*The Knowledge Engineering Review*

# Computational reflection

PATTIE MAES
*AI-LAB, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, BELGIUM*

**Abstract**

Computational reflection is the activity performed by a computational system when reasoning about (and by that possibly affecting) itself. This paper presents an introduction to computational reflection (thereafter called reflection). A definition of reflection is presented, its utility for knowledge engineering is discussed and architectures of languages that support it are studied. Examples of such procedural, logic-based, rule-based and object-oriented languages are presented. The paper elaborates on the design of these languages and the reflective functionality that results, elucidating concepts such as procedural reflection, declarative reflection, theory relativity of reflection, etc. The paper concludes with an assessment of outstanding problems and future developments in the area.

## 1 Introduction

This section presents a definition of reflection applicable to any model of computation, whether it be procedural, deductive, imperative, message-passing or other. We define *computational reflection* to be the behaviour exhibited by a reflective system, where a *reflective system* is a computational system which reasons about itself in a causally connected way.[1] In order to substantiate this definition, we next discuss relevant concepts such as "computational system", "reasoning about" and "causal connection".

A *computational system* is a computer-based system that reasons about (i.e. answers questions about and eventually also supports actions in) some problem domain. It grounds this activity on internal structures that represent this domain. These representations, together with the inference rules which define what can be deduced from them, determine the *knowledge* the system has about its domain. Notice that a representation involves two things: the representation itself (i.e. an internal structure), and that which it represents (i.e. the referent of the internal structure in the world).

A system is said to be *causally connected* to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other. A system steering a robot arm, for example, incorporates structures representing the position of the arm. These structures may be causally connected to the position of the robot's arm in such a way that (i) if the robot arm is moved by some external force, the structures

[1] Recently, B. Smith has adopted a different definition for reflection (Smith, 1986). What we call reflection here, he now calls introspection. Smith would say that an introspective system is also actually reflective if it has a representation of (and is thus also able to reason about and act upon) itself relative to it$ embedding world. For example, if it has a reflective representation about whether its data are an honest representation of the domain or not. Another example would be that the system can represent at a reflective level that the semantics exhibited by its program are inconsistent with the real semantics of the domain. This notion of reflection has sprung out of the idea of the *circumstantial relativity* of language and thought. According to (Smith, 1986) a representational structure not only has

(1) a *content*, that which the structure refers to in a particular use of the structure (depending on the set of circumstances of that particular use),
(2) a *meaning*, indicating what and how this representational structure contributes to the content of any larger structure in which it participates (something the structure has on its own),

But a representational structure also has, and this is new,

(3) a *significance*, including not only the content of a particular use, but the full conceptual and functional role that the representational structure plays in and for the system that incorporates it.

Content and meaning of a structural representation are specifiable independent of conceptual and functional role. But a great deal of the significance of a representational structure is not directly or explicitly represented by any of the structures of which it is composed. Instead it is often relative to its circumstances. Systems which are able to reason about and act upon themselves in more than just a local way, which are able to "de-relativize" their thoughts and actions of the "here, now and self", are called reflective systems according to Smiths definition. No concrete reflective systems have yet been built.

change accordingly and (ii) if some of the structures are changed (by internal processes), the robot arm moves to the corresponding new position. So a causally connected system always has an accurate representation of its domain and it may actually cause changes in this domain as a mere effect of its reasoning.

A reflective system is a system which incorporates structures representing aspects of itself.[2] We call the sum of these structures the *self-representation* of the system. This self-representation makes it possible for the system to answer questions about itself (in a limited way) and support actions on itself. Because the self-representation is causally-connected to the aspects of the system it represents, we can say that:

1 The system always has an accurate representation of aspects of itself.
2 The status and behaviour of the system are always in compliance with this representation. This means that a reflective system can actually bring modifications to itself by virtue of its own reasoning.

## 2 Examples of reflection

At first sight, the concept of computational reflection may seem a little far-fetched. However, if we take a closer look at the computational systems that are built everyday, especially the knowledge-based ones, we can identify many instances of reflective computation, implemented in very ad hoc ways. This section presents a few such examples. They clearly demonstrate that a lot of functionalities in computation require reflection and that consequently the concept of reflective computation should be supported by programming languages.

*Example 1*
The need is often felt in expert systems to reason about the status and computation of the system itself during computation. The expert system shell EMYCIN, for example, provides several reflective mechanisms allowing an expert system to change the flow of its own computation at run time. Consider the following example from MYCIN (Van Melle, 1980):

(a) IF it is not known whether there are factors that interfere with the patients bleeding, THEN it is definite (1.0) that there are no factors that interfere with the patients bleeding.

**Figure 1** Reflective computation in EMYCIN

Reflective computation is used here to deal with the problem of incomplete knowledge. Rule (a) shows a special type of rule, called a self-referencing rule. This rule refers to the system's own status. It is used to form a conclusion out of uncertainty. When all the regular rules have failed to make a conclusion about whether there are interfering factors or not, rule (a) will conclude that there are no interfering factors.

Notice however, that this functionality is wired into the inference engine in an ad hoc way. In principle, the order in which rules are executed in EMYCIN is arbitrary. Yet, in order to be able to deal with self-referencing rules, the EMYCIN interpreter incorporates an obscure piece of code, which ensures that self-referencing rules are only applied after all regular rules that can possibly make a conclusion about the goal have been consulted.

EMYCIN incorporates other reflective mechanisms to alter the flow of computation from within the computation, such as prevention of circular reasoning, antecedent rules, the preview mechanism, the concept of a unity path, the initial-data mechanism, etc. (c.f. Van Melle, 1980). The primary need for these mechanisms lies in the flexible control of the flow of computation. Reflection allows systems to decide about what to do next, not only on the basis of data about the problem domain, but also on the basis of data about the status of the program.

[2] There are of course limitations to the self-representation of a system. First, because it is an intrinsic property of representations that they never contain complete information about the thing they represent and second, because there are limits to how far a system can modify itself and still be able to run. There is necessarily a kernel of the system which cannot be made explicit and modifiable (c.f. problems of bootstrapping).

*Example 2*
LISP incorporates various functions which are actually of a reflective nature. Examples are boundp, eval, catch and throw, etc. Some of these actually modify the program, the run-time environment or the interpreter, by means of destructive operations. We present an example of a concrete use of such a reflective construct provided by Zeta-LISP (Weinreb and Moon, 1981). This facility is refered to as "condition signalling and handling". It makes it possible to set up a monitor which temporarily watches the computation and checks whether a certain event happens. This monitor may, for example, be on the look out for whether an error occurs, whether a certain variable is set, or whether a variable obtains a specific value. If the event takes place, the monitor will activate a procedure, called a "handler", which can alter the flow of computation.

Consider a function "search-graph" defined as

```
(defun search-graph (node attribute)
      (if (has-attribute node attribute)
         node
         (if (father-node node)
            (search-graph (father-node node) attribute)
            NIL)))
```

The purpose of the function is to search an inheritance-graph of nodes, in order to find the father-node of a given node which has an attribute with a given name. When the user defines a circular inheritance-graph by accident, this will cause the function search-graph to loop infinitely. Figure 2 illustrates how this special event can be guarded by a condition signaller and handler.

```
(defun search-graph (node attribute)
   (catch 'circular
      (condition-bind ((sys:pdl-overfow
                          '(lambda (error-flavor-instance)
                               (circularity-checker
                                  ',node
                                  error-flavor-instance))))
         (if (has-attribute node attribute)
            node
            (if (father-node node)
               (search-graph (father-node node) attribute)
               NIL)))))

(defun circularity-checker (node error-flavor-instance)
      (do ((already-encountered ()
                               (cons current-node already-encountered))
           (current-node node (father-node current-node)))
          ((or (member current-node already-encountered)
               (null (father-node current-node)))
           (if (father-node current-node)
              (throw 'circular
                     (format t "The computation was halted due to an
                                error in the inheritance-graph. Starting
                                from node ~A, a circular path of
                                father-nodes exists. Correct this
                                immediately to avoid further problems."
                             current-node))
              (send error-flavour-instance :proceed :grow-pdl)))))
```

**Figure 2** Conditions in Zeta-LISP make it possible to alter the flow of control from within the computation

Sys:pdl-overflow is one of the standard error events recognized by the Zeta-LISP interpreter. It is signalled when there is a stack-overflow. The condition-bind construct defines a local handler for this error. If the stack overflows in the interpretation of the body of the function search-graph, the form

(circularity-checker node error-flavor-instance)

will be executed. The function circularity-checker tests whether the graph is really circular (it might just be a very large graph). It puts the nodes that it encounters in the list "already-encountered". Note that this function was written in an iterative way, because it would otherwise also cause the stack to overflow. If the graph is circular, i.e. if current-node is a member of the list of already-encountered nodes, a message is printed and the interpretation of search-graph is halted. If the graph is not circular, i.e. if the current-node has no father-node, the computation will proceed from the error. The computation that was interrupted is continued with a larger stack.

Condition signalling and handling is used to implement limits on the available resources for computation, to implement active values, to implement default computation, to recover from errors, to add "side-computation" to computation (e.g. stepping, tracing), etc. LISP also provides other reflective functions, for example to access and modify the list of variable bindings, etc.

*Example 3*
Frame-based languages (Minsky, 1974; Roberts and Goldstein, 1977) have introduced the idea to represent the knowledge required for acquiring, maintaining, using and communicating domain knowledge explicitly within the knowledge-based system itself. A knowledge-item in a frame-based system is surrounded by a whole set of reflective data and procedures. These are used for different purposes:

- they help the user cope with the complexity of a large system by providing documentation, history, and explanation facilities,
- they keep track of relations among representations, such as consistencies, dependencies and constraints,
- they encapsulate the value of the knowledge-item with a default-value, a form to compute it, etc.,
- they guard the status and behaviour of the knowledge-item and activate specific procedures when specific events happen (e.g. the value becomes instantiated or changed).

Not only does the knowledge for the maintenance, acquisition and communication of the information contained by a system become represented in a uniform format, but it also becomes explicit, which means that the system itself can also make use of it, and take over part of the responsibility for these tasks.

Figure 3 gives an example. It represents the general form and a specific instance of a frame. The item "age of John" in the knowledge base contains a lot of data which convey internal system information. Apart from the slot "value", all the slots of an item represent reflective data about this item. These data are frequently used during update and retrieval of the value of the item. For example, when the item "age of John" is asked for its value, the interpreter checks whether this item is showable to the outside world (the salary of John would, for example, not be showable). When somebody wants to set the value of the age of John, the interpreter checks whether this item is modifiable, whether the proposed value is of the appropriate type and whether this value fulfills the constraints. Subsequently, it will reset the slot source-of-current-value to the supplier of the new value, and the slot items-this-value-depends-on to NIL. Finally the interpreter will set the values of the items listed in items-depending-on-this-value to undefined.

Frames also make it possible to specify reflective computation about a knowledge-item. They allow definition of subroutines which are activated by specific events in the computation. Typically, these implement tasks such as consistency maintenance, documentation and explanation facilities, acquisition of new data, appropriate communication of data, garbage collection, etc. Figure 4 shows an example.

```
ITEM a name
   value: the current value of this item
   source-of-current-value: user-supplied/default/computed/...
   modifiable: yes/no
   showable: yes/no
   documentation: a string documenting this knowledge-item
   part of: the more global item this item is part of
   author: the name of the author of this knowledge-item
   when-created: the date this item was created
   when-last-accessed: the date the value of this item was last accessed
   constraints: a list of constraints the value has to fulfill
   type: the type the value has to fulfill
   comparison-function: the method that should be used to compare the
                        value of this item with another value
   default-value: the value to be used when no value is stored and no
                  value can be derived or computed
   items-depending-on-this-value: the list of information items that
                                  have made use of this value to compute
                                  their own value
   items-this-value-depends-on: the list of information items whose
                                value was used to compute this value
ITEM age of John
   value: 27
   source-of-current-value: computed
   modifiable: yes
   showable: yes
   documentation: "this item represents the age of the object John"
   part-of: John
   author: pattie
   when-created: 3/10/76
   when-last-accessed: 7/12/85
   constraints: (and (<0 value) (>100 value))
   type: integer
   comparison-method: equal
   default-value: 25
   items-depending-on-this-value: (adultp-of-John)
   items-this-value-depends-on: (birthyear-of-John current-year)
```

**Figure 3** A frame includes reflective data about a knowledge-item

Several reflective procedures are associated with the knowledge item representing John's salary. When the salary of John is asked for, the system first checks whether this is a showable item. If not, it is not returned. Otherwise, the present value of the salary of John is returned and the system records the fact that somebody accessed this item at the current moment. When the salary of John is assigned a value for the first time, the system sets up the procedures that ensure John's salary is paid every month.

When somebody wants to modify the value of John's salary, the system checks whether (i) the requester is allowed to modify the salary of John and (ii) whether the proposed salary is acceptable. After the change is made, the system takes the appropriate actions to ensure the consistency maintenance of the knowledge base. All items in the knowledge base whose value depends on the value of John's salary will have to be rejected. For example, John's social security contribution will receive a new value.

ITEM *a name*
  ⋮ reflective data (c.f. above)
  before-retrieved: *a procedure to be executed before the item is asked for its value*
  after-retrieved: *a procedure to be executed after the item is asked for its value*
  when-initialized: *a procedure to be executed when the item is initialized*
  before-modified: *a procedure to be executed before the value of the item is modified*
  after-modified: *a procedure to be executed after the value of the item is modified*
  when-displayed: *a procedure to be executed when the item is displayed*

ITEM salary of John
  value: 23
  ⋮ reflective data (c.f. above)
  before-retrieved: *a procedure which checks in the slot "showable" whether the value of this slot is public*
  after-retrieved: *a procedure which resets the slot when-last-accessed to the present date*
  when-initialized: *a procedure which sets up procedures to pay John's salary every month*
  before-modified: *a procedure which checks whether: (i) the requester is allowed to modify the salary of John, (ii) whether the proposed salary is reasonable, etc.*
  after-modified: *a procedure which handles the consistency of the knowledge base*
  when-displayed: *a procedure which prints the salary of John with a £ sign in front of the amount*

**Figure 4** A frame includes reflective procedures about a knowledge-item

The success of the reflective facilities provided by frames is unquestionable. The idea has been incorporated in almost all commercially available expert system shells (c.f. KEE (Intellicorp, 1985) or ART (Clayton, 1985)).

## 3 The use of reflection in knowledge engineering

### *3.1 Reflective systems can be more flexible and robust*

The primary purpose for introducing reflection in knowledge-based systems is to make these systems more flexible and more robust. "Robust" meaning that they fulfill their functionality in a large number of different situations. "Flexible" meaning that they adapt their strategies for fulfilling their functionality to the specific situation at hand.

One of the fundamental problems of knowledge-based systems is that they are subject to *epistemological limitations* (Steels, 1985; Steels, 1987): the knowledge they can bring to bear to solve a particular problem is typically incomplete, inconsistent, and error-prone. This is the case because

(1) they are physical, limited systems,
(2) they have to fulfill some functionality in some environment, and
(3) they work with imperfect (i.e. natural) knowledge about their domain.

Examples of limitations imposed by their physical limitedness are finite memory and finite processor power. Examples of limitations imposed by the external environment (remember that a concrete system has to fulfill some functionality in the real world) are: the maximum time available for the reasoning task, or the relative importance of the reasoning task in the context of the current situation. Examples of limitations imposed by the limits of knowledge are: the world is hard to model, the world changes dynamically (through time and other agents), etc.

Systems which do not recognize the existence of epistemological limitations and thus do not foresee strategies to cope with them, fall victim to these limitations. They adopt the hypotheses that (i) they have unlimited resources, (ii) are not constrained by external factors and (iii) have perfect knowledge. As a consequence they are not very robust, in that they fail to fulfill their functionality as soon as one of these three hypotheses is not fullfilled. Another consequence is that they are not very flexible, in that they do not bother to adapt the realization of their functionality to the specific situation at hand (e.g. little time to find a solution versus plenty of time).

Notice that incorporating learning techniques into systems does not solve these problems. Most of the limitations discussed here are also valid for systems which automatically acquire their domain knowledge (i.e. it is unrealistic to think that a system could through learning, acquire perfect knowledge). The only solution for a system not to fall prey to these limitations is that the system is able to explicitly reason about them and incorporates strategies for coping with them. This implies that a system is reflective: it has to be able to reason about and act upon its knowledge, its functionality and its status.

For example, a knowledge-based system which has a representation of its own means (possible reasoning steps to take) and ends (the system's goal functionality) can exhibit flexible forms of reasoning. It can decide about what to do next, not only on the basis of data about the problem domain, but also on the basis of data about the current status (what knowledge is available, what are the limitations imposed by the current status of the environment and the current internal status, etc.). It is also more robust: it can use its reflective capacity to prevent circular reasoning, to implement defaults when not enough knowledge is available (or when there is no time to make use of the relevant knowledge), or to recover from impasse situations (e.g. conflict-resolution), and so on.

## *3.2 Enhancing the modularity of computational systems*

There also exists another, even more practical reason for the use of reflection. Besides domain-computation, most computational systems also exhibit some form of reflective computation which contributes to the internal organization of the system or to its interface to the external world. Examples of such reflective computation are to keep performance statistics, to keep information for debugging purposes, stepping and tracing facilities, interfacing (e.g. graphical output, mouse input), self-optimization (e.g. caching) and self-activation (e.g. through monitors or deamons). This type of computation is so inherent in real world computational systems that it should be supported as a fundamental concept in programming languages.

Programming languages today do not fully recognize the importance of reflective computation. They do not provide adequate support for its modular implementation. For example, if the programmer temporarily wants to follow the computation, e.g. during debugging, he often changes his program by adding extra statements. When debugging has finished, these statements have to be removed again from the source code, often resulting in new errors.

Some types of reflective computation might be supported by a programming environment for the language. Nevertheless, in general, the reflective computation of systems has to be programmed in ad hoc ways and reflective code has to be mixed into object-level code. Often programmers are forced to re-implement reflective computation over and over again for every new application.

Important advances in programming are often the result of the introduction of larger modular structures. Consequently, the evolution of programming languages is characterized by a search for more modularity. We believe that explicit support for reflective computation is such a step towards more modularity. As is generally known, enhanced modularity makes computational systems more manageable, more readable and easier to understand and modify. But these are not the only advantages of a decomposition of computation into object-computation and reflective computation. What is even more important is that it becomes possible to introduce computational abstractions which facilitate the programming of reflective computation in the same way abstract control-structures such as DO and WHILE facilitate the programming of control flow.

### *3.3 What not to do with reflection: dangers of reflection*

The previous sections showed that reflection provides interesting solutions to many programming problems. Although the examples prove that reflective computation can be very useful, there is a danger attached to its use.

There are limits to controlling and understanding reflective computational systems. A reflective system is able to make modifications to itself. This means that some of the control over computation is actually shifted from the programmer to the system. Consequently programming the computation of such a system is less straightforward. It involves programming the object-level computation, reflective computation, and so forth. This may become an extremely difficult task, since these different levels of computation actually act on one another. The computation of a specific level of computation is not only determined by the code for that level, but also by the code of the reflective levels above that level.

The shift of control also implies a loss of information. For example, if we build a system that is able to handle exceptional or erroneous situations, the programmer/user is no longer informed about these situations. It might be that the fact that a symbol is not bound for example, indicates to the programmer that his program is not correct. Often a programmer wants to know when and why a variable is not as it should be. This certainly suggests that reflection should not be used for exception-handling during the development phase of a system.

Another problem in reflective systems is to understand what a computational system actually does. Reflection makes the semantics of a system more explicit to the system itself. But at the same time, since a system may modify itself, the semantics becomes less clear to the programmers and users of a system. Actually, the semantics of a language becomes open-ended by reflection. The (object-) program of a system is no longer a means for understanding the behaviour of a system. It might be, for example, that at run-time a different program is executed, or that a different interpreter is executing the program.

Clearly reflection might sometimes be a dangerous facility. However, it would be wrong to conclude that the programming community should discard reflection. All powerful engineering tools can be dangerous (c.f. for example thc LISP-machine). One solution to the problem is a discipline in the use of reflection. What this discipline should look like is not yet known. Now that reflection is becoming better understood, further research into this area, can be entered into. We need to experiment with reflection in order to distinguish its positive and negative uses. On the basis of such a study, safer (weaker) versions of reflective facilities will have to be designed.

## 4 Reflective architectures

The previous section argued that a lot of computational systems might profit from reflective facilities. Some programming languages have responded to this need by incorporating reflective constructs without recognizing reflection as a primary programming concept. The reflective constructs they support are not part of the kernel of the language. Consequently, these languages only support a finite set of reflective constructs, designed and implemented in an ad hoc way. Often different dialects of the language support different reflective functionalities.

A programming language with a *reflective architecture* recognizes reflection as a fundamental programming concept and thus provides tools for handling reflective reasoning explicitly.[3] This means that:

(1) The interpreter of such a language has to give any system that is implemented in it access to structures representing (aspects of) the system itself. These systems then have the possibility of performing reflective reasoning by including code that prescribes how these structures may be manipulated.
(2) The interpreter has to guarantee the causal connection between these structures and the aspects of the system they represent. Consequently, the modifications these systems make to their self-representation are reflected in their own status and behaviour.

Reflective architectures provide a fundamentally new paradigm for thinking about knowledge-based systems. In a reflective architecture, a knowledge-based system is viewed as incorporating an object-level part and a reflective part. The task of the object-level is to solve problems and return information about an external domain, while the task of the reflective level is to solve problems and return information about the object-level reasoning.

For example, in a reflective architecture one can temporarily associate reflective computation with a program such that during its interpretation some tracing job is performed. Assume a session with a rule-based system has to be traced. The goal is to receive a trace of the rules that are applied. This can be achieved in a language with a reflective architecture by stating a reflective rule such as

```
IF a rule has the highest priority in a situation,
THEN print the rule and the data which match its conditions
```

In a rule-based language that does not incorporate a reflective architecture, the same result can only be achieved either by modifying the interpreter code (so that it prints information about the rules it applies), or by rewriting all the rules so that they print information whenever they are applied. Reflective architectures provide a means to implement reflective activities in a more modular way. It makes the implementation effort easier, and produces more elegant and more adaptable computational systems.

## 5 Examples of reflective architectures

### *5.1 Procedure-based example*

Procedure-based, logic-based, rule-based and object-oriented languages incorporating a reflective architecture can be identified. 3-LISP (Smith, 1982) and BROWN (Friedman and Wand, 1984) are two such procedural examples (variants of LISP). They introduce the concept of a *reflective function*, which is like any other function, except that it specifies reasoning[4] about the currently ongoing reasoning. Reflective functions should be viewed as local (temporary) functions running at the level of the interpreter: they manipulate data representing the current object-level reasoning. Reflective functions take a number of quoted arguments. They have access to two extra variables,

[3] On many points the concept of a reflective architecture runs parallel to the concept of a meta-level architecture. There is often confusion between these two types of architecture. Therefore it is interesting to work out their exact differences.

A programming environment has a *meta-level architecture* if it has an architecture which supports meta-computation, without supporting reflective computation. A meta-level architecture is designed for building systems which reason about other systems. However, it does not make it possible to build systems which reason about themselves.

We purposely use the term programming environment here, instead of programming language, because typically meta-level programming environments present different languages for the object-computation and the meta-computation. These two languages each have their own syntax and interpreter. Some authors call a system where meta-level and object-level have their own special-purpose language a *two-level architecture* as opposed to a meta-level architecture.

The main difference between a meta-level architecture and a reflective architecture is that a meta-level architecture only provides *static access* to the representation of a computational system, while a reflective architecture also provides *dynamic access* to this representation. The result is that a meta-level architecture is more efficient, but less general than a reflective architecture.

We will continue using the term "reasoning", although it might have been more appropriate to speak of "computation", or "deduction" etc., depending on the type of language at hand.

called "env" and "cont", which by default represent the environment (a list of bindings) and the continuation at the time the reflective function is called. A reflective function is able to inspect these (e.g. checking variable bindings) and to modify them (e.g. change the continuation or variable bindings). The env and cont variables are causally connected to the real environment and continuation of the system, in such a way that the results of this reflective reasoning are reflected in the system's future object-level reasoning. Figure 5 shows a very simple reflective program.[5] The code represented conforms with the Common-LISP conventions.

```
(define-REFLECT boundp-else-bind-to-one (symbol &optional env cont)
     (let ((value (binding symbol env)))
       (funcall cont
              (if value
                 value
                 (rebind symbol 1 env)))))
```

**Figure 5** A reflective procedural program

When the function in Fig. 5 is called, for example in

```
(let ((x36))
  (/x(boundp-else-bind-to-one y)))
```

The evaluation returns 36 after reflection (because the symbol y is not bound in this environment). On the other hand, the evaluation of

```
(let (x36)
    (y 12))
  (/x(boundp-else-bind-to-one y)))
```

returns 3.

This example illustrates the architecture for reflection incorporated in languages like 3-LISP and BROWN. Any program can specify reflective reasoning by means of an application of a reflective function. The evaluation of this sub-expression brings the interpretation of the program one level up, to the level where the interpretation of the program was run until that moment. This level reasons about and acts on the environment and continuation of the level below.

The interpreter takes care of the causal connection between the system and the representation it has of itself. Whenever the program specifies reflective reasoning (by calling a reflective function), the interpreter constructs variables denoting the environment and the continuation at that moment of interpretation. The reflective level can manipulate these variables. When the reflective level reactivates the object level, the interpreter continues the reasoning at the level below, after reflecting the bindings of these variables in the actual environment and continuation.

## *5.2 Logic-based example*

FOL (Weyhrauch, 1980) and META-PROLOG (Bowen, 1986) are two examples of logic-based languages with a reflective architecture. These languages adopt the concept of a *meta-theory*. A meta-theory again differs from other theories (or logic programs) in that it is about the reasoning of another theory, instead of about the external problem domain. Examples of predicates used in a meta-theory are "provable(Theory,Goal)", "clause(Left-hand,Right-hand)", etc. Figure 6 presents a small example.

The syntactical conventions adopted are that capitalized symbols represent variables, and commas should be read as "and". The theory called my-theory contains some facts and inference

[5] Note that the purpose of this example is to present the idea behind the design of a procedure-based language with a reflective architecture. Thus it is not written to serve as an example of a programming problem for which a reflective architecture should be used.

```
my-theory: mortal(X) :- human(X).

john's-theory: human(X) :- greek(X).
               greek(socrates).
               P :- reflect(meta-t,P).

meta-t: provable(T,F) :- theorem(T,F).
        provable(T,and(F,G)) :- provable(T,F), provable(T,G).
        provable(T,F) :- clause(T,F,G), provable(T,G).
        provable(T,F) :- clause(my-theory,F,G), provable(T,G), assert(theorem(T,F)).
```

**Figure 6** A reflective logic program

rules that I believe in. The theory called john's-theory contains facts and inference rules John believes in. Meta-t is a meta-theory for john's-theory (or any other theory which models the beliefs of somebody other than myself). The data (or variables) of meta-t are representations of the theorems and clauses of another theory T. This means that the predicates of meta-t range over predicates and clauses of T. Note particularly the final rule of meta-t which states that a clause from my-theory may be used to prove a theorem F in a theory T. If this happens,

theorem(T,F)

is asserted in meta-t. This implements the autoepistemic rule that I may assume that somebody else uses the same inference rules (but not the same facts) as I do, complemented by inference rules specific to him. An attempt to prove in john's-theory

mortal(socrates)

results, when the other clauses have failed to prove the goal, in the exploration of the last clause of john's-theory. This means that the interpreter proceeds with an attempt to prove

reflect(meta-t,mortal(socrates))

Reflect is a special predicate that attempts to prove a theorem in a meta-theory. If the meta-theory succeeds in proving the theorem, this result is reflected in the theory. For the above example, it means that the fact

mortal(socrates)

will be asserted in john's-theory, if the theory meta-t succeeds in proving the theorem

provable(john's-theory,mortal(socrates))

This example illustrates an architecture for reflection in a logic-based language. Programs are represented by means of theories. A theory may have one or more meta-theories which specify reasoning about the theory. The language interpreter has mechanisms that are responsible for the causal connection between a theory and a meta-theory. These mechanisms guarantee the communication of results between the two levels, i.e. they specify how to reflect the results of meta-theory reasoning in object-theory reasoning (and vice versa) and are therefore called reflection principles (Weyhrauch, 1980).

### *5.3 Rule-based example*

TEIRESIAS (Davis, 1982) and also to some degree SOAR (Laird, Rosenbloom and Newell, 1986), are examples of rule-based languages with a reflective architecture. They incorporate the notion of *meta-rules*, which are like normal rules, except that they specify reasoning about the ongoing reasoning. The data-memory these rules operate on represents the object-level reasoning. It contains elements such as "there-is-an-impasse-in-the-inference-process", "there-exists-a-rule-about-the-current-goal", "all-rules-mentioning-the-current-goal-have-been-fired", etc.

Meta-rules are often used to handle the problem of control (i.e. the problem of which rule should be applied next). When an "impasse" in the inference process occurs, the interpreter activates the meta-rules which will try to resolve the impasse.[6] A typical example of an impasse is when there is more than one rule which matches data in the current working-memory. Figure 7 presents an example.

```
WORKING-MEMORY ((o1.True)(o2.True)(o3.True)(p1.)(p3.)(p2.)...)

GOAL: ((p1.True) (p3.True))

RULE MEMORY: (1) IF o1 and o2
                     THEN set(p1,True) and set(p2,False)
             (2) IF o3
                     THEN set(p2,True) and set(p1,False)
             ...

META-RULE MEMORY: (3) IF error-flag-1
                          THEN set(data-elm(o2),False) and set(rule-to-be-fired(2),True)
                  (4) IF satisfied(1) and satisfied(2)
                          THEN set(error-flag-1, True)
```

**Figure 7** A reflective production rule program

The reasoning is in an impasse, because both rule (1) and rule (2) match the current working-memory. The system will try to solve this impasse by initiating a reflective production rule program. The goal of this reflective program is

rule-to-be-fired(?rule) = True

The program in Fig. 7 incorporates meta-rules which might help to solve this impasse. For example the meta-rule

```
IF satisfied(1) and satisfied(2)
THEN set(error-flag-1,True)
```

says that, when both rule 1 and rule 2 can be fired, this is a special event in the object-level inference process (note that rule 1 and 2 propose contradictory actions). Consequently, the data-element error-flag-1 has to be set true. This will enable rule 3 to fire, which modifies the status of the object-level inference process (think of rule 3 as an error-handling procedure).

When the meta-rules have succeeded in solving the "reflective subgoal" of the above example, the reflective reasoning will have affected the object-level reasoning by determining which of the two competing rules will be applied and by modifying the status of the working-memory. This example illustrates the reflective architecture incorporated in systems like SOAR or TEIRESIAS. Note that in this example, reflection is controlled (activated) implicitly as opposed to being programmed explicitly in the procedural and logic-based example.

### *5.4 Object-oriented example*

3-KRS (Maes, 1987a; Maes, 1987b) is an example of an object-oriented language with a reflective

[6] The notion of an "impasse" in the inference process was introduced by SOAR (Laird, Rosenbloom and Newell, 1986). The SOAR interpreter recognizes four specific types of impasses which trigger reflection: a tie, a conflict, a no-change and a rejection. These really represent states in which the inference process gets stuck. It might be a little inappropriate to import this term to discuss reflection in TEIRESIAS. The events that trigger reflective computation in TEIRESIAS are the search for a goal and the making of a conclusion. These events do not really represent a problem situation for the inference process.

architecture. The basic unit of information is the object. Everything in 3-KRS is an object, data as well as programs. 3-KRS introduces a new concept (or programming-construct), this being the notion of a *meta-object*. Meta-objects are like the other objects of the language, except that they represent information about the reasoning performed by other objects and also, are taken into account by the interpreter of the language when running a system.

Every object in the language has a meta-object. A meta-object also has a pointer to its object. An object exclusively represents information about the domain entity that it represents. The meta-object of an object holds all the information that is available about the object itself, i.e. about its implementation and interpretation. It incorporates for example bookkeeping data about the object (such as who created it, when it was created, etc.) and methods specifying how the object should be interpreted (how it inherits information, how the object is printed, how a new instance of the object is made, etc.).

An object may at any point interrupt its object-level reasoning, reflect on itself (as represented in its meta-object) and modify its future behaviour. Reflective reasoning may be guided by the object itself or by the interpreter. An object may cause reflective reasoning by specifying reflective code, i.e. code that mentions its meta-object. The interpreter causes reflective reasoning for an object whenever the interpreter has to perform an operation on the object and the meta-object of the object specifies a special method for this operation. At that moment the interpretation of the object is delegated to this special meta-object.

The example illustrates how locally a non-standard interpreter may be realized. A major advantage of a language with a reflective architecture is that it is open-ended, i.e. that it can be adapted to user-specific needs. But even more, a reflective architecture makes it possible to dynamically build and change interpreters from within the language itself. It allows for example the language to be extended with meaningful constructs without stepping outside the interpreter.

Figure 8 illustrates a very simple example. The 3-KRS language does not support multiple-inheritance. However, if a multiple-inheritance behaviour is needed for some object (or class of objects), it can be realized by a specialized meta-object. The object Mickey-Mouse has a non-standard interpreter which takes care of the multiple-sources inheritance behaviour of this object. The specific strategy for the search of inherited information is implemented explicitly in the language itself by overriding the inheritance-method of the default meta-object.

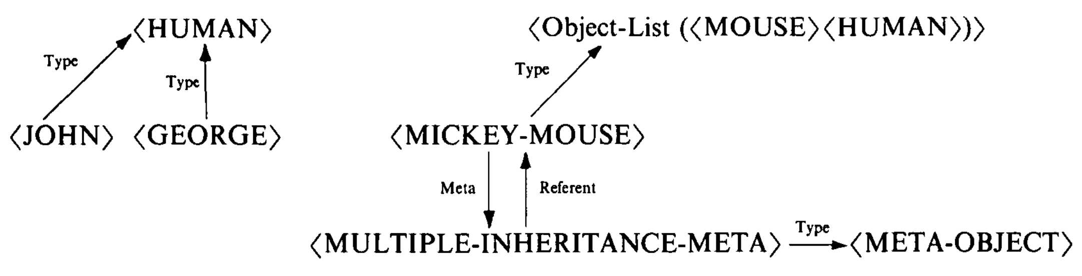

**Figure 8** Implementing a local variation on the language

## 6 Design issues for reflective architectures

### *6.1 Procedural versus declarative reflection*

If we study the reflective architectures discussed above, many common issues can be identified. One such issue is that almost all of these languages operate by means of a *meta-circular interpreter* (FOL is an exception and will be discussed later). A meta-circular interpreter is a representation of the interpreter in the language, which is also actually used to run the language. The interpretation of such a language virtually consists of an infinite tower of circular interpreters interpreting the interpreter below. Technically, this infinity is realized by the presence of a second interpreter (written in another language), which is able to interpret the circular interpreter (and which should be guaranteed to generate the same behaviour as the circular one).

The reason why all these architectures are built in this way is because a meta-circular interpreter presents an easy way to fulfill the causal connection requirement. The self-representation that is given to a system is exactly the meta-circular interpretation-process that is running the system. Since this is a procedural representation of the system, i.e. a representation of the system in terms of the program that implements the system, we say these architectures support *procedural reflection.*

The consistency between the self-representation and the system itself is automatically guaranteed because the self-representation is actually used to implement the system. So there is not really a causal connection problem. There only exists one representation which is both used to implement the system and to reason about the system. Note that a necessary condition for a meta-circular interpreter is that the language provides one common format for programs in the language and data, or more precisely, that programs can be viewed as data-structures of the language.

One problem with procedural reflection is that a self-representation has to serve two purposes. Since it serves as the data for reflective reasoning, it has to be designed in such a way that it provides a good basis on which to reason about the system. But at the same time it is used to implement the system, which means that it has to be effective and efficient. These are often contradicting requirements. Consequently, one has tried to develop a different type of reflective architecture in which the self-representation of the system would not be the implementation of the system. This type of architecture is said to support *declarative reflection* because it makes it possible to develop self-representations merely consisting of statements about the system. These statements could for example say that the reasoning of the system has to fulfill some time or space criteria. The self-representation does not have to be a complete procedural representation of the system, it is more a collection of constraints that the status and behaviour of the system have to fulfill.

The causal connection requirement is more difficult to realize here. It has to be guaranteed that the explicit representation of the system and its implicitly obtained behaviour are consistent with each other i.e. the interpreter itself has to decide how the system can comply with its self-representation. So, in a sense the interpreter has to be more intelligent. It has to find ways of translating the declarative representations about the system into the interpretation-process (the procedural representation) that is implementing the system.

Such an architecture can be viewed as incorporating representations in two different formalisms of one and the same system. During reasoning the most appropriate representation is chosen. The implicit (procedural) representation serves the implementation of the system, while the explicit (declarative) representation serves the reasoning about the system. Although in architectures for declarative reflection more interesting self-representations can be developed, how far such architectures are actually technically realisable is still an open question. GOLUX (Hayes, 1974) and Partial Programs (Genesereth, 1987) are two attempts worth mentioning.

Actually the distinction between declarative reflection and procedural reflection should be viewed as a continuum. A language like FOL (Weyhrauch, 1980) is situated somewhere in the middle: FOL guarantees the accuracy of the self-representation by a technique called *semantic attachment.* The impact of the self-representation is guaranteed by *reflection principles.* It is far less trivial to prove that the combination of these two techniques also succeeds in maintaining the consistency between the self-representation and the system.

## *6.2 Theory relativity of reflection*

Not all reflective architectures that are built provide equal possibilities for reflection. Languages with a reflective architecture give systems the ability to reflect by providing them with a causally connected self-representation they can access and manipulate at run-time. This self-representation determines what reasoning the system is able to perform about itself and what modifications it is able to make to itself. A specific self-representation defines a terminology the system can use to specify reflective reasoning. It fixes the set of terms a system can use, to specify reasoning about and acting upon itself. We call this the *theory relativity* (Smith, 1982) of reflection.

For example, an explicit representation of the LISP interpreter that runs the system might range from

```
(define meta-circular-interpreter-1 (expr &optional env cont)
   (eval expr env cont))
```

to a more extended version

```
(define meta-circular-interpreter-2 (expr &optional (env ()))
   (cond
      ((null expr) NIL)
      ((numberp expr) expr)
      ((eq expr T) expr)
      ((symbolp expr)(binding expr env))
      ((eq (car expr) 'quote) (cadr expr))
      ((primitive-function-p (car expr))
          (apply (car expr)
                 (make-list-of-evaluated-args (cdr expr) env)))
      (T (eval (definition-of (car expr))
               (lexical-make-env
                  (definition-args-of (car expr))
                  (cdr expr)
                  env)))))
```

or to one that makes all machine actions explicit (Sussman, 1982)

```
(defpc meta-circular-interpreter-3
   (save env)
   (save unev)
   (save argl)
   (save fun)
   (save retpc)
   (goto eval-dispatch))

(defpc pop-return
   (restore retpc)
   (restore fun)
   (restore argl)
   (restore unev)
   (restore env)
   (goto (fetch retpc))

(defpc eval-dispatch
   (cond ((self-evaluating? (fetch exp))
           (assign val (fetch exp))
           (goto pop-return))
          ((quoted? (fetch exp))
           (assign val (text-of-quotation (fetch exp)))
           (goto pop-return))
          ((variable? (fetch exp))
           (assign val
                   (get-variable-value (fetch exp)
                                       (fetch env)))
```

(*continued*)

```
   (goto pop-return))
  ((lambda? (fetch exp))
   (assign val
           (make-procedure (fetch exp)
                           (fetch env))
   (goto pop-return))
  ((conditional? (fetch exp))
   (assign unev (clauses (fetch exp)))
   (goto evcond-pred))
  ((no-operands? (fetch exp))
   (assign exp (operator (fetch exp)))
   (assign retpc apply-no-args)
   (goto meta-circular-interpreter-3))
  ((appliciation? (fetch exp))
   (assign unev (operands (fetch exp)))
   (assign exp (operator (fetch exp)))
   (assign retpc eval-args)
   (goto meta-circular-interpreter-3))
  (t (error "unknown expression type -- eval"))))
```

The terminology introduced by meta-circular-interpreter-1 is rather limited. It makes it possible to refer to four aspects of the interpretation of a LISP program: the interpreter program, the program itself, the environment and the continuation. A system which has this interpreter as a representation of itself, may for example refer to its own environment, or it may act on its own interpretation, as in

```
(define variant-on-meta-circular-interpreter-1 (expr &optional env cont)
   (do-something-with-the-result
         (eval (do-something-with-the-input expr))
              env)
              cont)))
```

This is a variant on the interpreter, which will do something before and after the (default) evaluation of an expression. The self-representation of meta-circular-interpreter-1 only makes it possible for a system to reason about and act on the LISP-interpreter as a whole. It does not make it possible to analyze how the interpreter works, or to modify internal aspects of the interpreter.

On the other hand, meta-circular-interpreter-2 introduces a more sophisticated terminology. This self-representation makes some of the internal aspects of the LISP-interpreter explicit. It makes it possible, for example, for a system to ask questions about the evaluation of an expression of a particular syntactical type. It also allows more finely grained modifications to the LISP-interpreter, such as

```
(define variant-on-meta-circular-interpreter-2 (expr &optional (env ()))
  (cond
    ((null expr) NIL)
    ((numberp expr) expr)
    ((eq expr T) expr)
    ((symbolp expr)(binding expr env))
    ((eq (car expr) 'quote) (cadr expr))
    ((primitive-function-p (car expr))
       (apply (car expr)
              (make-list-of-evaluated-args (cdr expr) env)))
(T (eval (definition-of (car expr))
         (dynamic-make-env
            (definition-args-of (car expr))
            (cdr expr)
            env)))))
```

This variant modifies the interpreter so that it has dynamical scoping. In the original interpreter a function-definition was interpreted in the environment containing only the bindings made by the arguments of the function-call. The function lexical-make-env used in the original interpreter was defined as follows

```
(define lexical-make-env (definition-args formula-args old-env)
   (cond ((null definition-args) nil)
         (T (add-binding (car definition-args)
                         (eval (car formula-args) old-env)
                         (lexical-make-env (cdr definition-args)
                                           (cdr formula-args)
                                           old-env)))))
```

In the variant interpreter, function-definitions are evaluated in the environment at the moment they are called, extended with the bindings made by the arguments of the function-call. The function dynamic-make-env adopted in this variant interpreter is defined as

```
(define dynamic-make-env (definition-args formula-args old-env)
   (cond ((null definition-args) old-env)
         (T (add-binding (car definition-args)
                         (eval (car formula-args) old-env)
                         (dynamic-make-env (cdr definition-args)
                                           (cdr formula-args)
                                           old-env)))))
```

Note that this variation cannot be made by means of meta-circular-interpreter-1. However, meta-circular-interpreter-2 does not make explicit the continuation of the evaluation (which is implicit in the recursion) as meta-circular-interpreter-1 does. Consequently, some variations on meta-circular-interpreter-1 will not be expressible in the terminology introduced by meta-circular-interpreter-2. Meta-circular-interpreter-3, on the other hand, presents a self-representation of the operational aspect of systems in terms of machine concepts such as registers and memory-adresses. This makes it suitable, for example, for analyzing and modifying the time and space requirements of systems. However, it leaves implicit other aspects of the interpreter, such as its recursive behaviour.

## *6.3 Implicit versus explicit reflection*

When designing a reflective architecture, another design choice to be made is how the reflective reasoning will be programmed, i.e. what the conditions are under which a system will reflect and what kind of reflective reasoning the system will perform when these conditions are fulfilled. Two types of architectures can be identified.

In an architecture for *implicit reflection*, reflective reasoning plays a crucial role. The reflective level is systematically activated by the interpreter of the language. This means that there are "holes" in the normal interpretation process that remain to be filled by reflective reasoning. If the interpretation process consists of the execution of tasks $a_1 \ldots a_n$, there exist one or more i, for which the action $a_i$ is not defined in the interpreter, i.e. some $a_i$ are necessarily a piece of reflective reasoning defined by the program of the system that is interpreted.

TEIRESIAS (Davis, 1982) is an example of a language with an architecture for implicit reflection. TEIRESIAS systematically reflects every time a new goal has to be pursued by the interpreter. The purpose of this reflective reasoning is to compute the domain-dependent strategy to be used for the exploration of the search space for the goal. SOAR (Laird, Rosenbloom and Newell, 1986) is another example of an architecture for implicit reflection. Although SOAR is also a rule-based system, the reflective reasoning in SOAR plays a different role in the interpreter. A knowledge-based system in SOAR reflects when the object-level reasoning gets stuck in an impasse. The goal of the reflective reasoning that is activated at that moment, is to resolve the impasse. As soon as this is achieved, the reflective level is abandoned and the object-level reasoning is continued.

In architectures for *explicit reflection*, reflective reasoning does not happen automatically. Reflection only takes place when the program of a system explicitly prescribes it. Reasoning normally takes place at the object level. Whenever the program prescribes reflective reasoning, the system enters a break to temporarily do something at the reflective level. Most languages which were initially designed without a reflective architecture, provide an architecture for explicit reflection. The reflective reasoning is not a crucial component of the interpretation process.

3-LISP is an example of an architecture for explicit reflection. Reflective reasoning occurs in 3-LISP whenever a reflective lambda-expression is evaluated. A system in 3-LISP may not perform any reflective reasoning at all. Another example of an architecture for explicit reflection is FOL. The FOL interpreter only jumps to a reflective level when the programmer explicitly states this. The 3-KRS language supports explicit as well as implicit reflection.

## 7 Reflections on past and future work in the field

In the past, reflection has often been put forward as a fascinating, rather philosophical issue, though one without technical value. Enormous advances have since been made, such that the state of the art can be characterized as follows:

- Reflection can be defined in a clear and technical way
- Reflection is recognized as being useful in real world programming, particularly in knowledge based systems
- A language with a reflective architecture provides better support for programming reflective computation
- The following are the critical issues in constructing a reflective architecture: the self-representation, programming reflective computation and the causal connection requirement
- A reflective architecture is necessarily limited. The limitations, but also the capabilities, are determined by design choices on these critical issues

This is only the beginning of a more systematic exploration of computational reflection. Some of the remaining problems that can (and are) now be tackled are:

- The development of representations of computational systems which are interesting from a reflective point of view (c.f. problem of declarative reflection)
- How to avoid negative uses of reflection, i.e. what are the interesting and safe uses of reflection
- How to implement reflective architectures efficiently
- Developing libraries of useful reflective abstractions (c.f. IF and WHILE, etc. as abstractions for programming the flow of control).

We hope that this paper will inspire the reader to reflect on these problems.

## 8 Conclusions

This paper has brought some perspective to various issues in computational reflection. A definition of computational reflection was presented, its importance was discussed and the design and construction of architectures that support reflection were studied. Illustrations were drawn from different sorts of languages including procedural, logic-based, rule-based and object-oriented languages.

Many details of reflection and reflective architectures have not been covered by this paper. The interested reader may consult (Maes, 1987a) or (Maes and Nardi, 1988).

## Acknowledgements

I am very grateful to Luc Steels, who supervised this research and when necessary corrected its direction.

This research was supported by ESPRIT project number 440: "Message Passing Architectures and Description Systems" and COST-13 project number 7: "Advanced Issues in Knowledge Representation". The author is a senior research assistant of the Belgian National Fund for Scientific Research.

## References

Bowen, K, 1986. "Meta-level techniques in logic programming", in: *Proceedings of the International Conference on Artificial Intelligence and its Applications*, Singapore.

Clayton, B, 1985. "ART: Programming primer", Inference Corporation, Los Angeles.

Davis, R, 1982. In: *Knowledge-Based Systems in Artificial Intelligence*, Davis R and Lenat D, McGraw-Hill, New York.

Friedman, D and Wand, M, 1984. "Reification: Reflection without meta-physics", *Communications of the ACM* **8**.

Genesereth, M, 1988. "Prescriptive introspection", in: *Meta-Level Architectures and Reflection*, Maes P and Nardi D (eds.), North-Holland Publishers, Amsterdam.

Hayes, P, 1974. "The Language GOLUX", University of Essex Report, Essex, United Kingdom.

IntelliCorp TM, 1985. "KEE TM. Software development system. User's manual", SEE Version 2.0, (Symbolics, LMI, Explorer), IntelliCorp.

Laird, J, Rosenbloom P and Newell A, 1986. "Chunking in SOAR: The anatomy of a general learning mechanism", in: *Machine Intelligence* **1**(1), Kluwer Academic Publishers.

Maes, P, 1987a. "Computational reflection", PhD thesis. T.R. 87-2, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels, Belgium.

Maes, P, 1987b. "Concepts and experiments in computational reflection", in: *OOPSLA-87 Proceedings*, Florida.

Maes, P and Nardi, D (Eds.), 1988. "Meta-level architectures and reflection", North-Holland Publishers, Amsterdam.

Minsky, M, 1974. "A framework for representing knowledge", AI-MEMO 306, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts.

Roberts and Goldstein, 1977. "The FRL Primer", AI-MEMO 408, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts.

Smith, B C, 1982. "Reflection and semantics in a procedural language", Technical Report 272, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, Massachusetts.

Smith, B C, 1986. "Varieties of self-reference", in: *Theoretical Aspects of Reasoning about Knowledge. Proceedings of the 1986 Conference*. Halpern J. (Ed.), Morgan Kaufmann.

Steels, L, 1985. Lecture Notes on AI, Internal report, AI-LAB, VUB, Brussels.

Steels, L, 1987. "The deepening of expert systems", *AI-communication* **0** (1), North-Holland, Amsterdam.

Sussman, G, 1982. "Implementing LISP", in: *Functional Programming and its Applications, an Advanced Course*, J Darlington, P Henderson and D A Turner (Eds.), Cambridge Universiry Press, London.

Van Melle, W, 1980. "System aids in constructing consultation programs", *UMI Research Press*, Ann Harbor, Michigan.

Weinreb, D and Moon, D, 1981. "LISP machine manual", *Symbolics Inc*, Cambridge, Massachusetts.

Weyhrauch, R, 1980. "Prolegomena to a theory of mechanized formal reasoning", *Artificial Intelligence* **13** (1), (2). Amsterdam, North-Holland, The Netherlands.