REFLECTION AS A TOOL FOR INTEGRATION: AN EXERCISE IN PROCEDURAL INTROSPECTION.

Roberto Ghislanzoni (†), Luca Spampinato (‡), Giorgio Torniclli (†)

(‡) QUINARY,via Scttcmbrini 40,20124 Milano, ITALY, 39-2-222703.
(†) Dipartimcnto di Scienze dclllnformazione, University di Milano, ITALY

ABSTRACT

This paper reports on a quite large experience in implementing a procedurally introspective system (PIS), ALICE, in which a well known problem is faced: the integration between LISP and Horn clauses. This exercise is motivated by a recognized lack of experience in implementing PISs to deal with actual A.I. programming problems. ALICE is composed of two procedurally introspective languages based on LISP and on Horn clauses, respectively. The integration is achieved by means of a new kind of reflection called mutuai *reflection.* The design of ALICE required the generalization of several concepts and mechanisms introduced in 3-LISP. The discussion is completed with a set of general retrospective considerations.

I INTRODUCTION

The mcta-levcls architectures (P.Maes and D.Nardi, 1987) constitute one of the most interesting research lines in the field of knowledge representation. In this kind of systems, some (meta) knowledge about the structure and the role of the domain knowledge can be explicitly represented. The problem solving activity is thus carried on by alternating the use and the transformation of domain and meta knowledge. An interesting subclass of the meta level systems is constituted by the *introspective systems.* In this case, in the system is explicitly available a description of the structure and of the behaviour of the system itself. The system is thus able to reason about itself in some ways.

A quite general definition of introspective systems is proposed in (B.C.Smith 1984). It is given by means of three conditions that have to be satisfied by a candidate introspective system, i) The system must embody in itself a description of the system which can be consulted and modified by means of tools available in the system, ii) The description has to be *causally connected* to the system structure and behaviour. Any event and relation in the system must have a corresponding representation in the description and any modification in the description must cause a modification in the structure and behaviour of the system, iii) The description must have a proper vantage point. It has to represent the system at a right level of detail to the extent of introspection. (The definition in (P.Maes 1987) is more detailed, but for our work the above structural definition is accurate enough).

The first proposal for an introspective system, with respect to the definition above, was 3-LISP (B.C.Smith 1984). 3-USP, as well as Brown (M.Wand and D.P.Friedman, 1986), arc *procedurally introspective* systems. They restrict their attention on procedural knowledge. So the relevant aspect in computational processes is the behaviour to achieve. In such a framework, the expressions of a language are description about how to manipulate domain objects and a language is described by its interpreter. This restriction is useful to concentrate the efforts on the design of the basic mechanisms for introspection. In these systems, an explicit representation of the interpreter for the language is available and causally connected to the behaviour of the system itself. *Reflection* is the mechanism by which the user can change the focus of attention from the domain level to the interpreter level to inspect and influence the current computation or to modify the interpreter's code. Reflection can be used during meta level operations, thus the user can make access to a virtually infinite tower of causally connected interpreters each interpreting the one below. PISs actually are a first step toward general introspection.

The 3-LISP approach to introspection, although constrained to the procedural aspects, is elegant and very powerful. Nevertheless, this kind of ideas have had a relatively limited spreading in the A.I. applications community and a very small set of relevant examples of introspective programming are available. In our opinion, these problems are caused by the lack of a general understanding of the power of PISs and of a pragmatical experience on them. In fact, i) 3-LISP and Brown are quite difficult to use, due to the unusual power of reflection as a programming tool; ii) few PISs have been developed on languages different from LISP; iii) the 3-LISP and Brown implementations are quite complex to understand and they arc based on a set of innovative concepts. It is currently difficult to abstract and export the basic mechanisms of procedural introspection.

So, all we need is experience in implementing procedural introspection and in exploring its capabilities while facing actual A.I. programming problems.

The ALICE system is an attempt to deal with the integration of LISP and Horn clauses in a conceptually clean way in the framework of PISs using reflection as a tool for integration. The currently available solutions are based mainly on the implementation of a Horn clauses resolutor embedded in the LISP environment (J.A.Robinson and E.E.Sibert, 1982) (CMellish and S.Hardy, 1983). We want to achieve the goal without privileging a single language to which all the representations eventually collapse in, and without introducing an ad hoc communication language. ALICE was developed at the University of Milano and a complete running version of it is currently available in Franz Lisp on a VAX 11/750.

As ALICE is mainly an exercise in procedural introspection, the focus of attention is on the experience in using and implementing PISs. This paper briefly presents the work on ALICE in this perspective. We follow a step by step approach. Firstly, a new procedurally introspective dialect of LISP is presented.Then, an independent PIS, based on Horn clauses, is introduced. Finally, the two languages are integrated into a single PIS. This requires a new point of view on self description and the introduction of the mutual reflection mechanism. Due to space problems, the presentation is very schematic and all the details are omitted. A complete report on ALICE can be found in (R.Ghislanzoni, L.Spampinato and G.Tornielli 1986), (R.Ghislanzoni, L.Spampinato and G.Tornielli 1987). We assume the reader knows the ideas in (B.C.Smith 1984) and we don't undertake a presentation of this theory and we use the Smith's notation in his same sense.

II STEP ONE: A PROCEDURALLY INTROSPECTIVE LISP A LITTLE MORE GENERAL THAN 3-LISP

The LISP part of ALICE is based on 3-LISP, of which we adopt all the basic concepts but for two points: the use of explicit data structures instead of closures to represent continuations and the introduction of *reflecters* instead of lambda-reflect

The first choice de-couples the processed language from the language in which the processor is written. The one-language choice binds strictly introspective actions with structures of the particular implemented language. In the perspective of mutual reflection, the representation of the future of the normalization process must be abstracted from the implemented language.

In the LISP part of ALICE, a reflection is achieved by means of a reflecter: a new type of pair (notated as a pair but preceded by a ~), which indicates that the function application has to be performed one level above the current one. So, there are only *simple* lambda expressions which are possibly *reflectible.* For example, in 3-LISP:

```
(DEFINE QUIT (LAMBDA REFLECT [Q ENV CONT] 'QUIT!))
3-USPI>(QUIT)
3-USP2>'QUIT!
```

In ALICE LISP:

```
(define quit (lambda [[] env com] 'quit!))
ALICE-LISP 1> -(quit)          ALICE-LISP 1> (quit [] [] [])
ALICE-LISP 2> 'quit!           ALICE LISP 1 > 'quit!
```

In PIS we consider three concepts about procedures: procedure definitions describe manipulations on structures, procedure applications describe which manipulation on which structures and reflecters describe at which level the manipulation has to be realized. In 3-LISP the first and the third mechanisms are too strictly coupled. Reflection is not a static property of a procedure but a particular way to use it

The implementation of the ALICE LISP is very similar to the 3-LISP one. The finiteness criteria presented in (B.C.Smith and J.des Rivifcres 1984) apply directly to our modified processor definition (reflecters are treated by the normalize function).

III STEP TWO: A PROCEDURAL INTROSPECTIVE SYSTEM FOR LOGIC PROGRAMMING

The second part of ALICE is called Logic. Logic, as an independent system, is a simple programming language, in which programs arc sets of Horn clauses. The basic idea is to recover as much as possible of the clear and simple initial concept of logic programming (R.A. Kowalski 1974). We are willing to obtain at least the power of current PROLOG (L.Sterling and E.Shapiro 1986) by means of introspection. The goal is to build a system based on 'pure' refutation at domain level and on explicit procedural import at meta level. From this point of view, Logic can be considered yet another proposal in the set of Horn clauses based languages with meta level programming (K.A.Bowen and R.A.Kowalski 1982).

Sentences, clauses, conjunctions and variables are represented with new types of structures added to the structural field of the LISP part. The structures of the LISP part are terms for Logic.

We define the *procedural import* (1st factor,) and the *designation* (2nd factor 4>) (B.C.Smith 1984) of Logic with respect to refutation realized with the SLD resolution procedure. Thus, the designation of a sentence is the truth value TRUE or FALSE whether it can be proven in the current theory with the defined procedure. The refutation by SLD procedure defines the behaviour of Logic: it actually constitutes the normalization process. Thus sentences normalize to the boolean $T or $F whether the procedure refutates them or it doesn't.

In order to make Logic semantically flat we defined a category alignment for it (R.Ghislanzoni, L.Spampinato and G.Tornielli 1986), following the way traced by B.C.Smith.

Logic contains the code of a metacircular processor as the description of its own behaviour. It is the code (in Logic) of an implementation of the refutation by SLD resolution procedure. The state of the normalization process is explicitly represented by a search tree and an environment, which is the structure representing the current theory. The resolution tree is a complex rail in the structural field with construction and selection primitives.

Reflective acts are represented in Logic with *logic reflecters.* Their notation is similar to sentences' one but with a - at the beginning. The basic idea of reflection is the same of the LISP part: the atomic sentence in the reflecter is modified to include the terms for the representation of the environment and the tree and it is processed as it were inserted in the processor's code at a defined point.

The implementation of Logic is also based on a shifting processor which recognizes the need of a reflection, makes explicit *(reifies)* the state of the computation, and shifts up to process the processor code augmented with the user reflective code. If it realizes it is processing a part of the processor code, it shifts down to directly process the user code, after recording the state of the upper level for a potential successive reflection.

The precondition for the realization of a shifting processor is the finiteness of the metacircular processor which describes the system. The point, in the context of Logic, is the creation of the representation of the search tree to be used by the processor at the level n+2 when a reflection takes place carrying the computation from level n to level n+1. We need a standard way to produce a search tree for the state of a processor processing itself. Otherwise, we have to provide the state of the processor at any level and a finite implementation can't be afforded.

As for continuation in 3-LISP, it is possible to verify that the search tree of the processor processing the processor is always the same. The tree must not change when considered at any two successive resolutions involving a processor's clause. In (R.Ghislanzoni, L.Spampinato and G.Tornielli 1986), we introduce a concept of invariance for the resolution tree of Logic. We also provide an informal proof of the invariance of the Logic processor tree while processing itself.

The construction of a new PIS, based on a language so different from LISP, has revealed very interesting and stimulating although not completely straightforward. It lead us to the following considerations.

1) The crucial points are the choices of a proper representation of the normalization process' state and the way reflection is described in the processor's code. The first choice has to be made in the perspective of the system actual use. In Logic, the representation of the search tree was designed to achieve the power of current logic programming systems by means of reflection. For SLD a stack is enough, but a richer representation allows the user to profoundly influence the search strategy writing simple reflective code.

2) The mechanism which restarts the computation after each input-normalize-output loop, is quite difficult to design. In Logic, the generalization of the reply-continuation approach of 3-LISP, required several attempts and the exploitation of some tricky properties of the resolution tree representation.

3) The finite nature of the processor can be a little painful to verify also if it is easy to guess. The notion of tail recursion must be considerably generalized to be applied to interpretation structures more complex than a stack.

IV STEP THREE: INTEGRATION AND MUTUAL REFLECTION

We introduce mutual reflection with two observations about the preconditions for procedural introspection.

1) Only when introspection takes place (by means of a reflective act), it is necessary to explicitly know the formalism in which the processor is written. If a program does not introspect, it is processed by an implicit processor the description of which has not a role in computation. In other words, the actual form of the processor description has to be fixed when some reflective code makes access to it.

2) The description of the system itself, which a PIS embodies, must: i) completely describe the reflection mechanism; ii) be suitable for a finite implementation (shifting processor); iii) be completely processed by the processor it defines. Such requirements are implicitly satisfied by 3-LISP and by every PIS based on a single language. Nevertheless, it is not mandatory for the self-description of a PIS to be homogeneous with respect to the representation formalism.

With *mutual reflection* the formalisms in which the processor currently running is explicitly represented can be chosen dynamically at each reflective act.

In ALICE the user can employ, at the starting level, LISP or Horn clauses starting the normalization of a program in the chosen formalism; let's say LISP. If there are no reflections, the running program is normalized by a LISP processor whose description is unspecified. In the user program LISP or Logic reflecters can be freely used. The normalization of a LISP reflecter forces the current processor to be represented in LISP and the reflective user code is merged in it. The second level processor is a LISP one but its representation remains unspecified. The normalization of a Logic reflecter at the starting level forces the processor at the first level to be a LISP one represented in Logic and the reflective user code (a sentence) is merged in it. So the second level processor is a processor for Logic but its representation remains unspecified. When starting from Logic the same holds with Logic for LISP and vice versa. As ALICE is a PIS, reflective acts give access to the representation of a normalization process' state.

This behaviour can be obtained if the descriptions of four processors are available: one for each pair representation language - normalized languago. The normalization of a LISP reflecter in a LISP program gives access to the LISP in LISP processor. A Logic reflecter in a logic program gives access to the Logic in Logic one. The normalization of a LISP reflecter in a Logic program gives access to the Logic processor written in LISP and a Logic reflecter in a LISP program gives access to the LISP processor written in Logic. In this sense we speak about mutual reflection.

Interaction is now easy. When the user introduces a reflection, whatever reflecter she may use, she can both get the state of the current normalization of her code in the starting language and start a new normalization process with respect to the other language. In this way she is given a simple and powerful tool to construct the 'bridge' between Horn clauses and LISP more convenient for each specific application.

The ALICE description available (in ALICE) is composed of two parts. The first consists of the code of the two processors written in LISP (the LISP description). The second consists of the two processors written in Logic (the Logic description). Each description represents the two normalization processes. Thus, each of the normalizations has a double description. The choices in the representation of the state of the normalization process are the same for the corresponding descriptions. This makes it possible to keep the representation of the state independent from a normalization process and from the particular language the processor is written in.

The LISP description represents the way LISP reflecters are processed (reflection mechanism) when they are inserted both in LISP and in Logic code. The Logic description represents how Logic reflecters are processed symmetrically. In this sense, the two descriptions are complementary in describing the whole system. ALICE is procedurally introspective as a whole but its model of itself is decomposed in two parts. It has a single polymorphic introspective ability.

A finite implementation is possible due to the finiteness of the whole ALICE description as informally proved in (R.Ghislanzoni, L.Spampinato and G.Tornielli 1986).

V EXAMPLES

In ALICE it is possible to move the computation above the current level and start an interaction with the chosen processor. The reflectible function quit, as defined above, allows to move in the LISP description of the current level:

```
'Lisp 1> "(quit)          'Horn 34> "(quit)
'Lisp 2= 'quitted         'Lisp 35= 'quitted
```

One can also move in the Logic description of the current level:

```
{clauses: {QUIT :x :y :z}}
```

```
•Hornl> ~{QUIT}          *Lisp89> '{QUIT}
'Horn2=$F                'Horn90« $F
```

A reflectible ALICE LISP function to be used in LISP reflecters within Logic programs follws. It asks about the provability of a sentence when it cannot be proven otherwise.

```
(define ask (lambda [[to-prove] env tree]
    (let  [[unif (unif-env-and (and-level tree))]]
        (cond  [(prove env (make-tree to-prove unif))
                (prove env tree)]
               [$F (block
                    (output  'can-be-proven?)
                    (output (expand to-prove unif))
                    (if (= (input) 'yes)
                       (prove env tree)
                       (prove env
                             (clcar-and-level tree)))))]))))
```

For example:

```
    'Horn 1 > {A 1}
    Horn 1 = $F
    Horn 1 >~(ask{Al})
    'can-be-proven? '{A 1}
    yes
    'Horn 1 « $T
```

The last example is a couple of Logic reflectibe predicates to be used in Logic reflecters within a LISP expression. It is the definition of the well known *catch-*throw control structure.

```
(clauses: {rule {*catch :args xatch-env xatch-cont :result}
        {lst:args:tag}
        {2nd :args xatch-body}
        {up xatch-cont :upped-catch-cond}
        {bind :tag xatch-cont xatch-env :augmented-env}
        {normalize :catch-body :augmented-env
                                    xatch-cont :result}}}
{clauses: {rule {*throw :args :throw-env :throw-cont :result}
        {1st :args :tag}
        {2nd :args :throw-exp}
        {binding :tag :throw-env :upped-catch-cont}
        {down :upped-catch-cont xatch-cond}
        {normalize :throw-exp :throw-env
                                    xatch-cont :result}}}
```

They can be used as follows:

```
(define member (lambda [atm list]
    -{♦catch stop
        (mapc (lambda [x]
                      (if(=xatm)
                        ~{*throw stop $T}))
              list)}))
```

VI CONCLUSIONS

The experience of the ALICE project lead us to a couple of general consideration about the actual development of PISs.

1) The shifting processor approach can actually help in embodying procedural reflection in systems more complex than 3-LISP. Moreover, the method - implicit in it - of incremental generation of the explicit representations when they are needed, make it possible to use reflection as a powerful tool for the integration of different formalisms.

2) Mutual reflection can be extended to any set of procedural languages, provided that each of them can be implemented in a PIS and a new reflecter type can be added to its structures. This extended ALICE system would have *n* descriptions of itself, (for *n* involved languages). Each of the descriptions would contain an interpreter for each language and describes the system when only reflecters of the language in which is written are used. All the descriptions together would constitute the self description of the system which would be procedurally introspective as a whole. In this sense, ALICE can be seen as an achitectural proposal for systems in which many different formalisms are integrated in a modular way.

ACKNOWLEDGEMENTS

We are grateful to G.Degli Antoni, B.C.Smith and J.des Rivieres for the discussions which were of great help in making clear to us the role of our work. We would also thank D.Friedman for his encouraging comments .

REFERENCES

K.A.Bowen and R.A.Kowalski, *Amalgamating Language and Metalanguage in Logic Programming,* in K.L.Clark and S-A.Tarnlund (eds.), *Logic Programming,* Academic Press, London, 1982.

R.Ghislanzoni, L.Spampinato and G.Tornielli, *The Reflective System ALICE,* QUINARY QR-86-2, Italy, 1986.

R.Ghislanzoni, L.Spampinato and G.Tornielli, *Communication between USP and Horn Clauses by Mutual Reflection,* In (P.Maes and D.Nardi 1987)

R.A.Kowalski, *Predicate Logic as a Programming Language,* Proc. of IFEP Conference, 1974.

P.Maes, *Computational Reflection,* Ph.D. Thesis, Vrije Universiteit Brussels, Belgium, 1987.

P.Maes and D.Nardi (eds.) *Proc. of of the Workshop on Meta Level Architectures and Reflection,* Alghero, Italy, North Holland, 1987 (in publication).

CMellish and S.Hardy, *Integrating Prolog into the POPLOG Environment,* Proc.UCAI-83, Karlsrhue, RFG, 1983.

J.A.Robinson and E.E.Sibert, *LOGUSP: Motivations, Design and Implementation,* in K.L.Clark and S-A.Tarnlund (eds.), *Logic Programming,* Academic Press, London, 1982.

B.C.Smith, *Reflection and Semantics in USP,* Xerox PARC ISL-5,PaloAltoCA, 1984.

B.C.Smith and J. des Rivieres, *Interim 3-USP Manual,* Xerox PARC ISL-1, Palo Alto CA, 1984.

L.Sterling and E.Shapiro, *The Art of Prolog: Advanced Programming Techniques,* MIT press, London, 1986.

M.Wand and D.P.Friedman, *The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower,* Proc.ACM LISP Conference, Boston MA, 1986.

$