Multilanguage Interoperability

Giuseppe Attardi and Mauro Gaspari

Technical Report UBLCS-93-18

24th of July 1993

Laboratory for Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)

The University of Bologna Laboratory for Computer Science Research Technical Reports are available via anonymous FTP from the area `ftp.cs.unibo.it:/pub/TR/UBLCS` in compressed PostScript format. Abstracts are available from the same host in the directory `/pub/TR/UBLCS/ABSTRACTS` in plain text format. All local authors can be reached via e-mail at the address *last-name*`@cs.unibo.it`.

UBLCS Technical Report Series

92-1 *Mapping Parallel Computations onto Distributed Systems in Paralex*, by Ö. Babaoğlu, L. Alvisi, A. Amoroso and R. Davoli, January 1992.

92-2 *Parallel Scientific Computing in Distributed Systems: The Paralex Approach*, by L. Alvisi, A. Amoroso, Ö. Babaoğlu, A. Baronio, R. Davoli and L. A. Giachini, February 1992.

92-3 *Run-time Support for Dynamic Load Balancing and Debugging in Paralex*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, September 1992.

92-4 *Paralex: An Environment for Parallel Programming in Distributed Systems*, by Ö. Babaoğlu, L. Alvisi, S. Amoroso, R. Davoli, L. A. Giachini, October 1992.

93-1 *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanism*, by Ö. Babaoğlu and K. Marzullo, January 1993.

93-2 *Understanding Non-Blocking Atomic Commitment*, by Ö. Babaoğlu and S. Toueg, January 1993.

93-3 *Anchors and Paths in a Hypertext Publishing System*, by C. Maioli and F. Vitali, February 1993.

93-4 *A Formalization of Priority Inversion*, by Ö. Babaoğlu, K. Marzullo and F. Schneider, March 1993.

93-5 *Some Modifications to the Dexter Model for the Formal Description of Hypertexts*, by S. Lamberti, C. Maioli and F. Vitali, April 1993.

93-6 *Versioning Issues in a Collaborative Distributed Hypertext System*, by C. Maioli, S. Sola and F. Vitali, April 1993.

93-7 *Distributed Programming with Logic Tuple Spaces*, by P. Ciancarini, April 1993.

93-8 *Coordinating Rule-Based Software Processes with ESP*, by P. Ciancarini, April 1993.

93-9 *What is Logic Programming good for in Software Engineering*, by P. Ciancarini and G. Levi, April 1993.

93-10 *Scheduling Real Time Tasks: A Performance Study*, by F. Panzieri, L. Donatiello and L. Poretti, May 1993.

93-11 *Specification and Detection of Behavioral Patterns in Distributed Computations*, by Ö. Babaoğlu and M. Raynal, May 1993.

93-12 *Interaction Systems II: The Practice of Optimal Reductions*, by A. Asperti and C. Laneve, May 1993.

93-13 *Reliability Analysis of Tree-Based Structures and its Application to Fault-Tolerant VLSI Systems*, by Marco Roccetti, June 1993.

93-14 *Guard Compilation in Logic Shared Dataspace Languages*, by M. Garpari, June 1993.

93-15 *Data Algorithm: A Numerical Method to Extract Shape Information from Gray Scale Images*, by R. Davoli and F. Tamburini, June 1993.

Multilanguage Interoperability[1]

Giuseppe Attardi[2] Mauro Gaspari[3]

Technical Report UBLCS-93-18

July 1993

Abstract

We present an approach to the interoperability of programming languages, based on a Common Runtime Support, which provides general mechanisms for storage management, symbol table management and concurrent execution, for modern high level languages. We focus in particular on the CRS approach to the interoperability of Prolog and Common Lisp. The CRS provides support for logic variables, so that both a Lisp Abstract Machine and a subset of the Warren Abstract Machine, including all the unification primitives, are implemented on it. We present and compare two alternative implementation of backtracking, through success continuations and failure continuations. Both Lisp and Prolog programs are compiled to C code to run on the C based CRS. The interoperability is achieved with minimal overhead and this allows programmers to select the most appropriate programming paradigm for each task: functional, logic and object–oriented.

---

1. An abstract of this report appeared in the proceedings of PLILP'91, 3rd International Symposium on Programming Language Implementation and Logic Programming, Springer-Verlag, LNCS, n. 528, 1991.

2. Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I–56125 Pisa, Italy, net: attardi@di.unipi.it

3 Dipartimento di Matematica, Università di Bologna, Piazza S. Donato, 5, I-40127, Bologna Italy, net: gaspari@csr.unibo.it

1 Introduction

The interoperability of programming languages is increasingly becoming an essential need. As applications become more elaborate, the facilities required to build them grow in number and sophistication. Each facility is accessed through a specific package, quite complex itself, like in the cases of: modeling, simulation, graphics, hypertext facilities, data base management, numerical analysis, deductive capabilities, concurrent programming, heuristic search, symbolic manipulation, language analysis, special device control.

Reusability is quite a significant issue: once a package has been developed, tested and debugged, it is undesirable having to rewrite it in a different language just because the application is based in such other language.

One cannot expect that all useful facilities be available in a single language, since for each task developers tend to prefer the language which provides the most appropriate concepts and abstractions and which supports more convenient programming paradigms. This is specially true in the field of AI, where a number of innovative programming paradigms have been developed over the years.

On the other hand, it would be quite important for the success of AI facilities, which are often built using specialized languages, that they could be accessible from other languages. Given the unusual execution environment requirements for AI languages, the lack of interoperability has so far limited such possibility.

Several approaches have been proposed to the problem of combining code from different languages [Atk89]:

- client/server model with remote procedure calls. Each language executes in a separate process maintaining its own representation of data; external calls go through a remote procedure call protocol. This, however reduces efficiency and requires transcoding of the parameters to a common external data representation.
- foreign function call interfaces. This requires a common base language to which others must conform. Limitations exist though on types of parameters which can be passed to foreign procedures and during debugging it becomes difficult to trace the execution of programs in the foreign language.
- common intermediate form on which all languages are translated. An example is the Poplog Abstract Machine [Mel86], on which different languages (Common Lisp, Prolog, ML and Pop11) are translated. This puts too severe restrictions on the language designers and implementors for wide use.

Only the last approach achieves tight interoperability among languages, i.e. procedures in one language can invoke procedures in another language and viceversa, and data can be shared or passed back and forth between procedures in different languages, without the overhead of transforming data representations.

The approach we are proposing achieves tightly coupled interoperability, requiring only an agreement on some essential components of the run time environment. A quite significant advantage of our approach is that the interopreability is bidirectional, in the sense that not only languages with more sophisticated facilities (like memory management) can call procedures of less sophisticated languages, but also the opposite direction is supported, allowing for instance a C based application to call a package developed in Prolog or LISP.

2 Common Runtime Support

Our approach to interoperability is based on an intermediate support layer between the operating system and the high level programming language. This layer is called CRS (Common Runtime Support), and provides the essential functionalities which can be abstracted from modern high level languages. The CRS uses C as the common intermediate form for all languages. Such usage of C has been applied successfully to several programming languages, for instance Cedar, Common Lisp and Scheme at Xerox PARC [Atk89], Modula3 and C at DEC SRC, Linda and also to Prolog [Wei88]. Though convenient for portability, the use of C as intermediate form is not essential to the approach, and in fact the fundamental requirements are agreement on procedure call conventions and access to the facilities provided by the CRS (memory, I/O and processes) only through the functional interface provided by CRS.
The CRS provides the following facilities:

- storage management, which provides dynamic memory allocation and reclamation through a *conservative* [Bar88] garbage collector
- symbol table management, including dynamic linking and loading, and image dumping
- support for debugging, allowing to debug mixed code programs through a single debugging tool
- multiple threads [Att87], to support concurrent programming constructs
- generic low level I/O, operating both on files and on network streams
- support for logic variables and unification
- support for nondeterminism

The last two facilities have been added to support Prolog, and will constitute the focus of this paper. Figure 1 shows the CRS in relation with other elements of a programming environment.

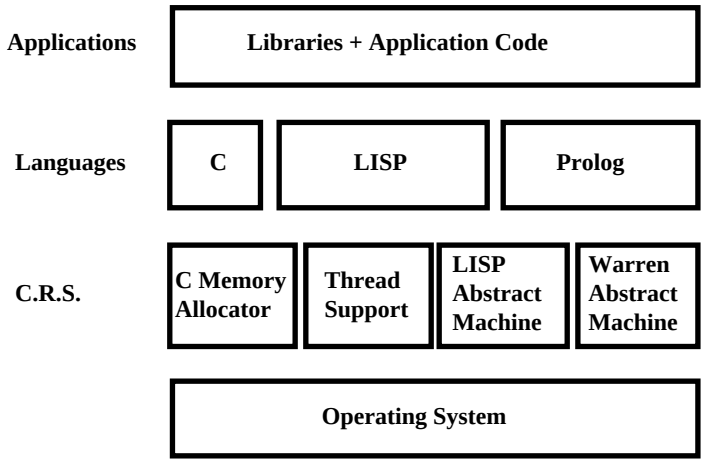

Figure 1: Common Runtime Support.

The Lisp Abstract Machine as well as the Warren Abstract Machine are provided as a library of C routines and data structures. The COMMON LISP compiler translates Lisp code to C, which is then

handed over to the native C compiler to produce executable code. Similarly Prolog is translated to C through an intermediate Lisp form.
One consequence of the CRS approach is that a single programming environment is used for all languages. Therefore tracing, stepping and debugging are handled by a single set of tools. Some additional utilities would be helpful though, in order to be able to display or access data structures during debugging in the specific notation of each language. For the moment we provide a few commands which are oriented to the debugging of Prolog code, which are based on the technique of meta interpretation.
Using the same data types for Lisp and Prolog provides tight interoperability which allows sharing of data between the two languages:

- Prolog atoms coincide with Lisp symbols
- Prolog and Lisp lists are the same
- Logic variables are implemented as locatives
- Prolog compound terms correspond to Lisp structures, where the functor name is represented as the structure name.

From the identity between Prolog atoms and Lisp symbols derives that atoms inherit the package mechanism of COMMON LISP, which allows to partition symbols into several (disjoint or overlapping) namespaces. This can be conveniently used for instance to partition into groups the database of predicates for Prolog.
In our implementation COMMON LISP structures and CLOS [Bob 88] objects have the same internal representation. As a consequence, any CLOS instance can be passed to a clause, and by unification its slots can be accessed or assigned.

2.1 WAM Instructions

The WAM [War83] consists of several data areas and a Prolog specific instruction set. Prolog predicates are compiled into sequences of WAM instructions, roughly one instruction for each Prolog symbol. In the CRS, some of the facilities required by the WAM are shared with other languages, and a few are specific to the WAM, as follows:

- the *heap* is dealt by the storage management of the CRS.
- code is stored in the heap and linked/loaded through the mechanisms of the CRS.
- the local stack is the same for Lisp and Prolog. It holds only variables, not choice points which are kept in the continuation stack. Actually there are several local stacks, one for each thread.
- the *trail stack* is implemented as a statically allocated C array.
- *get* and *unify* instructions have been added as part of the CRS.
- *put* and *procedural* instructions are unnecessary since standard parameter passing conventions are used for procedure invocation.
- *indexing* instructions are dealt through local/non local control transfer constructs.

2.2 Locative Data Type

A logic variable is basically a place holder for a value, and variables get bound to values during unification. Such bindings may have to be undone later if a failure in the deduction requires backtracking or a search for an alternative path to a solution. During deduction, several new variables are generated at each deductive step. Therefore a significant saving of space and time can be obtained if no space is allocated for these variables. This can be achieved by representing them

as pointers to the locations which contain the slot or the value, and to which the variable is bound. A special data type, called *locative*, is provided by the CRS, to implement these temporary variables. A *locative* is similar to the locative data type of Lisp Machines [Kah86] and is a pointer to a single memory cell, which can be either a slot within a structure or the cell for another variable. The garbage collector recognizes locatives and avoids to follow them during the mark phase, on the assumption that they always point to cells within structures which are reachable otherwise for marking by the garbage collector, as it is guarantied in the code produced by the Prolog compiler.
Locatives are implemented as *immediate* data, so they do not use memory from the heap. To accommodate immediate data, the CRS uses a tagging schema, where the two low order bits of a pointer denote the type of the object. The memory allocator of CRS ensures that all objects allocated on the heap are rounded to a size which is a multiple of 4, therefore a legal reference to a heap allocated object must have zeros in the last two bits of the address. The tagging schema is designed to fit also other kinds of data which are conveniently handled as immediate data, like small integers (Lisp fixnums) and characters.

2.3 Nondeterminism

Nondeterministic computations are typical of search problems, and so languages oriented to problem solving provide mechanisms to support them. A nondeterministic expression returns producing one result from a set of choices but may be later forced to resume execution to produce a different value to the original caller. Since nondeterministic expressions can be nested, a search tree is produced. Searching this tree in a depth–first manner is the most economical solution, and backtracking is the mechanism which supports this strategy.
Backtracking can be implemented by means of *continuations*, which represent the rest of a computation, in two different ways. The first solution consists in saving the continuation at the time when a nondeterministic choice is made so that it can be later resumed if an alternative choice has to be made. The second solution consists in supplying a continuation, representing what remains to be done, to the nondeterministic expression to be invoked after one choice results successful: if the continuation fails to find a solution, it needs just to return to its caller, the nondeterministic expression, so that it may perform another choice. The first technique is called *upward failure continuations* and the second one is called *downward success continuations* [Coh85].
In the technique of *failure continuations*, a continuation is created after each choice, to represent the whole state of the computation at that point. It would be however rather expensive to create such continuation by making a full copy of the state. We can exploit the fact that backtracking uses these continuations in an orderly way: the last continuation created is resumed first and no more than once. Therefore we need just to ensure that the present control stack is preserved, so that we can go back to it. This is done through a machine language primitive, which moves the SP (Stack Pointer) to ensure that any function to be called later will work on a portion of the stack past the choice point. The following picture illustrates this mechanism. Three stacks are present: the ordinary *C stack*, where the current frame is identified through the machine registers FP and SP, the *Failure stack*, identified by FSP, and the *Trail stack*, used for recording variable bindings, identified by TSP. The last two stacks grow against each other. The picture illustrates three stages through the execution of a clause like `P :- Q, R`. When `Q` is executing, frames for `P` and `Q` are present on the C stack; when `Q` succeeds, it pushed a failure continuation on the failure stack, and returns to `P`; `P` extends its stack frame to include that of `Q`, and then proceeds in calling `R`.

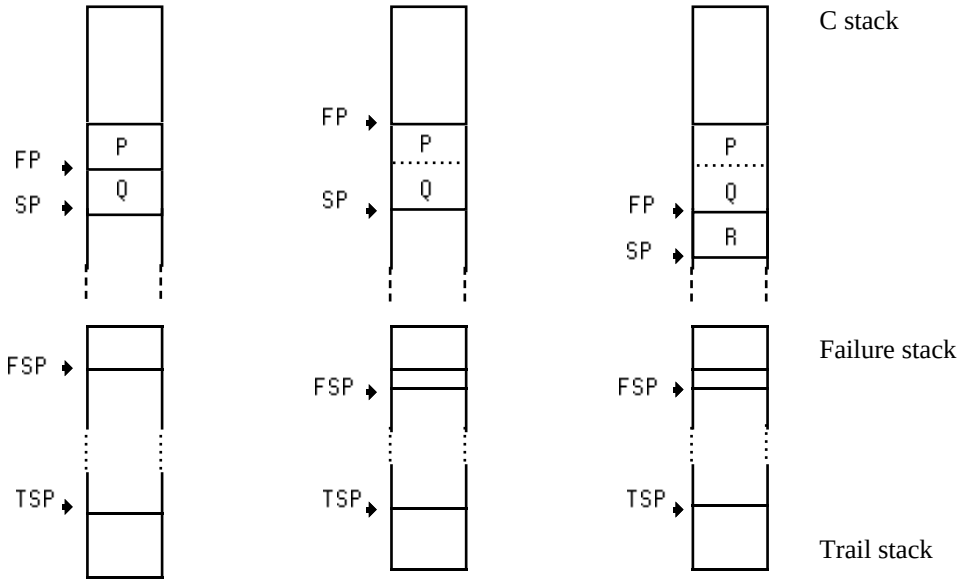

Figure 2: Failure Continuations.

The CRS implements the mechanism for failure continuations through the following primitives (actually C macros):

`SUCCEED`	pushes a failure continuation into the failure stack and returns `SUCCESS` to the caller
`FAIL`	transfers control back to last failure continuation
`FRAME_SKIP`	preserves the stack frame containing the last failure continuation

In figure 3 there is an example of a nondeterministic choice function which returns, in the location pointed by its first argument, successive values selected from the list in its second argument. A fancy way of iterating through the elements of a list might be expressed using nondeterminism.

```
bool choose (x, list)
object *x, list;
{  while (list != NIL) {
       *x = head(list);
       list = tail(list);
       SUCCEED;
   }
   return(FALSE);
}

if (choose(&y, alist) == SUCCESS) {
   FRAME_SKIP;
   … do something with y …
   FAIL;}                                  /* and then backtrack */
```

Figure 3: Nondeterminism.

We distinguish three possible return values for nondeterminsitic functions: `TRUE`, `FALSE` and `SUCCESS`. The last value is used to distinguish the case when a successful choice has been made but there may be alternative choices for which a failure continuation has been set up. This information is helpful to distinguish between determinate and nondeterminate predicates in Prolog, allowing for certain optimizations to be performed. Backtracking for Prolog can be split into two parts: restoring the value of any Prolog variable to its state before the last alternative was selected, and transfering control to the next alternative. For the first task we use, as customary, a *trail* stack, were each location affected by unification is recorded. Upon backtracking, all such locations are restored to unbound.
For transfering control one can use either success or failure continuations. In [Att90] we discussed a solution based on success continuations, here we present a solution based on failure continuations and compare the two alternatives.

2.4 Failure vs success continuations

In the technique of *success continuations*, each Prolog predicate is translated into a boolean function with as many arguments as the predicate plus an extra one, the continuation, which is a function of zero arguments, representing any remaining goals carried over from previous calls, i.e. the resolvent except the first goal. The continuation in general has to be a *closure*, since it usually must refer to some arguments of the predicate. To achieve interoperability with COMMON LISP, [Att90] used Lisp closures as success continuations. A representatio build through assembly language procedures is used instead by [Wei88].
If an atomic clause of the predicate succeeds, the continuation is invoked to carry out the remaining goals. If one clause fails, execution proceeds with the following clause. A predicate does not return successfully until a solution has been reached to all the goals of the original problem. In other words, for each attempted goal we have a nesting of function invocation, and so the control stack grows proportionately to the overall number of attempted subgoals. This is a drawback of this approach which may in part be mitigated by detecting and compiling appropriately calls to determinate predicates [Wei88, Mel85].
With the technique of failure continuations, the continuation argument is not needed: predicates are translated into functions which return normally either `TRUE`, or `FALSE` or `SUCCESS`. `SUCCESS` is returned when a failure continuation has been created, i.e. there are still other choices for the predicate, while `TRUE` is returned for the only (or last) choice in a predicate. Therefore determinate predicates return either `TRUE` or `FALSE`. Stack frames for predicates returning either `TRUE` or `FALSE` are immediately reclaimed, and this works both for determinate predicates and for predicates at the last choice.
Another difficulty with success continuations is in the handling of the predicate `cut`. Consider a clause such as:

```
a :- b, !, c.
```

With success continuations, the predicate `cut` is executed in the continuation of `b`, i.e. two or more stack frames down from `a`. These frames may be reclaimed, since one need not return to them, however this is hard to do in a portable way.
With failure continuations, the `cut` can just pop from the continuation stack all the elements since the invocation of `a`.
Another optimization which is impossible with success continuations is discarding the last choice point when entering the last clause in a predicate. With failure continuations a similar effect is

obtained by avoiding to issue a SUCCESS after the last body goal: any subsequent failure will skip this predicate to return to the immediately preceding failure continuation.
Tail recursion optimization can be done for success continuations when the body consists of a single goal or all predicates involved in the body goals are known to be determinate. With failure continuations we can detect during execution determinate predicates and as a consequence perform tail-recursion optimization whenever appropriate.
The approach of success continuations has an advantage in portability, since there are no machine dependencies in its implementation. Instead the FRAME_SKIP primitive, required to implementing failure continuations, is definitely machine dependent.
In [Att90] we described how to exploit mode declaration to produce more efficient code and how to implement indexing into the clauses of a predicate. Such techniques apply equally well to both approaches.

2.5 Examples

The Prolog predicate concat, may be translated to the following C code.

```
concat([], X, X).
concat([X1|X2], Y, [X1|Z2]) :- concat(X2,Y,Z2).

bool concat (x, y, z)
object x, y, z;
{ trail_mark;
  if (get_nil(x) &&                          /* concat([],                  */
      get_value(y, z))                       /*           X, X)             */
      SUCCESS;                               /*                .            */
  trail_restore;
  { object x1, x2, z2;
    if (get_cons(x) &&                       /* concat([                    */
        unify_variable(x1) &&                /*        X1,                  */
        unify_variable(x2) &&                /*           X2],              */
                                             /*        Y,                   */
        get_cons(z) &&                       /*       [                     */
        unify_value(x1) &&                   /*        X1,                  */
        unify_variable(z2))                  /*           Z2])              */
      return(concat(x2, y, z2)));            /* :- concat(X1, Y, Z2).       */
  }
  trail_unmark;
  return(FALSE);
}
```

Figure 4: Concat.

The function trail_mark sets up a choice point by placing a mark on the trail for backtracking. The function trail_restore unwinds the trail stack up to the latest choice point, undoing the bindings for variables recorded in the trail, while trail_unmark performs a trail_restore and removes the choice point mark from the trail.
In figure 5 there is the C translation for naive reverse.

```
int reverse(x, y)
object x, y;
{
    trail_mark;
    if (get_nil(x) &&
        get_nil(y))
       SUCCEED;
L1: trail_restore;
     { object x1, x2, ys = UNBOUND;
       if (get_cons(x) &&
           unify_variable(x1) &&
           unify_variable(x2)) {
                  BEGIN_CLAUSE
                  reverse(CDR(x), ys);
                  COMMA(L2)
                  append(ys, CONS(x1, NIL), y);
                  END_PREDICATE
       }
      }
L2: trail_unmark;
    return(FALSE);
}
```

where:

```
#define BEGIN_CLAUSE int res, successes = 0; res =

#define COMMA(L) \
if (res == FALSE) \
   if (successes == 0) goto L; \
      else {--successes; FAIL;} \
if (res == SUCCESS) {successes++; FRAME_SKIP;} \
res =

#define END_CLAUSE \
if (res == TRUE) {SUCCEED;} else {END_PREDICATE;}

#define END_PREDICATE \
if (successes > 0) {if (res == TRUE) return(SUCCESS); else FAIL;} \
if (res != FALSE) return(res)
```

Figure 5: Naive Reverse.

Notice that the variable `successes` holds the count of the subgoals which have set up some choice point. If its value is 0 before the call to the last predicate in the last clause, we are in a position to perform tail-recursion optimization.

2.6 Implementation

For the interoperability between C and Prolog, the creation of a failure continuation requires to save the state of the C virtual machine, for which the information produced by `setjmp` is usually sufficient:

```
int success() {
  if (setjmp(FSP) == 0) {
    FSP += sizeof(struct jump_buf);
    return(TRUE);}
  else
    return(FALSE);
}

#define SUCCEED       if (success()) return(SUCCESS)
#define FAIL              {FSP -= sizeof(struct jump_buf); \
                           longjmp(FSP, 1);}
```

Figure 6: Implementation.

The function `setjmp` belongs to the standard C library: it saves the current machine state in `FSP` and returns `0`. When `longjmp` is called, the saved state is restored and the execution continues as if the call of `setjmp` had just returned with value `1`.
Not all implementations of `setjmp` are adequate for our purposes: some implementation do not save the full state, but just the incremental change with respect to the previous frame, and `longjmp` is performed by unrolling the stack up to the original position. On some machines, like the VAX, we had to reimplement `setjmp` and `longjmp` to suit our needs. This is already part of the CRS, since the same primitives are used to switch execution between threads.
For the proper handling of determinate predicates, it is essential that unification avoids to bind earlier variables to later variables, therefore creating references from older cells to younger cells in the stack. If a term in a body goal contains a variable present in a later goal, such term must be stored in a local of the predicate, to protect the term from garbage collection and to avoid a dangling reference during the execution of the later goal.

2.7 Performance

There are a few other optimization techniques exploited by Prolog implementations, notably environment trimming, which it may be worth while to add to the present implementation. Some optimizations are ruled out by the very nature of the system. For instance, since we allow Prolog data to be passed to Lisp functions, Prolog terms must be allocated on the heap, therefore not with a stack discipline. Nevertheless the performance of our implementation meets our goal to stay within a factor of 2 to the performance of the best native Prolog implementations. The difference is exclusively due to the higher cost of memory allocation primitives which include also a small overhead of synchronization for multithread. Our benchmarks indicate for instance a performance of 56000 LIPS on a VAX 8530 and 110000 LIPS on SUN4/260.

3 Nondeterminism in Lisp

Once we have added the facilities for nondeterminism to the CRS, it is easy to make them accessible from Lisp and achieve the interoperability between Lisp and Prolog. We represent failure with the symbol `NIL`, success with the symbol `SUCCESS`, and true with any other value. We then introduce a function and a special form:

`(success)` pushes a failure continuation and returns the symbol `SUCCESS` to the caller

`(goals g1 g2 … gn)` this special form executes sequentially the forms `g1`, `g2`, … and succeeds only if all of them succeed. If any one of them fails, the form backtracks to the previous successful goal, and fails if there is none.

The Lisp version of `success` must do additional work with respect to the version presented earlier, since it must save and restore not only the state of the C machine, but also that of the Lisp Abstract Machine: the dynamic binding stack, the non local transfer stack, etc.

As a simple example we show in figure 7 how a compiler could translate to Lisp the predicate naive reverse.

```
(defun reverse (x y)
 (trail-mark)
 (or
  (and (get-nil x)                    ; reverse([],
       (get-nil y)                    ;            [])
       (success))                     ;               .
  (trail-restore)
  (let (x1 x2 (y2 (make-variable)))
    (and
      (get-cons x)                    ; reverse([
      (unify-variable x1)             ;          X1|
      (unify-variable x2)             ;          X2], Y)
      (goals                          ;  :-
         (reverse x2 y2)              ;     reverse(X2,Y2),
         (concat y2 (list x1) y)))))  ;     concat(Y2, [X1], Y).
 (trail-unmark))
```

Figure 7: Lisp Naive Reverse.

3.1 Prolog/Lisp Integration

Integration of the two languages shows up for instance in the implementation of Prolog built–in predicates, most of which are easily implemented using either Lisp constructs or available Lisp library predicates. The Prolog `is` operator, used to perform arithmetic computations, has this implementation:

```
(defun is (var expr)
  (get-value expr var)))
```

Figure 8: Prolog is operator.

Most Prolog built–in operators are turned into calls to an equivalent Lisp library predicate. The same solution applies to user defined Lisp predicates. For convenience, the Prolog operator `lisp_predicate` is available which automatically creates an interface function, suitable for use as a Prolog predicate. For instance in figure 9 a declaration makes the Lisp predicate `<` available to Prolog, as a binary predicate, through the definition of an interface function.

```
:- lisp_predicate(<, 2).

(defun </2 (left right)
  (< (deref left) (deref right)))
```

Figure 9: Lisp predicates from Prolog.

The `deref` primitive returns its argument if it is not a locative, otherwise it dereferences the locative following a chain of locatives until a value is reached, and signals an error if the locative is unbound. A Lisp function can be called directly from Prolog by means of the `is` construct. For instance figure 10 shows how to call a Lisp function `clx:open-display` fron Prolog.

```
hello(H) :- D is clx:'open-display'(H),
            display('Hello', D).
```

Figure 10: Lisp functions from Prolog.

Viceversa, to call Prolog from Lisp, the following macro is provided for conveniently creating variables for use in a query, which may be referred and used within its scope. For instance, the code in figure 11 performs a database selection with grouping, computing the total amount due each month from the relation (predicate) `Account`. The array `table` is filled with the total for each month.

```
(with-logic-variables (name date amount)
 (goals (Account name date amount)
    (incf (aref table (month (deref date))) (deref amount))
    nil))
```

Figure 11: Calling Prolog from Lisp.

As a benefit from integration with Lisp, a simple mechanism for partitioning data bases of clauses is immediately available through the package facility of COMMON LISP.

4 Exploitation

A nice example of interoperability between Prolog and Lisp, is illustrated in figure 12 by the implementation of the second order predicate `find_all`, which computes the multiset of instances for which a goal is true.

```
(defun find_all/3 (var goal list)
  (let (set)
   (goals
     (call/1 goal)
     (push (deref-copy var) set)
     NIL)
   (get_value (nreverse set) list)))
```

Figure 12: findall

The goal is invoked by means of the built–in predicate `call/1`, with a continuation which always fails, but records in a list each solution to the goal. The value produced in each solution must be copied before it can be added to the list, in order to eliminate any variable binding which might get undone during backtracking: the function `deref-copy` is used for this purpose. This example is just one of the many situations where a Prolog programmer would have to resort to the less efficient assert/retract mechanism to save global information.

We ported to our system the Prolog technology theorem prover (PTTP) by Mark Stickel [Sti88]. This is an extension of Prolog that is complete for the first order predicate calculus. It differs from Prolog in its use of unification with the occurs check for soundness, depth–first iterative–deepening search instead of unbounded depth–first search to make the search strategy complete, and the model elimination rule that is added to Prolog inferences to make the inference system complete. Stickel has developed two implementations of PTTP, one in Lisp and one in Prolog. The Lisp implementation is 2–3 times faster. The Lisp implementation uses its own implementation of logic variables and unification, while the Prolog implementation uses the built–in mechanisms of Prolog. Stickel analyzes [Sti90] the difference in performance and suggests some extensions of Prolog that would enable higher performance. However, some of the suggestions, like the introduction of global variables for Prolog, may require substantial redesign in a standard Prolog implementation, while they are readily available in our approach.

We achieved higher performance within our implementation since logic variables and unification are supported efficiently by the WAM primitives and we exploit the same shortcuts used by the Lisp implementation of PTTP [Att91b].

The CRS approach may be profitably used also in the development of new languages which integrate several programming paradigms [Ait88, Bel86].

5 Related Work

There are many works in the literature address the issue of integration between logic programming and functional programming. The goal of this work is usually the development of a new language which combines the features of both approaches and therefore achieves greater expressive power.

Several authors have explored implementations of Prolog in a functional language or additions of logic programming to a functional language through the use of continuations [Hay87, Sri85], where “backtrack” continuations are either explicit in the user code, or hidden through macros.

LogScheme [Ruf90] is an extension to Scheme which adds unification and nondeterminism to Scheme. Besides the normal continuations of Scheme, LogScheme has first–class failure continuations, which allow the user to build his own nondeterministic abstractions.

In [Hay87] a logic variable is a reference either to value, or to another logic variable or to unbound. A logic variable can only be used in the restricted scope of the special construct which creates it, and it must be always explicitly dereferenced.
In LogScheme logic variables are a new data type and their bindings are kept in logic environments, separate from the normal variable environment. They can be used only within gamma–closures, a variation of standard closures, where they are automatically dereferenced.
For the great part these solutions, though of theoretical interest, have lower practical impact. In fact, it is quite hard to develop efficient implementations of a combined logic/functional language (most papers leave unexplored the issue of efficiency [Ruf90] or admit its difficulty), since the conflicting requirements of the two approaches (in terms of data mobility, memory management, control flow) invalidate the techniques that had been developed separately for each approach.
A second reason is that any new language introduces a problem of backward compatibility: i.e. programs written previously will have to be rewritten or translated into the new language. The rate of acceptance of new languages is becoming lower and lower as the amount of code, libraries and programming tools for the language that one expects to have available from the start is becoming significantly large.
Our approach is quite different, since we are providing a way by which each language can coexist with any other. Applications libraries written in a language can be accessed readily from other languages. Predeveloped Prolog or Lisp programs can be embedded into full applications without any change.
In our solution, backtracking is dealt through failure continuations, with a mechanism that allows to detect dynamically determinate predicates and to reclaim their stack frame immediately after return. Logic variables are implemented through locatives, and are first–class values which can be stored in variables or data structures. There is only one environment for variables, not just for Prolog, but for C and Lisp as well.
[Wei88] presents the techniques used in implementing Prolog through cross–compilation to C. Backtracking is dealt by passing success continuations created and invoked through primitives coded in machine language. An attempt is made to mitigate the inefficiencies of success continuations, by avoiding their use in the case of predicates, which the user explicitly declares as determinate.
[Mel85] uses a virtual machine in which different languages can be translated. Backtracking for Prolog is dealt with success continuations and by means of a trail stack and a stack for continuations.

6 Conclusions

We have presented an approach which allows tight interoperability of programming languages. We have explored in particular the issue in the case of Prolog and Lisp, putting particular care into the efficiency of the solution. A Common Runtime for the languages is an appealing solution, since it leaves programmers the freedom to use the most adequate language for each task, and also avoids giving some particular language a special role. The CRS provides a few general facilities, and additional facilities are added for each language. In our case a Lisp Abstract Machine and a subset of the Warren Abstract Machine have been developed. The last component consists of support for logic variables through a locative data type and unification primitives.
The current implementation of Prolog for the CRS uses standard Edinburgh syntax, provided by an extensible operator precedence parser written in Lisp, and includes a comprehensive set of built–in predicates, compatible with those of Quintus[4]. Its performance, measured through the usual Prolog

4 Quintus is a trademark of Quintus Computer Systems, Inc.

benchmarks, compares quite well with native Prolog implementations, demonstrating the viability of the approach.
The work described is embedded in DCL (Delphi COMMON LISP) [DCL89], a COMMON LISP implementation which includes X window graphics, object–oriented programming (CLOS) and facilities for concurrent thread execution. The graphics and object–oriented programming facilities of DCL, are therefore accessible from Prolog.

7 Acknowledgments

We would like to thank M. Simi, B. Giannetti and F. Saracco for fruitful discussions and comments and F. Passaro for help in the implementation.

Bibliography

[Ait88] A. Ait–Kaci, R. Nasr, Integrating logic and functional programming, Lisp and Symbolic Computation, 2, 1988, 51–89.

[Atk89] R. Atkinson, et al., Experiences creating a portable Cedar, Proceedings *of the SIGPLAN 89 Conference on Programming Language Design and Implementation*, 1989.

[Att87] G. Attardi, S. Diomedi, Multithread Common Lisp, Technical Report MADS TR–87.1, DELPHI, 1987.

[Att90] G. Attardi, M. Gaspari, F. Saracco, Interoperability of AI lanaguages, *Proceedings of 9th European Conference on Artificial Intelligence*, 1990, 41–46.

[Att91a] G. Attardi, M. Gaspari, Multilanguage Interoperability, in proceedings of *PLILP'91, 3rd International Symposium on Programming Language Implementation and Logic Programming*, Springer-Verlag, LNCS, n. 528, 1991.

[Att91b] G. Attardi, M. Gaspari, P. Iglio, Efficient Compilation of First Order Predicates, In Trends in Artificial Intelligence, E. Ardizzone, S. Gaglio, F. Sorbello, (Eds.), Springer-Verlag, LNAI, n. 549, 1991.

[Bar88] J.F. Bartlett, Compacting garbage collection with ambiguous roots, DEC Western Researcg Lab Research report 88/2, Feb. 1988.

[Bob88] D. G. Bobrow, et al., Common Lisp Object System Specification, X3J13 standards committee document 88–003, ACM SIGPLAN Notices, **24**(6), 1988.

[Bel86] M. Bellia, G. Levi, The relation between logic and functional languages: a survey, *Journal of Logic Programming*, 3, 1986, 217–236.

[Coh85] J. Cohen, Describing Prolog by its interpretation and compilation, *Commun. ACM*, **2**8(12), 1985, 1311–1324.

[DCL89] Delphi SpA, Delphi Common Lisp, User Reference Manual, Release 1.9, June 1989.

[Hay87] C.T. Haynes, Logic continuations, *The Journal of Logic Programming*, **4**(2), 1987, 157–176.

[Kah86] K. M. Kahn, M. Carlsson, How to implement Prolog on a Lisp Machine, in J. Campbell (Ed.), *Implementations of Prolog*, Wiley, 1986.

[Mel85] C.S. Mellish, Some global optimizations for a Prolog compiler, *The Journal of Logic Programming*, **1**(1), 1985, 43–66.

[Mel86] C.S. Mellish, S. Hardy, Integrating Prolog in the POPLOG environment, in J. Campbell (Ed.), *Implementations of Prolog*, Wiley, 1986.

[Rob82] J.A. Robinson, E.E. Sibert, LOGLISP: motivation, design, implementation, Int. Report, School of Computer and Information Science, Syracuse University, 1982.

[Ruf90] E. Ruf, D. Weise, LogScheme: integrating logic programming into Scheme, *Lisp and Symbolic Computation*, **3**, 1990, 245–288.

[Sri85] A. Srivastava, D. Oxley, An(other) integration of logic and functional programming, in *Proceedings of the 1985 IEEE Symposium on Logic Programming*, IEEE Press, 1985.

[Ste84] G.L. Steele Jr., COMMON LISP: the language, Digital Press, 1984.

[Sti88] M.E. Stickel, A Prolog technology theorem prover: implementation by an extended Prolog compiler, *Journal of Automated Reasoning*, **4**(4), 1988, 353–380.

[Sti90] M.E. Stickel, A Prolog technology theorem prover: a new exposition and implementation in Prolog, in Design and Implementation of Symbolic Computation Systems, *Lecture Notes in Computer Science*, **429**, 1990, 154–163.

[War83] D.H.D. Warren, An abstract Prolog instruction set, Tech. Note 309, SRI International, Menlo Park, Oct. 1983.

[Wei88] J.L. Weiner, S. Ramakrishnan, A piggy–back compiler for Prolog, *Proceedings of SIGPLAN 88 Conference on Programming Language Design and Implementation*, 1988.