The Embeddable Common Lisp

Giuseppe Attardi
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, I-56125 Pisa, Italy
net: attardi@di.unipi.it

Abstract

The Embeddable Common Lisp is an implementation of Common Lisp designed for being embeddable within C based applications.

ECL uses standard C calling conventions for Lisp compiled functions, which allows C programs to easily call Lisp functions and viceversa. No foreign function interface is required: data can be exchanged between C and Lisp with no need for conversion.

ECL is based on a Common Runtime Support (CRS) which provides basic facilities for memory management, dynamic loading and dumping of binary images, support for multiple threads of execution. The CRS is built into a library that can be linked with the code of the application. ECL is modular: main modules are the program development tools (top level, debugger, trace, stepper), the compiler, and CLOS. A native implementation of CLOS is available in ECL: one can configure ECL with or without CLOS. A runtime version of ECL can be built with just the modules which are required by the application.

1 Introduction

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 [4]:

- 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 [Mellish 86], 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 interoperability 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.

We followed this approach to interoperability by building an intermediate support layer between the operating system and the high level programming language, called CRS (Common Runtime Support).

By means of the CRS we have built an Embeddable Common Lisp, a full Common Lisp implementation designed for being embeddable in C applications.

The goals of the ECL design can be summarized as follows:

- Lisp implementation with a small kernel and modular components
- Common Runtime Support provided through a C library
- applications may link CRS and required Lisp modules
- C routines can call Lisp and viceversa
- standard tools can be used for program development (e.g. `dbx` and `make`)

In the next section we describe the CRS and than we survey the critical design solutions for interoperability.

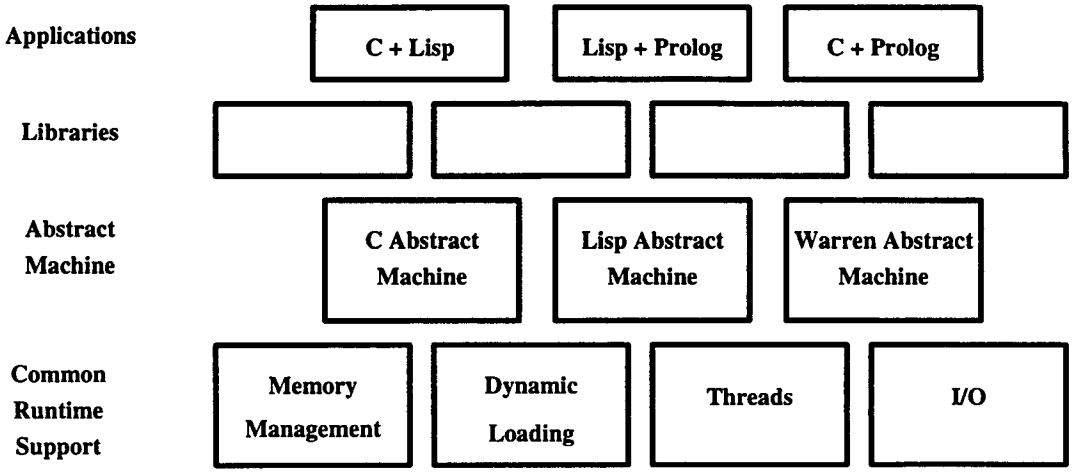
2 Common Runtime Support

The CRS 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 [4], Modula3 [13] and C at DEC SRC, Linda and also to Prolog [20].

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 [6] garbage collector
- symbol table management, including dynamic linking and loading, and image dumping
- multiple threads [5], to support concurrent programming constructs
- generic low level I/O, operating both on files and on network streams
- support for logic variables and unification

The last facility provides support for logic programming languages.
The following diagram shows the CRS in relation with other elements of a programming environment:



An important facility that we plan to add in the future is support for debugging, allowing to debug mixed code programs through a single debugging tool.

3 Solutions for the Runtime

3.1 Memory Management

Memory management is the most critical aspect for enabling *coexistence* of programs in different languages. Code and objects built in ECL can be exchanged with traditional code and libraries. No restrictions should exist on whether a Lisp object can point to a non Lisp object and viceversa. We wanted to be able to pass Lisp objects to programs unaware of garbage collection, allowing them to store such objects in data structures, without special burden on the programmer or risk that the object would be garbage collected. Alternative solutions require the programmer to put an object in an "escape list" before passing it to an external procedure.

Our solution is based on the technique of *conservative* garbage collection [6]. The collector assumes that anything that *might* be a pointer actually *is* a pointer. A random value is assumed to be a pointer by the collector if it corresponds to an address inside the current heap range: any such value is called an *ambiguous pointer*. The CRS incorporates a conservative collector which scans the C stack looking for ambiguous roots.

We have also developed memory management system which is more general as well as customisable. The Customisable Memory Manager [3] has been developed for C++ and allows user to specialise the collector strategies to the need of particular algorithms. We plan to incorporate the CMM in future releases of the CRS.

3.2 Data Model

Lisp data structures are implemented as C data structures, therefore any C program can manipulate them: it needs just to include the file `ecl.h`. Of course Lisp objects can be more abstractly manipulated through a higher level functional interface of C procedures which implement primitive operations on them.

C strings are directly used within certain Lisp objects and this also is quite useful to avoid the need for conversion when crossing language boundaries.

ECoLisp uses a *direct wrapped* [10] representation for certain data types, including `FIXNUM`, `CHARACTER` and `LOCATIVE`, by using the lower 2 bits as a tag. This representation avoids memory allocation and costs

only 2 cycles on arithmetic operations on integers. Other data types are represented as object pointers. The compiler however exploits type information to *unwrap* data whenever possible to avoid generic arithmetic.

3.3 Execution Model

The execution model of ECL is based on the use of the normal C stack for parameter passing and function invocation. This solution has several advantages: the calling convention from C to Lisp is simple, optimized machine instructions can be used for function dispatch, no extra registers must be reserved for maintaining an extra stack, standard tools can be used to examine the stack during debugging.

In more details, the following conventions are adopted:

parameter passing each Lisp function receives an extra argument which represents the count of actual argument in the call:

```
fun(narg, arg1, ..., argn)
```

values return function values are returned on the array `Values`, the result of the function is their count. The macro `VALUES` is actually used in the code since there is a separate `Values` array for each thread.

lexical/closure environments lexical environments are created as arrays, one for each successive level of lexical nesting. Closure environment are represented as lists. So the most general form of lexical function is the following:

```
lexfun(lex0, ..., lexk, narg, env0, arg1, ..., argn)
```

The elements in a closure environment are dereferenced after entering the closure, so that cost for accessing a lexical variables is just one extra indirection.

The choice of these conventions has been enabled by the use of a conservative garbage collector: arguments on the C stack are examined by the collector, lexical environments are also local arrays on the C stack.

For example the code generated for the function:

```
(defun foo (x y)
   (flet ((gen (v) (h y #'(lambda () (h x v)))))
      (list x (gen y))))
```

is the following:

```
/*      function definition for FOO                                    */
static L1(int narg, object V1, object V2)
{ object T0; object lex0[1]; object env0, *CLV0;
        env0 = Cnil;
        CLV0=&CAR(env0=CONS(V1,env0));          /*  X                 */
        lex0[0]=V2;                             /*  Y                 */
        LC3(lex0,1,env0,lex0[0]);               /*  GEN               */
        VALUES(0) = list(2,*CLV0,VALUES(0));
        RETURN(1);
}

/*      closure GEN                                                    */
static LC3(object *lex0,int narg, object env0, object V1)
{ object *CLV1;
        narg--;
        CLV1=&CAR(env0=CONS(V1,env0));          /*  V                 */
```

```
        VALUES(0) = make_cclosure(LC2,env0,&Cblock);
        RETURN((*LK0)(2,lex0[0],VALUES(0)));      /*  H              */
}

/*      closure within GEN                                            */
static LC2(int narg, object env0)
{ object *CLV0, CLV1;
   ...
}
```

ECL is written in ANSI C and generates ANSI C. We exploit though one non standard feature of GCC: variable size arrays. They are used whenever the interpreter needs to temporary allocate a number of objects, for instance during variable binding or argument evaluation. In compiled code they are used for temporary saving multiple values in `multiple-value-call` or `multiple-value-prog1`. On installations where GCC is not available, `alloca` is substituted with some increase of stack occupation.

3.4 Alternative Techniques

Alternative techniques have been considered: for instance one could use the convention of passing arguments to Lisp functions in a single C array. This solution, used in the Scheme to C compiler, is appropriate when interoperability is not a concern and all invocations are generated by the compiler. If we expect programmers to write calls to Lisp functions, having to allocate arrays for the arguments of each call does not seem too appealing. The problem could be alleviated by providing an interface function for each Lisp function, for instance:

```
static inline Lcons(int narg, object car, object cdr) {
  object args[2];
  args[0] = car; args[1] = cdr;
  return LIcons(2, args);
}

LIcons(int narg, object *args) { ... }
```

Programmers could call `Lcons` using ordinary notation and a good C compiler would take care of inlining `Lcons` avoiding extra overhead. This solution has still some drawbacks: `Lcons` would have to be placed in header files, and the optimization is only done by some compilers and when explicitly enabled.

An alternative for returning multiple values would be to use functions returning a structure containing the values. This however would require the caller to pass the number of expected values to the callee, again deviating from most common practice.

For the implementation of lexical functions one could have used the mechanism of lexical procedures available in GNU GCC. This solution keeps pointers to outer frames in registers, but much the same effect is achieved with our solution for most RISC architectures where parameters are passed on registers.

GCC also provides lexical closures, which however work only for dynamic extent closures. The mechanism requires runtime code generation and it is not well supported on all implementation of GCC.

Since the benefits were marginal, in the end we decided not to use such non standard feature of GCC.

3.5 Building Function Calls

The implementation of functions `eval`, `apply` and `funcall` requires the ability to construct a function call with a variable number of arguments.

This is a critical operation because it appears very often in interpreted as well compiled code.

While the standard C library provides a mechanism (<stdarg.h>) for accessing the arguments of a function with variable number of arguments, C does not provide any direct way to build a call with variable number of arguments.

Our solution is based on the use of the variable size arrays of GCC and an assembler macro to perform the actual call.

For example, the code for `eval` which needs to evaluate `n` arguments and then call function `fun` with those values, is implemented as follows:

```
CSTACK[n]
CPUSH(arg_1)
...
CPUSH(arg_n)
CCALL(fun, n)
```

The macro `CSTACK[n]` allocates a variable size array of size `n`. Each call to `CPUSH` pushes a value onto this array and finally `CCALL(fun, n)` issues an assembler branch instruction to procedure `fun`. To ensure portability, a less efficient C version of `CCALL` is available as default.

3.6 Dynamic Loading

Dynamic loading is implemented as a function which takes the name of a binary file as parameter:

```
dld(char *faslfile, struct codeblock *Cblock)
```

`dld` is capable of interpreting various binary file formats (a.out, coff, ecoff). It identifies the sections of text and data which must be loaded into memory, transfers them into memory and then performs the tasks of loader: relocation and linking of external symbols. The start address of the block of memory allocated for the code and its size are returned in the structure passed as the second argument.

Dynamic linking is used to load compiled Lisp code as well as any other binary file into a running application.

The inverse operation, of dumping an executable image of an application is provided by the `unexec` function, which takes as arguments the name of the file where to save the image and the name of the original file containing symbols and relocation information.

```
unexec(char *save_file, char *original_file, ...)
```

Some of the versions of `unexec` used in ECL are derived from those supplied with GNU Emacs.

4 CLOS

A native implementation of the Common Lisp Object System is provided with ECL. This consists in specific data types for instances and generic function dispatchers. Generic function invocation is performed through an efficient method lookup which computes a hash code based on the types of the arguments to the function. This code is used to access a method-specific hash table which cashes the effective method for each series of specializers. The full protocol for computing the effective method is only invoked the first time a method is called for a particular combination of argument types.

Moreover, the generic function dispatcher is cashed in each place in the code where a function is called via a reference to its symbol. This allows the code to jump directly to the dispatch code for a generic function without the need to go through the function cell of the symbol and testing the type of function to which it is bound.

Method accessors for instances derived from class `STANDARD-CLASS` are optimized and turned into indexed indirect memory references. At entry in a method, index values are retrieved for all slots accessed within

the method. Thereafter accesses to the slots use such index into the vector of slots of the instance. The indirection is required to support the class redefinition protocol of CLOS.

The bootstrap of CLOS is done at the C level of the Lisp kernel by creating three basic classes: `T`, `CLASS`, `OBJECT`, related as described in [2].

Common Lisp structures are implemented as instances of classes derived from class `STRUCTURE-CLASS` and therefore are completely integrated with other classes: for instance a structure class can be specialised to a subclass by inheritance.

5 Threads

Multiple threads of executions are possible within a single ECL process. Thread scheduling is arranged by a preemptive scheduler. The thread mechanism is implemented by means of the Unix software interrupt handlers and the `setjmp/longjmp` primitives. The interrrupt handler invokes the scheduler to switch control among threads at the expiration of each time slice.

At the Lisp level threads provide a model of execution based on the notion of `continuation`. In this model control flow proceeds normally through function calls, unless the program requests access to its continuation. It can then decide whether it wants to resume such continuation at the end, thereby proceeding normally, or to resume a different continuation, thereby transferring control to another thread, or to suspend until its continuation is resumed by another thread. Continuations are first-class objects which can be passed as arguments or returned as values from functions.

Given this model of computation, the only addition which is necessary to be able to handle concurrency is the ability to create multiple threads of execution. Control of flow among different threads is obtained just by the use of continuations.

The main Lisp functions available for using threads are: `make-thread`, and `resume`. `resume` is called with a continuation `cont` and several values: its effect is to resume execution of `cont` while returning those values to the expression where such continuation was created.

A continuation is created with the construct `(let/cc cont body)` [14] which binds variable `cont` to a newly created continuation within the thread where it is executed, executes the body and then suspends the thread until the continuation is resumed. Here is a simple example of the use of these constructs: the `consumer` function starts a producer thread and then waits to be resumed until something has been produced; the `producer` function executes in a separate thread and resumes the consumer thread after each run of production, suspending itself in turn.

```
(defun consumer ()
   (let ((producer (make-thread #'producer)))
     (loop
       (setq producer (let/cc me (resume producer me)))
       (consume))))

(defun producer (consumer)
   (loop
     (produce)
     (setq consumer (let/cc me (resume consumer me)))))
```

In this example only one thread is executing at any one time, but in general several threads can execute in pseudo concurrency.

6 Compiler

The ECL compiler is derived with extensive rewriting from the KCL compiler.

Here is an example of actual code generated by the ECL compiler for the classical factorial function:

```
/*      function definition for FACTORIAL                              */
static L1(int narg, object V1)
{
        if(!(number_compare(MAKE_FIXNUM(0),(V1))==0)){goto L2;}
        VALUES(0) = MAKE_FIXNUM(1);
        RETURN(1);
L2:
        L1(1,one_minus((V1)))                    /*  FACTORIAL       */;
        VALUES(0) = number_times((V1),VALUES(0));
        RETURN(1);
}
```

The compiler can exploit type declarations to produce better code which avoids generic arithmetic operations, whenever possible.

Here is an example from a real application:

```
(defun logprob (p q)
  (declare (type double-float p q))
  (if (zerop p)
      0.0
      (* p (log q))))
```

which is turned into:

```
/*      function definition for LOGPROB                                */
static L1(int narg, object V1, object V2)
{
        {double V3;
        V3= lf(V1);
        if(!((V3)==0)){goto L2;}
        VALUES(0) = VV[0];
        RETURN(1);
L2:
        VALUES(0) = make_longfloat((double)(V3)*
                                   (double)(log((double)(lf((V2))))));
        RETURN(1);
        }
}
```

The macro `lf` unwraps a pointer to a long float. In order to produce this code, the compiler exploits a simple type inference mechanism which propagates information both bottom-up from the arguments to functions as well as top-down from the places where function values are used.

6.1 Optimizations

A number of optimization which cannot be done at the source level are performed on the intermediate form produced by the first pass of the compiler. For example, replacing a `let` variable used just once in the body with its corresponding expression, functions with identical code share the same C code.

The compiler also arranges to allocate frequently accessed variables in registers.

6.2 C Code Optimization

Fine tuning of critical functions can be achieved through the mechanism of inline C optimization. One can specify directly the C code to be used to perform a certain function, as in the following example, taken from the ECL CLX implementation:

```
(definline aref-card29 (string fixnum) fixnum
  "((*(unsigned long *)((#0)->ust.ust_self+(#1))) & 0x1fffffff)")
```

Any call to function `aref-card29` with arguments of type `string` and `fixnum`, is compiled into the C code supplied, where `#0` and `#1` represent the two actual arguments.

7 Modularization

The original Common Lisp design did not pay much attention to modularization. More recent work on languages like Eulisp [14] stresses the importance of modularization in the language, introducing the separation between the kernel, the libraries and the environment. Many people now recognise the importance of a language made out of modular components. It is fairly clear that the requirements for a programming and developement environment are quite different from those of a delivered application. So it should be possible to build the application without being encumbered by parts which are not needed. The approach called "tree shaking" has been proposed to solve this problem, i.e. using an automated tool which analyses the program to decide which functions are actually used in it in order to discard them from the final image. We consider more practical the approach which is traditional in most languages: i.e. leaving this task to the linker with some indication from the user. The user indicates which libraries his application requires and the linker takes care of incorporating what is actually needed in the built image. Tools like `autoconf` and `make` at the operating system level and `defsystem` at the Lisp level are typically used.

Another issue related to modularisation is the attempt to design a subset of Common Lisp which could be suitable as target for compilation. The APPLY project [15] in Germany has produced the specification for CL0, a subset of Common Lisp, which is the source language for their Lisp to C compiler CLiCC. CL0 puts severe restrictions on many Common Lisp constructs, in order to ensure that the final code produced can run as a standalone application.

This approach is therefore different from ours since we do not impose any such limitations to Lisp code, given that our runtime support allows full Common Lisp compatibility. The CRS is small enough (700K) that the increase in size of the overall application is acceptable for nowaday technology. The APPLY work is however relevant since it indicates a way to split Common Lisp into independent components.

The current version of ECL consists of the following separate modules which can be selected through options to the `configure` shell script:

- basic runtime (470K)
- lisp libraries (280K)
- compiler (380K)
- CLOS (226K)
- development environment (160K)
- CLX (400K)
- threads (50K)
- unification (10K)

8 Logic Programming Support

The CRS provides support for logic variables and unification by implementing the `get` and `unify` primitives of the Warren Abstract Machine (WAM) [19]. Further details on the CRS support for logic programming are presented in [1].

8.1 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 in the heap 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 [12] and is a pointer to a single memory cell, which can be either a slot within a structure or the cell for another variable.

Locatives are implemented as immediate data, using 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. Lisp `fixnums` and `characters` are also represented as immediate data.

9 Performance

The performance of ECL is quite satisfactory. Here is a sample of comparisons with the most well known public domain Lisp implementations:

Bench	**ECL** (0.4)	**AKCL** (1.615)	**CMU** (16f)	**CLISP** (5.5)
BOYER	2.250	2.233	4.100	22.600
BROWSE	3.233	4.167	9.130	15.510
FFT	0.200	71.333	0.410	9.900
FPRINT	0.183	0.150	0.990	0.260
FRPOLY	21.083	175.733	11.010	37.915
TRIANG	10.050	13.133	32.210	198.220

Times are in seconds on a SparcStation ELC diskless with 16 MB memory. The comparison with AKCL is particularly interesting, since the use of the C stack is the most considerable difference in the two implementations. The data show that ECL is 30-40% faster than AKCL, and this seems a good justification for the design choices of ECL. A few cases were ECL performs significantly better are due to the effect of type inference in the compiler. Moreover, also the size of compiled code is smaller, from 30% to 50%: code sharing produces even more significant effects on files where a lot of auxiliary functions are generated by macros (as in CLX).

10 Related Work

KCL [11] has been the first Common Lisp implementation to adopt C as its intermediate compilation language. However C was used more as an implementation language rather than providing integration

between the two languages: in particular Lisp uses a separate stack for parameters and values, garbage collection only applies to Lisp objects, basic data types like strings have different representations. Commercial versions of KCL were developed by Ibuki and by DELPHI.

In AKCL, a version of KCL developed by William F. Schelter, several improvements to KCL were introduced, including a conservative garbage collector and a dynamic loader. The latest version of AKCL is distributed as GNU Common Lisp (GCL).

In all these KCL derivatives one can incorporate C code in a program either by including it directly within Lisp files or by dynamically linking and loading a binary file. However special conventions have to be followed in such C code for parameter passing and also to ensure that live Lisp objects are not prematurely garbage collected. A special form `defentry`, recognised by the compiler, was used to create an interface function between a Lisp function and a C function. Invocation of Lisp functions from C was feasible but even more cumbersome.

ECL instead addresses the issue of allowing a C based application to call Lisp programs, thereby reverting the approach of its predecessors.

Chestnut Software Inc. produces a Lisp-to-C translator [9] which is attempting to be as close as possible to Common Lisp, and has a Run-Time Library which appears analogous to the CRS. In terms of size, the entire library - including all Common Lisp functionality, the garbage collector, CLOS, and runtime support for all Chestnut extensions, occupies approx 1 MB of memory in a translated application.

11 Conclusions

Common Lisp is not an easy language to integrate with others. For instance, if Common Lisp had no multiple values, like Scheme or Eulisp, it would have been straightforward to compile Lisp functions into C functions returning the value. We have shown nevertheless that interoperability between Common Lisp and C is feasible and effective. Our embeddable Lisp implementation is useful for those who want to build Lisp packages to be incorporated into other non Lisp applications. But we hope also that our experience might be useful in designing future evolutions of Lisp which take more seriously interoperability as a concern.

ECL is available for anonymous ftp from site `ftp.icsi.berkeley.edu` in the directory `/pub/ai/ecl`. Please address comments, suggestions, bug reports to `ecl@di.unipi.it`.

Several ideas and techniques used in ECL were inspired or derived from solutions developed by William Schelter for AKCL. Mauro Gaspari and Tito Flagella participated in various stages of the development. Andreas Stolcke supplied significant test programs and helped in the debugging.

References

[1] G. Attardi, M. Gaspari and F. Saracco "Interoperability of AI languages", Proceedings of 9th European Conference on Artificial Intelligence, Stockholm, 1990, 41-46.

[2] G. Attardi "Metalevel Programming in CLOS", in *Object Oriented Programming: the CLOS perspective*, A. Paepke (Editor), MIT Press, 1992.

[3] G. Attardi and T. Flagella "A customisable memory management framework", Proceedings of USENIX C++ Conference 1994, Cambridge, Massachusetts, April 1994.

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

[5] G. Attardi and S. Diomedi "Multithread Common Lisp", Technical Report MADS TR-87.1, DELPHI, 1987.

[6] J.F. Bartlett "Compacting garbage collection with ambiguous roots", DEC Western Research Lab Research report 88/2, February 1988.

[7] J. F. Bartlett "Scheme-¿c a portable scheme-to-c compiler", Research Report 89 1, DEC Western Research Laboratory, Palo Alto (CA), January 1989.

[8] D. G. Bobrow, et al. "Common Lisp Object System Specification", *ACM SIGPLAN Notices*, **24**(6), 1988.

[9] Chestnut Software Inc., "Lisp-to-C Translator", Technical Specification Release 3.0, 1991.

[10] D. Gudeman, "Representing Type Information in Dynamically Typed Languages", TR 93-27, Department of Computer Science, University of Arizona, October 1993.

[11] M. Hagiya and T. Yuasa "Kyoto Common Lisp Report", RIMS, Kyoto University, 1985.

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

[13] G. Nelson, editor "Systems Programming with Modula3", Prentice Hall, 1991.

[14] J. Padget and G. Nguyen (Eds.) "Eulisp Version 0.9", December 1993.

[15] W. Goerigk, U. Hoffmann, H. Knutzen "Common Lisp to C Compiler", Christian-Albrechts-Universität zu Kiel, 1993.

[16] R. M. Stallman "GNU GCC Version 2.5.8", Free Software Foundation, Cambridge (MA), 1993.

[17] G. L. Steele, Jr "Common Lisp, the Language", Digital Press, Burlington (MA), 1984.

[18] G. L. Steele, Jr "Common Lisp, the Language", Digital Press, Burlington (MA), 2nd edition, 1990.

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

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