new mechanism for dealing with exception handling in VIP, based on the idea of taking advantage of both statical and dynamical treatment of exceptions (see section 2.4.) yielding as a consequence an hybrid behaviour.
In order to give a clear description of the proposed mechanism we distinguish three components encompassing static versus dynamic features and the exception handler.

Section 3.1. STATIC PART OF VIP EXCEPTION MODEL

Definition: -
The command for defining an exception is

```
def_exception(Exception_name, Action_predicate)
```

For example:

```
def_exception(file_write_error, writescratch(Text)).
where writescratch is defined as
writescratch(X) :- tell(scratchfile), write(X).
```

The def_exception command performs a mapping between the name of the exception and the action to be performed by the handler. This means that def_exception forms the static part of our exception model. New definitions are added to the beginning of the Prolog database, changing the state of the database. Overloading is thus automatically performed as the handler just invokes the first definition that he finds in the database. Def_exceptionis is used as a command:

```
?- def_exception(E,A).
```

From now on the state of the VIP system becomes aware of the new definition. To get information about the existing exceptions the command

```
?- ask_exceptions(X,Y).
```

may be used that gives static information about all system and user defined exceptions in the VIP system.

Exception_name may be:
- a reserved name of a system exception. The default system action provided for this exception is then overloaded by the action provided by the user.
- a user defined term that is not a member of the list of system exception names. This kind of exception can never be raised by the system. If the term is a structure with arity greater or equal to one, then variable arguments are local to the clause where the def_exception goal appears. This term may serve to interchange arguments with the action predicate and with the place of raising (see figure 1).
- a Prolog variable at at the moment of definition. This serves as a ´catch all´ for all exceptions in the system.

Action_predicate may be any (backtrackable) Prolog predicate. The handler tries to satisfy this predicate. The control flow then depends on failure or success of the predicate.
Action_predicate can either be defined directly in the def_exception command, or it can be the head of a Prolog clause. Parameters that appear in the action goal are considered local to the clause where def_exception appears.

Example:

```
def_exception(overflow, (action1 :- write(error)).

is exactly the same as:

def_exception(overflow, action2).
action2 :- write(error).
```

Action_predicate supports parameter passing (see figure 1).

Deletion:
For deletion we provide the command:

```
revoke_exception(Exception_name, Action_predicate).
```

The first exception, the definition of which is unifiable with the arguments of the revoke command, is dropped.

Raising:
An exception may be raised by the system or by the user. The system automatically is aware of exceptions while the user needs a special command for it:

```
raise_exception(Exception_name).
```

For example:

```
ground(I) :- int(I), !.
ground(I) :- atom(I), !.
ground([H|T]) :- ground(H), ground(T).
ground(S) :- S =.. L, ground(L).
ground(T) :- write(T), raise_exception(notground).

testground(X) :- def_exception(notground, diagnostics), ground(X).

where the action predicate diagnostics is defined as:
```

```
diagnostics :- write('term is not ground, abort ...'), abort.
```

If testground is started with a nonground term then the exception notground is raised, an error message is written to standard output, and the proof is aborted.

Remark: as mentioned above, Exception_name can be an arbitrary Prolog term, thus enabling the user to pass arguments.

Section 3.2. DYNAMIC PART OF VIP EXCEPTION MODEL

Marking:
In order to be able to perform resumption we provide the built-in predicate

```
mark(Mark).
```

Mark can be any Prolog term. It has nothing in common with Exception_names defined in the system. The aim is just to mark the place where the control flow shall go on, after the handler has executed an Action_predicate. The marked scope is the whole proof tree that dynamically follows.

Overloading of marks for the same name is allowed : the youngest definition is valid. On backtracking older definitions are detected.

The mark can be seen as the label of a goto-command. The jump is defined by the user in the body of the Action_predicate: There are two goals that can be used in the body of the action predicate: abort or resume. Abort causes that the whole proof is aborted, it normally exists as a standard built-in predicate in common Prolog systems. The task of resume is to jump to the mark defined somewhere before in the proof tree by means of mark:

```
resume(Mark, true).
resume(Mark, fail).
```

The exact description for the semantics of the arguments for resume is given in section 3.3. Marking supports parameter passing (see figure 1).

Deletion:
The mark command needs no extra command for deletion. It is automatically deleted on backtracking by the VIP interpreter.

Raising:
For raising an exception with a dynamic mark we may use the same command as before. For the user there is no difference in raising exceptions in the static or dynamical model. Raise_exception supports parameter passing:

programmazione logica, che avvertano la mancanza di un ambiente ben strutturato e soprattutto flessibile per facilitare lo sviluppo di programmi intelligenti.

La generalita' della shell deriva dal fatto che in letteratura non esiste un numero di blackboard shells realizzate in Prolog tale da definire un modello preciso dell'ambiente e delle sue funzionalita'. Inoltre, storicamente, la progettazione dei blackboard framevork esistenti e' stata pilotata dalle applicazioni; sembra quindi ragionevole sviluppare un nucleo minimale di funzionalita' da arricchire, se necessario, con altri strumenti o con nuovi componenti, sulla base dell'esperienza derivata dall'utilizzazione dell'architettura per la soluzione di diversi problemi reali.

Un altra ragione e' quella che il Prolog [4,11] a differenza di altri linguaggi usati per l'Intelligenza Artificiale, come il Lisp, e' di per se' un potente formalismo per la rappresentazione della conoscenza; e' possibile definire con poco sforzo dei motori inferenziali ad hoc e formalismi diversi per la rappresentazione della conoscenza.

Si da' ora una descrizione dettagliata dell'ambiente: la prima parte e' una descrizione logica, sono cioe' specificati i tipi dei componenti del sistema e le possibili interazioni tra essi, la seconda parte descrive come questa struttura logica e' legata al modello a blackboard ed al linguaggio logico Prolog.

2.1 La struttura logica dell'ambiente

L'ambiente e' composto da due tipi di componenti principali (attivi e passivi) e da uno o piu' moduli di controllo che specificano il modo in cui i componenti devono essere usati.
I componenti passivi si dividono in due tipi: quelli *statici* sono usati per realizzare la memoria a lungo termine e possono essere modificati solo a livello del controllo; i componenti *dinamici* realizzano la memoria a breve termine, permettono ai componenti attivi di comunicare tra loro e contengono lo stato attuale della soluzione.
I componenti attivi contengono la conoscenza di dominio e cooperano scambiandosi informazioni attraverso la memoria a breve termine; questi componenti contengono la conoscenza operativa sul dominio.
Il controllo fornisce tutti i meccanismi per gestire l'attivazione dei componenti attivi sulla base delle informazioni contenute nei componenti passivi, ed i meccanismi per la gestione dei componenti passivi; e' possibile, per esempio, trasferire informazioni dalla memoria a breve termine a quella a lungo termine.

```
dynamic left:    do_something :-  write(´this was the wrong way´),
                                 write(´continue with b´),
                                 resume(xxx, true).
dynamic right:   do_something:-  write(´retry c´),
                                 resume(xxx, fail).
```

Section 4. EXAMPLE

In this section an example is given in order to illustrates the use of the structure.

Example 1: We show how the concept of transactions can be implemented with the help of VIP exceptions in a more elegant way. A transaction consists of preconditions, that must be fulfilled before the main part of the transaction can be performed. Secondly, the main part of a transaction consists of database commands. Lastly a transaction holds postconditions, that must be checked, after the main part has been executed. If one postcondition fails, the whole transaction must be undone. In our example we assume a predicate ´undo(Goal)´ that is able to rollback a goal. Furthermore we do not consider the case, that a database command may fail.

```
/* static part */
def_exception(rollback, clean_up).
clean_up :- (computed(X), undo(X), fail);
            (retractall(computed(_)),  resume(undomark, fail)).

/* dynamic part */
transaction(PreconditionList, DB_CommandList, PostconditionList) :-
            mark(undomark),
            trans(PreconditionList, DB_CommandList, PostconditionList).
trans(Pre, Com, Post) :-    test(Pre),
                            execute(Com),
                            test(Post).
test([]).
test([H|T]) :- call(H), test(T).
test(_) :- raise_exception(rollback).
execute([]).
execute([H|T]) :- call(H), asserta(computed(H)), execute(T).
```

The execute predicate protocolls its work, by inserting every called goal to the Prolog database. The action-predicate clean_up can then use this information to perform the rollback.

Example 2: We expand the above example by providing an exception for the case that the disk is full. The desired resumption is to clear files, that are no longer used, on the disk, and then to retry the whole transaction. Information about what files may be cleared is interactively interrogated from the user. This example shows how parameters can be passed from the place of raising to the handler. In our case this information comprises the name of the disk that has run out of space.


```
/* static part */
def_exception(diskfull(DiskName), try_clear_file(DiskName)).
try_clear_file(DiskName) :-  write(DiskName),
                             write(´is full, what shall I clear ? ´),
                             readfilenamelist(Flist),
                             dropfiles(Flist),
                             resume(undomark, true).

/* dynamic part */
...,   dbask(result(A,B,C) :- relation1(A,B,C)),
       dbinsert(relation2(A,B,C),
       dbinsert(relation3(C,B,A), ...

/* dbinsert raises the exception:

       raise_exception(diskfull(floppy_1))

if there is no space to write on the floppy disk. Resumption is performed undoing the last
transaction, making space on disk, and retrying the whole transaction */
```

CONCLUSIONS

In this paper we presented a new structure for exception handling which is suited for logic programming languages and supports the construction of reliable software. Compared to similar constructs of existing languages, like Prolog-1, Prolog-2 and MPROLOG it provides considerable advantages such as:

STATIC DEFINITION: the explicit declaration of exception supports conventional static checking, and allows one to catch undeclared exceptions. Moreover the listing of existing exceptions is available.

DYNAMICAL TREATMENT: it is the only approach that offers a real recovery of the program from errors.

PARAMETER PASSING: the presence of parameters associated to exceptions may be useful for the treatment, since they may supply information on how an exception occurred. In VIP we provide two mechanisms for parameter passing: from the place of marking to the place of raising (and vice versa), and from the place of raising to the handler (and vice versa).

COMPLETENESS OF TREATMENT: the VIP exception mechanism covers all the features that appeared separately in other implementations. Moreover it provides parameter passing and an explicit propagation.

DEFAULT EXCEPTIONS: they do not contrast with our structure.Their handlers may be infact overridden by an explicit handler associated to the same exception name. In such a way it is possible to disable default exceptions and consequentely to allow the user to define different handlers for predefined exceptions. Further uniqueness of the handler is ensured, since a mapping between the name of the exception and the explicitely defined handler (action predicate) is created.

As we can deduce from the brief review of the existing exception handling mechanisms in logic programming languages, our structure is much more powerful than the one offerred by using an implicit approach (Prolog-1), it presents all the advantages offered by the statical treatment (Prolog-2), and, compared with the dynamical treatment of

MPROLOG, it allows much more flexibility of the hanldler, permitting the mark to be set everywhere in the proof tree. The best advantage of that is the realization of a clean mechanism for propagating exceptions. Infact, by means of a raise_exception statement inside an action_predicate and a mark statement somewhere in the proof tree, we achieved propagation in a very simple and explicit way.

References

[CoDu82a] Cocco, Dulli. A Mechanism for Exception Handling and its VerificationRules. Computer Languages, Vol.7, 1982

[CoDu82b] Cocco, Dulli. Costrutti per la gestione delle eccezioni: confronto tra Clu, Chill e Ada. Rivista di Informatica, vol.XII, n.3, 1982.

[GhJa82] Ghezzi, Jazajeri. Programming Language Concepts. Wiley & Sons, Inc., 1982.

[Good75] Goodenough. Exception Handling: Issues and a proposed Notation. C.A.C.M. vol. 18, n.12, 1975.

[Harp86] Harper. Introduction to Standard ML. ECS-LFCS-86-14, nov. 1986

[HMQM86] Harper, MacQueen, Milner. Standard ML. ECS-LFCS-86-2, march 1986.

[Kral87] Krall. Implementation of a high-speed Prolog Interpreter. ACM Proceedings of the SIGPLAN 87 on Interpreters and Interpretative Techniques, St. Paul Minnesota, 1987

[MProlog] MPROLOG Language Reference, Release 1.5, Logicware, Toronto, Nov.1984.

[Prolog1] Prolog-1 8086 Reference manual, Expert Systems Limited, Oxford, Dec.1983.

[Prolog2] Prolog-2, Technical Reference Manual, Expert Systems Limited, Oxford, 1985.

Applicazioni di un Linguaggio Logico + Funzionale di Ordine Superiore

P.G. Bosco, C. Cecchi, C. Moiso

CSELT

Centro Studi E Laboratori Telecomunicazioni

via Reiss Romoli 274 - 10148 Torino

Sommario

Le caratteristiche di IDEAL - un linguaggio integrato logico-funzionale di ordine superiore - sono evidenzate nell'articolo in due momenti: (a) s'illustra la programmazione effettiva di un *simulatore logico*, mettendo cosi` in risalto gli aspetti dei linguaggi funzionali (mancanti in quelli logici) che si rendono necessari nella programmazione dichiarativa; (b) si mostra la "invertibilita`" dei programmi (funzionali) IDEAL, per cui il simulatore logico e` immediatamente utilizzabile come un *fault-finder*, il che mette in luce un aspetto fondamentale dei linguaggi logici (mancante in quelli funzionali) che ancora e` di notevole interesse per la programazione dichiarativa.
IDEAL e` realizzato mediante compilazione in un linguaggio del I ordine, che e` Prolog stesso se si vuol ottenere una valutazione eager (call by value), oppure K-LEAF - un linguaggio logico+funzionale del I ordine - se si desidera ottenere una valutazione lazy (call by need). Il linguagio K-LEAF si traduce in codice WAM (Warren Abstract Machine) mediante una opportuna estensione del procedimento di compilazione e dell'insieme d'istruzioni della WAM, cosi` da implementare una regola di selezione dinamica.

1. Introduzione

La programmazione logica e quella funzionale sono i due stili piu` diffusi di programmazione dichiarativa ed e` tuttora in corso il dibattitito sui rispettivi pro e contro; una soluzione per superare questa discussione e` quella di combinare i due paradigmi e quindi sviluppare un linguaggio che contenga i loro aspetti positivi, eliminandone gli svantaggi.

Tra le caratteristiche peculiari dei due paradigmi, una delle piu` significative differenze tra i linguaggi funzionali e quelli basati sulle clausole di Horn e` la presenza nei primi di costrutti di ordine superiore, un potente strumento che puo` essere sfruttato nella cosiddetta programmazione *in the large*, nel sintetizzare programmi da specifiche, ecc. .

Inoltre, i linguaggi funzionali offrono una varieta` di altri utili concetti di programmazione (come i sistemi di tipi, le differenti strategie di riduzione, ecc.) che