ISTITUTO DI ELABORAZIONE DELL'INFORMAZIONE - C.N.R. - P I S A

MAGMA-LISP: A "MACHINE LANGUAGE" FOR ARTIFICIAL INTELLIGENCE

Carlo Montangero[+]
Giuliano Pacini[+]
Franco Turini[+]

Nota Interna

B75-3

[+]Lavoro svolto nell'ambito della collaborazione con l'Istituto di Scienza dell'Informazione dell'Università di Pisa

Stampato in proprio

Abstract

The paper describes MAGMA-Lisp, an extended Lisp system proposed as an implementation tool for A.I. languages exploiting nondeterministic techniques. The main idea informing MAGMA-Lisp is that a tree structure of conceptually indipendent computation environments (context tree) is the supporting structure of any nondeterministic system. MAGMA-Lisp proposes this structure in a quite virginal form, so that the user can state his own techniques to prune, select and explore the available alternatives. In this sense, MAGMA-Lisp is to be viewed as a "machine language".

The ideas of MAGMA-Lisp are contrasted with the systems that most influenced its design. The technique used in context implementation is described, showing how a very flexible context mechanism can be realized with a tolerable loss of efficiency. In particular, in spite of the complexity of the system, garbage collection does not result much more time consuming than in standard Lisp systems.

1. Introduction

Many original features, embedded in the languages developed for A.I. so far, involve nondeterminism <3,4,6,12>. In fact, features like "pattern directed procedure invocation" and "associative data-base retrieval" give rise to situations in which many possibilities are open for the computation to proceed. Such a feature is typical of nondeterministic procedures, which are characterized by the presence of choice points, where the subsequent actions are not univocally determined.

The nondeterministic behaviour of A.I. systems has been first realized by automatic backtracking. Different alternatives are attempted one by one. Whenever one fails, the state of the computation is automatically restored as it was before the last choice was done, and a new one is attempted. As a matter of terminology, we say that each alternative is attempted in a newly created context, while the context in which the choice point has been encountered is "frozen" to maintain the state of the computation as when the attempt begins.

Automatic backtracking, although superficially attractive, turns out to be not adequate to face the main problem with A.I. programming: that is, the ability to prune the open alternatives, hopefully by exploiting informations gathered during the execution of the program itself. In fact, pure backtracking forbids the programmer's intervention in the choice policy, limiting the system to an exhaustive search of all the possibilities and leading to inefficient computations. A few features, like failure

messages, have been proposed since PLANNER design <7>, to override this handicap. Anyway, a really satisfactory solution seems to reside in the design of languages in which the programmer can state his own rules to prune, select and explore the alternatives. From this viewpoint, the ability of transferring informations from the context of an alternative to another one turns out to be particularly important. It is this ability, indeed, that allows to state rules not based upon a-priori knowledge only, but also upon informations gathered at run-time.

Systems allowing the user to define his own choice policy have been already proposed and realized. It is convenient to discuss briefly some characteristics of the systems that most influenced the design of MAGMA-Lisp, in order to contrast it with them.

Bobrow and Wegbreit <2> proposed a very "fluid" control structure that surely allows the user to define his own heuristics and policies. There is no notion of context, in the sense of a built-in ability to save the state of computations. The system provides instead a mechanism to save the continuation point and the anonymous partial results of procedure activations, while the user is in charge of restoring the state of the data base.

CONNIVER <14> provides a built-in mechanism (context frame) to save and restore the data base incrementally, supported by a control structure similar to that of Bobrow and Wegbreit. The generation and the deletion of context frames are associated with procedure invocations and returns. The restoring mechanism appears, therefore, on the same level of the control structure, while, more generally,

it should be able to save and restore the control structure itself.

Finally, both systems appear to lack a mechanism to allow straightforward saving of complete "snapshots" of computations, to be resumed afterwards, if needed. In this sense, they may lead the user towards an attitude of mind that imperfectly matches the nonderministic programming style. Such design choices are mainly motivated by the sake of efficiency. Undoubtedly, this is not a trifling point, but it is our feeling that to much has been sacrificed to efficiency, with respect to the clarity and intuitiveness of the basic concepts of the language. MAGMA-Lisp has been designed in order to realize a good compromise between clarity and intuitiveness on one hand and efficiency on the other one.

To summarize, the main idea of our proposal is the notion of context, to be thought of as a complete environment capable of a "deterministic" computation, inclusive of the control structure. Nondeterminism is attained by exploring different alternatives in different contexts. Generation, deletion, switching and communication among contexts are in charge of the programmer, so that the choice-policy is under user's control. In other words, the user has the neat and intuitive notion of context as a basic programming concept: he can think of having as many indipendent computation environments as he needs to attempt all the alternative paths to his goal. On the other hand, the system enables him to break the "apartheid" among contexts, in order to transfer useful informations from one to another.

2. MAGMA-Lisp: a user view.

MAGMA-Lisp has been designed as an implementation support for the realization of sophisticated programming systems allowing nondeterminism and complex control regimes. This section sketches the main features of MAGMA-Lisp, describing its basic characterizing concepts: contexts and applications.

2.1 The context tree

Intuitively speaking, the simplest viewpoint for the user of a nondeterministic language is the following one: at each choice-point crossing, as many new computation environments are created as the alternatives to be explored are. So, a tree of computation environments (context tree) grows and contracts according to choice-point crossings and failures. The initial state of newly created contexts is identical to the one in which the choice point is encountered.

MAGMA-Lisp considers the context tree to be the structure underlying any nondeterministic system, brings it to the light and commits its control to the user.

A small set of primitives allows adding and dropping nodes to the context tree; this way, computation environments can be created and destroyed. The function

newcxt(cxt)

creates a new computation environment, whose initial state

is that of the context identified by cxt (*). The newly created context is added to the control tree as a son of cxt. newcxt returns the identifier of the created context: context identifiers are regarded as a special data type, allowing the explicit management of contexts.

The size of the context tree can be reduced by the function

contract(cxt1,cxt2)

whose effect is the substitution of the subtree rooted in cxt1 by the subtree rooted in cxt2 (**).

The format of this function reflects the nondeterministic trend of MAGMA-Lisp. In fact, if cxt2 is missing, contract allows to drop contexts associated to failed computations, while, if cxt2 is not missing and it is a leaf, contract allows to draw the final consequence of a successful computation, reducing the whole tree to the single node cxt2. Any intermediate strategy can be programmed as well.

The context tree can be inspected from inside any context, using the functions

son(cxt)

and

getcxt(n,cxt)

that return the list of the context identifiers of the sons

(*) If cxt is missing, the context in which newcxt is executed is assumed.

(**) cxt2 must be a descendant of cxt1; if cxt2 is missing, the whole subtree rooted in cxt1 is dropped.

of cxt and the n-th ancestor of cxt in the context tree, respectively (*).

2.2 Applications and tree control structures

Each node of the context tree represents a computation environment, which is essentially that of a standard LISP system. That is, adding a new node to the context tree results in creating a new computation environment, whose state is initially a copy of its father (**). Computation environments provided by MAGMA-Lisp differ from that of standard LISP systems in what their nature has been extended to allow growing a tree control structure. In other words, inside each context it is possible to depart from the normal last-in-first-out discipline of LISP in order to define more complex control regimes, like coroutines etc.

The basic component of tree control structures is the application, which is to be viewed as a function activation frame. In fact, an application contains:

- -a function definition;
- -a local environment, i.e. bindings and locals;
- -a return pointer, i.e. the identifier of the application to which the value of the function has to be returned;

(*) If cxt is missing, the context in which the function is executed is assumed.

(**) It goes without saying that this is only a user's view. Contexts are simulated as described in sect. 3.

-an environment pointer ,i.e. the pointer to an association list;

-a continuation point in the application itself.

An application A is a son of B in a tree control structure if its return pointer is the identifier of B. Applications are generated either implicitely by calling a function defined as an EXPR, or explicitely by invocation of apply. In the former case, both the return and the environment pointer are set according to the normal LIFO rule. In the latter case, both pointers can be defined in the invocation of apply.

Application identifiers can be obtained by the function

getap (n, appid)

that returns the identifier of the n-th ancestor of appid(*) in the control tree.

2.3 The function apply

MAGMA-Lisp is supposed to be a uniprocessor system, so, at any time, there is only one context in which the computation is proceeding (active context). Switching among different contexts can be obtained by apply. In our system apply is generalized both to manage tree control structures and to execute switching among contexts.

The format of apply is the following:

apply(fn, args, envp, retp, cxt)

It activates a new application to apply fn to args. The new

(*) If appid is missing the identifier of the actual application is assumed.

application is added to the control tree of context cxt as a son of retp, while envp defines the environment pointer. The environment is built up appending bindings and locals to the association list pointed by application envp. If the last argument is missing, the active context is assumed.

Apply is the only means to execute context switching and to depart from the normal LIFO rule inside any single context. Then, it is the basic tool for the user to control the nondetermistic features of the system, and, at the same time, to define procedures that exploit non-standard regimes of control.

2.4 Communications between contexts

Contexts, as they are provided by MAGMA-Lisp, must be thought of as nodes in a tree structure of indipendent environments. The computation can proceed in the leaves as well as in the other nodes. Actions in any context do not affect the state of any other, unless the programmer explicitely states the contraries.

Information can be transferred between contexts via apply: in this case there is a control transfer too. To make information transmission easier, the functions put and get have been extended in MAGMA-Lisp by an additional argument, specifying the context in which properties are to be set or inspected. As a rule, changes performed by put are localized to a single context, according to the general philosophy. In many cases, however, it seems useful that modifications be global, i.e. they propagate from a context to all its descendants. This happens if the additional argument of put is a list (of a single context identifier) instead of a

context identifier.

3. Context implementation

This section describes some techniques used in the implementation of the system. The main problems that have been faced have to do with the tree control structure and the context mechanism. Here special emphasis is given to the solutions adopted to realize the context mechanism.

Before discussing the details, it is worth to outline the general approach used to implement Magma-lisp. The system has been realized in two quite distinct subsequent steps. First, a LISP system extended by a tree control structure was implemented, then the context mechanism has been superimposed on this basic support.

Contexts are introduced in a very simple and general way: informations having a contextual nature are referred to in an indirect way, through multi defined value lists (MVL). MVLs are lists providing all the values in the different contexts, with a technique that somewhat recalls that of QA4 <12>. MVL's organization will be discussed in details, and it is such to guarantee efficient search and bookkeeping. This technique is applied throughout the whole system, control information included. In other words, there is a unique tree control structure in the system. Any element of the tree may belong to more than one context. In such a case, the corresponding memory block may contain a few pointers to MVLs. Typically, the continuation point is a context depending information.

No sophisticated techniques have been studied in order to implement the tree control structure, using instead a

straightforward list organization. Whenever a function is invoked (either implicitely or explicitely via apply) an application block is allocated to store basic information (i.e. the function definition, the return and environment pointers and the continuation point) together with a suitable amount of memory for bindings and locals. Finally, auxiliary memory may be allocated during the computation for intermediate results.

Memory is obtained from areas organized in free lists. Returning from a function application, the corresponding memory is given back to the free lists, in the simplest situations; otherwise, its recovery is deferred to garbage collection.

The technique of retention is quite simple: if an application A has undergone a getap (see sct. 2.2), A and all the applications on the path from A to the root are retained; recovery, if that will be the case, will occur at garbage collection time.

3.1 Summary of MVL techniques

Whenever a piece of information (e.g. the property of an atom) has different values depending on the context, the cell which should point to the value points instead to an MVL.

An MVL is a list of dotted pairs (c.v) where c is a context identifier and v the value of the information in c. When a new context is created, no memory is allocated at all. Memory will be allocated, by growing or generating MVLs, only when updating is actually performed in that context. Moreover, the size of an MVL depends on the number

of contexts in which updating occurred; in other words, it does not depend on the number of existing contexts, i.e. on the size of the context tree.

It is fundamental that MVLs are generated only when they are really needed, both with respect to execution time and memory space. There are two cases: the global data base and the local environments. MVLs are generated to record properties in property-lists (the global data base) when the context tree is not trivial, i.e. there are more than one context. MVLs are generated to record items of information in locals environments, only if the involved application is to be retained even if it is exited. By this simple technique, MVLs are generated almost only when they are really needed.

To find the value corresponding to a context c in a MVL, a pair (c.v) is looked for firstly; if it does not exists, then, accordingly to the definition of context tree, it is enough to look for the pair corresponding to the nearest ancestor of c. Thus, the crucial point with regard to efficiency is to find a MVL organization capable to speed up searching an element or its nearest ancestor actually present in the MVL.

The following subsections describe an organization that allows to search and update MVLs with a satisfactory efficiency.

3.2 The context table

Magma-lisp memorizes the context tree in an array (context table) indexed by context identifiers. Each row stores pointers in order to memorize the context tree as a

binary tree. Moreover, row $\underline{c}$ associates context $\underline{c}$ with a pair of integers, that will be denoted by r($\underline{c}$) and s($\underline{c}$):

- r($\underline{c}$) is the number of nodes preceeding $\underline{c}$ in the preorder traversal <8> of the context tree;
- s($\underline{c}$) is the r-number of the last descendant of $\underline{c}$ in the preorder traversal, i.e. s($\underline{c}$) is the largest r-number among $\underline{c}$'s descendants; s($\underline{c}$) is set to r($\underline{c}$) if $\underline{c}$ has no descendants.

r-numbers and s-numbers are characterized by the following property: given two contexts $\underline{c}'$ and $\underline{c}''$, $\underline{c}'$ is a descendant of $\underline{c}''$ if and only if

$$r(\underline{c}'') < r(\underline{c}') \leq s(\underline{c}'').$$

3.3 Searching MVLs

MVL's components are listed by decreasing values of r-numbers.

The follcwing algorithm searches a MVL for the value in context $\underline{c}$.

Algorithm 1

1) scan the MVL until a pair ($\underline{c}'$.v') is found, such that

$$r(\underline{c}') \leq r(\underline{c})$$

2) restarting from ($\underline{c}'$.v'), scan the MVL until a pair ($\underline{c}''$.v") such that

$$s(\underline{c}'') \geq r(\underline{c})$$

is found.

This way, $\underline{c}''$ is either $\underline{c}$ or the ancestor of $\underline{c}$ with the greatest r-number, i.e. the nearest ancestor of $\underline{c}$, actually present in the MVL, is found.

3.4 Updating MVLs

In updating a MVL with respect to c, care must be paid so that the modification does not propagate to any descendant of c, or does propagate to all its descendants, according to the assigning modality (see sct. 2.4). In the first case, for each son of c not already in the MVL a pair must be added to preserve the old value in context c. In the second case all the descendants of c must be eliminated from the MVL. In both cases the operation can be performed in a single scanning of the MVL. This is obvious in the second case. In the first one the operation can be performed by algorithm 2, that needs the list of sons of c ordered by decreasing r-numbers. Such a list can be drawn from the context table directly.

Algorithm 2

1) while there are sons of c: search (algorithm 1) the pair corresponding to the son of c with the greatest r-number; if there is no such a pair, insert it and push the pointer to it on a stack, say S;
2) when all the sons have been considered, look for the value in c (algorithm 1), store it in all the pairs pointed by stack S, finally update the value in c (inserting a pair, if missing).

Whenever algorithm 1 is needed, it can be applied restarting from the last considered pair, because of the ordering of the MVL.

Assuming the number of accesses to the context table as a measure of the complexity of search and updating

algorithms, it follows from the previous discussions that the complexity has an upper bound which is linear with the length of the MVL, whereas it does not depend on the size of the context tree. This is one of the most interesting advantages offered by MAGMA-lisp organization. Another advantage, estimable in a system designed to allow sophisticated explorations of goal trees, is that switching among contexts is practically gratis, consisting in changing the active context indicator.

A small overhead is imposed in creating new contexts, since the context table must be updated. We note, however, that this task can be accomplished by a single scanning of the context table. In fact, the rule to update the context table in order to add a node c' as a son of c is:

1) let rc and sc denote r(c) and s(c) respectively;
2) for each row:
   if r≤rc and s≥sc then increment s by 1;
   if sc<r then increment r and s by 1;
3) finally set r(c') and s(c') to s(c).

3.5 Garbage collection

The function contract deletes contexts only from the user's viewpoint. In reality, contract simply marks the rows of the context table corresponding to deleted contexts, making it possible to detect attempts of further use of references to dropped contexts (illegal references). Rearrangement of the context table is deferred to Garbage Collection time, as well as the rearrangement of MVLs, i.e. the actual elimination of the pairs corresponding to deleted

contexts.

The direct extension of the standard garbage collection philosophy to a system supporting a context mechanism would result in repeated tracing of the whole system. The direct extension, indeed, is the following: first, trace and mark all items reachable starting from the actual "position" in the active context; then, trace the system again and again until all contexts whose identifiers have been found in previous tracings have been considered. Finally go through the system once more to rearrange MVLs.

MAGMA-Lisp garbage collector <13> takes instead advantage of the fact that the management of the control tree is completely committed to the user. The system is traced only once, since the context tree defines explicitely the contexts to be retained.

The main input information of the garbage collector are:

-the list of the contexts to be retained;

-a list of starting points (application identifiers) in the tree control structure.

The first list is drawn from the context table. The second list contains the identifiers of all the applications, which have undergone a getap operation and are still legally referable in one context at least. This list is handled by th system according to the following philosophy, which defines, from the user's viewpoint, the behaviour of the tree control structure inside each context: whenever an application is exited, the subtree rooted in it is dropped from the control tree. This is only a user's view; the system has instead a mechanism to update the list of

starting points in the unique tree control structure actually existing in the system.

Finally we note that the rearrangement of MVLs is not a trivial business. In fact, it is not sufficient to eliminate the pairs corresponding to deleted contexts. In many cases, the pair corresponding to a deleted context c must be retained and updated with regard to the context identifier in it, since it is possible that pairs corresponding to surviving descendants of c are not present in the MVL. The rearranging algorithm exploits the ordering by decreasing r-numbers of MVLs, and the fact that the list of the contexts to be retained is ordered in the same way: MVLs are rearranged in a single scan and the number of accesses to the context table is linear with the sum of the length of the MVL and the number of contexts to be retained.

Conclusions

The main idea informing MAGMA-Lisp is that a tree of conceptually indipendent computation environments is the supporting structure of any programming system allowing nondeterminism. MAGMA-Lisp provides this basic support in a quite virginal form: in this sense it must be considered as a "machine language". The language has neither primitives to set up choice points in programs nor primitives to fail, etc.. There are, instead, primitives that add and delete nodes from the context tree and facilities to switch and transfer informations among environments. So, the user can tune the power of the system to match his own requirements by the definition of a suitable set of functions, expressing his own techniques to explore the available alternatives.

Experiments in this line are presently in progress. A language for nondeterministic programming, ND-Lisp <10>, has been defined. ND-Lisp is much more problem-oriented than MAGMA-Lisp, providing specific ways to set up choice points, primitives to fail and to suspend and resume alternatives. ND-Lisp enjoys a fine structure and still allows complete freedom in the choice-policy definition. This is obtained filtering the wildness of MAGMA-Lisp, that ND-Lisp users are not encouraged to employ directly, through a few functions realizing the primitives of ND-Lisp. Besides, a pattern-matching language (SNARK <9>), which is, more generally, a formalism for the definition of symbolic interpreters, and a proof-checker <1>, which allows a high degree of intervention of the user in the proof, are presently being implemented in MAGMA-Lisp.

In the light of these experiments, MAGMA-Lisp shows itself as a good tool for the implementation of A.I. systems (*). This is mainly due to its nice balancing of clarity and intuitiveness of its basic programming concepts, on one hand, and a reasonable level of efficiency on the other.

(*) At present, MAGMA-Lisp is written in FORTRAN and runs on the IBM 360/67 under CP/CMS.

References

1. Aiello, L., Aiello M., Attardi, G. and G.P. Prini. "A basic frame for proof-checker implementations" Scientific note, Istituto di Scienze dell'Informazione, Pisa (to appear).
2. Bobrow, D.G. and D.V. Wegbreit. "A model and stack implementation of multiple environments". Comm. ACM, vol. 16,n. 10, pp. 591-603,
3. Confield Smith, D. and H.J. Enea. "Backtracking in MLISP2". Proc. 3rd IJCAI, Stanford 1973, pp. 671-695.
4. Davies, D.J.M. "POPLER: a POP2 PLANNER". MIP-89, school of A.I., University of Edinburgh.
5. Floyd, R.W. "Nondeterministic algorithms". J. ACM 14, 4 (October 1967) pp. 636-644.
6. Hewitt, C. "Procedural embedding of knowledge in PLANNER". Proc. IJCAI 2, 1971, pp. 167-182.
7. Hewitt, C. "PLANNER: a language for manipulating models and proving theorems in a robot". A.I. memo 168, MIT 1970.
8. Knuth, D. "The art of computer programming". Vol. 1, Addison Wesley, 1971.
9. Levi, G. and F. Sirovich. "Valutazione simbolica e unificazione". Proc. of "Convegno di Informatica teorica" Mantova (italy), November 74.
10. Montangero, C., Pacini, G. and F. Turini. "Two-level control structure for nondeterministic programming". Internal Note B74-31, IEI Pisa, October 74.
11. Reboh, R. and E. Sacerdoti. "A preliminary QLISP manual". Technical Note 81, SRI Project 8721, A.I. Center, Stanford, August 73.
12. Rulifson, J.F., Waldinger, R.J. and J.A. Derksen. "QA4: procedural calculus for intuitive reasoning". Technical Note 73, A.I. Center, Stanford, November 72.
13. Simi, M. "Un efficiente Garbage Collector per un sistema Lisp non deterministico". Thesis, Istituto di Scienze dell'Informazione, Pisa, November 74.
14. Sussman, G.J. and D.V. McDermott. "From PLANNER to CONNIVER, a genetic approach". Proc. FJCC 41 (November 72) pp. 1171-1179.