Proc. of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI. pp. 29–38. August 1995

Reflection in logic, functional and object-oriented programming: a Short Comparative Study

François-Nicola Demers and Jacques Malenfant*
Département d'informatique et de recherche opérationnelle
Université de Montréal, Montréal, Québec, CANADA

Abstract

Reflection is a wide-ranging concept that has been studied independently in many different areas of science in general, and computer science in particular. Even in the sub-area of programming languages, it has been applied to different paradigms, especially the logic, functional and object-oriented ones. Partly because of different past influences, but also because researchers in these communities scarcely talk to each others, concepts have evolved separately, sometimes to the point where it is hard for people in one community to recognize similarities in the work of others, not to speak about cross-fertilization among them. In this paper, we propose a synthesis covering mainly the application of computation reflection to programming languages. We compare the different approaches and try to identify similar concepts hidden behind different names or constructs. We also point out the different emphasis that has been given to different concepts in each of them. We do not claim neither completeness nor closeness in our treatment. We rather aim at building bridges between programming languages communities, and address other aspects when they appear relevant.

1 Introduction

Reflection has long been studied in philosophy and formalized to some extent in logic [Fef62]. It arised naturally in artificial intelligence, where it is intimately linked to the end goal itself: reflection is viewed as the emergent property responsible, at least in part, for what is considered an "intelligent behavior". Perhaps surprisingly, it has been also applied in the area of programming languages under the name of *computational reflection*. Computational reflection dates from Brian Smith's seminal work in the early 80s [Smi82, Smi84]. In his way to formalize the concept of reflection, he developed two languages: 2-Lisp and 3-Lisp. Although Smith gave much more attention to the representation relation he called ϕ that a computational process bears to its subject domain, his work quickly became famous in the functional community for the ability 3-Lisp was giving to a program to reflect about its own computation using *reflective towers*.

This approach inspired much work in the functional programming community until the late 80s, probably because reflection was growing in a fertile ground. Thanks to its quote construct, Lisp is famous for its metalinguistic power. It allowed from its very beginning the manipulation and execution of program fragments, which are a primitive manifestation of reflective concepts. Lisp metacircular interpreters were also widely explored and debated. The Lisp community therefore had a long experience with the kind of things reflection wanted to enable.

By the beginning of the 90s, it was becoming quite clear that in order to master the inherent complexity of a fully reflective programming language, structuring mechanisms were badly needed. The object-oriented paradigm imposed itself to take up the challenge. This trend was not completely innocent. A whole subset of the object-oriented community has been profoundly influenced by Lisp, especially the one revolving around Smalltalk and naturally, around object-oriented extensions of Lisp itself (the quintessence of which being the Common Lisp Object System, CLOS). In fact, some of the proeminent persons in the OO community are the same that were very influential in the Lisp one a decade before. But, the implementation of the original concepts in this new paradigm prove to be different enough to make it difficult to relate them to the functional ones. Unsurprisingly, Smalltalk is one of the most reflective language, with its "object-to-the-bottom" principle, unfortunately mostly ignoring itself as such. The Metaobject Protocol of CLOS is the most achieved reflective programming language and system to date.

Quite independently, the logic programming community become interested in concepts near to the reflective ones, especially metaprogramming. As Lisp, Prolog included from its very first implementations mechanisms for the manipulation of programs at run-time, therefore enabling metaprogramming: metavariables are used to transform a term into a goal and thus play the role of apply, clauses are accessible through the system predicate `clause/2` (not to speak about the debated `assert/1` and `retract/1`), which returns them in the form of terms, themselves inspectable through the system predicates `=../2`, `arg/3` and `functor/3`. Metaprogramming quickly became so common in Prolog that any self-respecting textbook includes the so-called vanilla meta-interpreter for Prolog written in three clauses (see, for example, [SS86]). A series of conferences devoted exclusively to metaprogramming in logic has also been organized [AR89, Bru90, Pet92, MET94]. Finally, the new language Gödel [HL94] has been designed with the aim of making metaprogramming more declarative, by implementing primitively the manipulation of terms in a ground representation (see §3.2.1).

The development of reflection independently in three different programming paradigms makes it difficult to understand the relationship between similar concepts expressed in different ways from community to community, and even from language to language. The goal of the present paper is to compare and contrast some of the reflective languages and

*Authors' current address : C.P. 6128, Succursale Centre-ville, Montréal, Québec, Canada H3C 3J7, phone : (514) 343-7479, fax : (514) 343-5834, e-mail : {demers,malenfan}@iro.umontreal.ca This research has been supported by FCAR-Québec and NSERC-Canada.

more generally results coming from the three communities. We do not claim to address all aspects of such a comparison in this short paper, but rather to build bridges between communities by sharing our experience of being involved in the three for almost eight years. We don't concentrate on computational reflection in programming languages either, but we rather address some other work relevant to our goal.

The rest of the paper is organized as follows. The next section introduces the terminology and concepts of computational reflection. The Section 3 presents the design of several reflective languages in the three paradigms of logic, functional and object-oriented programming. The Section 4 discusses some formal notions associated to computational reflection, such as the notion of reflection coming from logic as well as some attempts to give a formal semantics of reflective programming languages. We then draw some conclusions.

2 Terminology and basic concepts

For a long time, terminology and basic concepts have been the Achille's heel of computational reflection. As noted earlier, the actual implementation of reflection in programming languages uses a lot of techniques and mechanisms that existed a long time before reflection: the ability to manipulate programs as run-time data, the ability to inspect data structures at run-time (such as terms in Prolog), first-class entities, metaprogramming, etc. In the past years, we have seen a tendency to mix reflection and the use of these techniques. In a sense, the tools were taken for the concept. A link certainly exists between first-class entites, higher-order functions, metaprogramming and reflection, but no one stands for the other.

Reflection has been defined in a fairly general way by Brian Smith during the ECOOP/OOPSLA'90 workshop on reflection [WRM90]:

> "An entity's integral ability to represent, operate on, and otherwise deal with its self in the same way that it represents, operates on and deals with its primary subject matter."

In programming languages, the incarnation of this definition appears as follows [BGW93]:

> "Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation : *introspection* and *intercession*. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called *reification*."

Reification must be distinguished from first-class entities. When an entity is first-class in a language, it means that this entity can be created at run-time, passed as actual parameters to procedures, returned as result of a function and stored in a variable. All what is needed is to provide programs with values of a suitable abstract data type defined by the language. In functional languages, functions are first-class for example. It means that functions (actually closures) are values that can be manipulated as any other values in the language. Reification goes well beyond by imposing that we have complete control over this value in the language itself. In particular, we should be able to modify its representation using the language itself. We should also be able to inspect it at run-time. Making an entity first-class is strictly included in its reification, but not the opposite.

Coming back to reflection, Pattie Maes gave in her thesis [Mae87] the following definitions that summarize pretty well the notion of reflection in computational systems:

- A **computational system** is something that **reasons** about and **acts** upon some part of the world, called the **domain** of the system.
- A computational system can be **causally connected** to its domain. This means that the system and its domain are linked in such a way that if one of the two changes, this leads to an effect upon the other.
- A **meta-system** is a computational system that has as its domain another computational system, called its **object-system**. [...] A meta-system has a representation of its object-system in its data. Its program specifies **meta-computation** about the object-system and is therefore called a **meta-program**.
- **Reflection** is the process of reasoning about and/or acting upon oneself.
- A **reflective system** is a causally connected meta-system that has as object-system itself. [...] When a system is reasoning or acting upon itself, we speak of **reflective computation**.

Although meta-interpreters have been used intensively in the implementation of reflective systems, metaprogramming is not reflection. But note that, in reflection, accessing the metalevel representation is done on purpose, to perform some task. The metaprogramming approach gives the programmer access to this representation because he has his hand on the meta-interpreter, which he can change to perform the task he was originally intending to do. Reflection goes the other way around. An implementation pre-exist to which we have no access, except from the program itself, by reflection. Metaprogramming played an important role in the implementation of reflective languages because, in practice, it is difficult to access the metalevel information in an existing language, where typically the implementation is completely sealed. A meta-interpreter is interesting because it is much easier to make its data available to the programs it runs than modifying the language's implementation.

Maes said [Mae87]: "The main difference between a metalevel architecture and a reflective architecture is that a metalevel architecture only provides *static access* to the representation of the computational system, while a reflective architecture also provides a *dynamic access* to this representation." The kind of change we can do in the metaprogramming approach is to modify the meta-interpreter prior the execution of the program. Reflection on the other hand allows the program to change its behavior while running, depending upon its current execution (such as the inputs and intermediate results).

In computational reflection, a distinction is traditionally made between **structural** and **behavioral reflection**. The two concepts are defined as follows:

- **structural reflection** implies the ability of the language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types[1];

[1] e.g., in a language providing lists as ADT, structural reflection calls for providing the programs with a complete reification of the implementation of the list ADT, whose internal representation or operations could then be modified.

- **behavioral reflection** implies the ability of the language to provide a complete reification of its own semantics (processor) as well as a complete reification of the data it uses to execute the current program.

The distinction has been made mainly because it is much simpler to implement structural reflection efficiently than behavioral reflection. For instance, languages such as Lisp and Prolog have included some structural reflection features for years. Accordingly, behavioral reflection is much less spreaded, but it is the main focus of today and future research.

3 Design of reflective programming languages

3.1 Functional programming

3.1.1 Lisp quote and its rationalization

Lisp has always enjoyed a powerful metalinguistic facility in its quoting mechanism. Quote introduces a simple metalevel designation for reified expressions, by using the data representation of that expression as designator. Moreover, the data representation "returned" by quote is the quoted object itself, and most of the time a list, the fundamental ADT of Lisp, which makes it inspectable and mutable at run-time using the usual list primitives. Quote is not only responsible for the ability of a Lisp program to manipulate expressions as data, it also make sure the data can be executed after a quick decoding making a true Lisp expression out of its quoted representation. This decoding is automatically done by the *eval* function. But, this behavior is considered anomalous [Mul92], and indeed it makes it difficult to manipulate the reified expressions.

When Smith invented computational reflection and implemented it in Lisp, he was aware of this problem. He first proposed 2-Lisp to address issues related to structural reflection and the representation relationship between a computational process and its subject domain. Crucial to 2-Lisp is a rationalization that clearly distinguishes levels of designation between the metalevel data representation and the base level code representation. In 2-Lisp, no automatic decoding appears in the evaluation process. Instead, two primitives, UP and DOWN, help to mediate the metastructural hierarchy, and there is no other mean to remove quotes. More recently, Muller [Mul92] proposed M-Lisp with a similar motivation. M-Lisp is claimed to be a simpler, or rationalized, 2-Lisp.

3.1.2 3-Lisp and behavioral reflection

Perhaps because the 2-Lisp solution to self-reference was too complicated (according to Muller), Smith's work is much more famous for his second language 3-Lisp implemented in collaboration with des Rivières [dRS84], which introduced computational reflection. The goal of 3-Lisp is to give programs the ability to reason about and modify their own computation represented by an expression, an environment and a continuation. The reflective system proposed by 3-Lisp is described as follows:

- a *reflective tower* is constructed by stacking a virtually infinite number of meta-circular interpreters, each one executing the one under itself and the bottom one (level 1) executing the end-user program (level 0).
- reflective computations are initiated by calling *reflective procedures*, which are procedures of three parameters; upon invocation, the reflective procedure is passed the argument structure of the current expression (its own invocation), the current environment and the current continuation.

The reflective tower is needed to overcome the well-known problem of introspective overlap: if a reflective procedure is passed the current environment and continuation, its own execution modifies these data. The 3-Lisp solution is to execute the reflective procedure not as the code in the program but as code within the interpreter itself. A reflective procedure `p` invoked at level n is thus executed on the current arguments, environment and continuation of level n (which are manipulated by level $n+1$'s meta-interpreter) by the meta-interpreter of level $n+2$. Since a reflective procedure can call another reflective procedure, we potentially need an infinite number of levels in the reflective tower. In practice, in the same way there exists well-formed recursions whose individual execution always need a finite number of recursive calls, a well-formed reflective program will need a finite number of tower levels for each of its runs.

It is worth notice that Smith called 3-Lisp a procedurally reflective language. In Smith's mind, the word procedural reflection stands for systems where there is a strict implementation relationship between the object system and its metalevel. The approach is not the only possible one for reflection in general, but it is the one of choice for most reflective languages. When the metalevel data structures are the same that are actually used to run the object level, the causal connection essential to reflection comes for "free" since the object level and the metalevel representation evolve in perfect synchronization. Another important aspect of 3-Lisp is that an implementation has been given that avoids the actual levels of meta-intepretation by using a *shifting-level* processor [dRS84]. This implementation, although extremely operational (the words are Friedman's and Wand's ones [WF86]), was a first non-reflective description of the reflective tower.

3.1.3 3-Lisp followup

In the first of a series of three papers [FW84, WF86, WF88], Friedman and Wand undertook the task of implementing the kind of behavioral reflection offered by 3-Lisp but without resorting to a reflective tower. This idea led to a mini-language called Brown featuring reflective procedures, through which we can access a reification of the program's run-time data structure (expression, environment and continuation) as well as the inverse operation now called reflection which reinstalls reified data structures into the interpreter. On the other hand, Brown (and its successors) completely avoided the representation and designation issues raised by 2-Lisp.

Brown84 turned out to implement a subset of the reflective capability exhibited by 3-Lisp that didn't need to resort to towers at all. Challenged by Smith to extend their techniques to the tower itself, Friedman and Wand proposed in the second and third paper a denotational account of reflective towers using *metacontinuations*. One of the salient feature of this new Brown86 is that it models the reflective tower with only one unique interpreter active at any time (a property that Danvy and Malmkjaer called *single-threadedness* [DM88]). In this context, a metacontinuation can be thought as a list of continuations, each one representating the state of an interpreter above the one currently active. Brown86 defines *reification* the act of calling a reflective procedures that receives a reification of the current arguments, environment and continuation as actuals. It defines *reflection* as the act of reinstalling an expression, an environment and a continuation in the interpreter one level below in the tower. When a piece of code reflect, a new in-

terpreter is spawn with the given initial state and the state of the one currently executing is pushed into the metacontinuation. When the interpreter below invokes a reflective procedure, a thunk is built from the reflective procedures and is passed to the metacontinuation to be run one level higher in the tower. On the other hand, if the interpreter below returns, the first continuation in the metacontinuation is restored so that the intepreter above is restarted at the point it stopped when the lower one was spawned.

The Brown experience gave the whole reflection community much insights into computational reflection, but it actually failed to reach its goal to give a denotational account of reflection in a way we will explain in Section 4. After Friedman and Wand, Danvy and Malmkjaer pursued the work and proposed the language Blond [DM88]. Blond is similar to Brown except in some specific aspects. Brown turned out to be flawed in the treatment of environments, a problem solved in Blond. Also, Blond made a distinction between what it calls pushy and jumpy continuations in the management of metacontinuations. Three other languages contributed to this school of reflection: Stepper [Baw88], and more recently I_R [JF92] and Black [AMY93].

3.2 Logic programming

All the different issues mentionned earlier (esp. in 3.1) have also been studied in logic programming but under different names. We look at them (and also new one) in the following.

In logic programming, structural reflection has been the first one to be introduced. The main concern at that time was to facilitate the problem of managing system's database of clauses. The predicates `clause/1`, `assert/1`, `retract/1` and `call/1` are good examples of procedures used in Prolog for structural reflection because with them, goals and clauses are treated as first-class objects represented by terms. Computational reflection has been studied to respect the conceptual basis of logic programming that is logic programs are theories and execution is deduction. The need is for an ability to explicitly refer to theories (leading to structural reflection) and to "discuss" derivability from their theories (leading to computational reflection). The main approach was then to construct a system which amalgamates an object-level logic system L with a metalanguage M suitable for formalizing the derivability relation of the original object language system. The resulting system is more expressive and problem-solving power than the original object language system alone. As said in [Bow82] (and similarly in [Wey80]), the amalgamation of languages L and M consists of L and M together with

1. a **naming relation** (similar to the quote mechanism of functional programming) which associates with every linguisting expression of L at least one variable-free term of M. This relation constitutes the causal connection needed for reflective systems.
2. a representation of the derivability relation $\vdash_L$ by means of a predicate symbol `Demo` in the context of sentences Th of M.
3. the following rules called **reflection principles** [Fef62] [Bow82] (sometimes called **downward** and **upward reflection** [LMN91] respectively) given by:

$$\frac{Th \vdash_M \texttt{Demo}(A', B')}{A \vdash_L B} \quad \text{and} \quad \frac{A \vdash_L B}{Th \vdash_M \texttt{Demo}(A', B')}$$

A reflective system which respects these particularities is said to be an **amalgamating language** [Bow82] [Dem94]. Some reflective systems have been created in a way that they could be called amalgamating systems: metaProlog [Bow85], Reflective Prolog [CL89] and the different 3-Prolog languages [Dem94][2]. These systems are concerned mainly with the representation of the predicate `Demo` presented earlier.

However all these amalgamating systems are relatively inefficient (because of the meta-programming techniques they use). For this reason, some people have tried to create non-amalgamating ones. Although the latters are all much less powerful than the preceding ones because the predicate `Demo` is not entirely explicitly represented (by Th), they integrate reflective mechanisms useful for certains logical applications. Examples of the non-amalgamating systems are FOL [Wey80], R-Prolog* [Sug90], CPU [LMN91] and ALPES-IProlog [BCL+88]. Most of these systems have tried especially to resolve practically and theoritically problems related to the logical representation of goals, clauses and variables.

3.2.1 Quote mechanism

As we said earlier, in reflective systems a quote mechanism is essential. But this naming relation has been implemented in different ways and with different power. The most natural implementation is to represent all reflective objects as logical terms in the language. This technique is used in most Prolog implementation, and it expects object-level variables that are represented by meta-level variables. This representation is said to be non-ground. In the language Gödel [HL94], a ground representation (object-level variables are represented by ground logical terms) is preferred to make the quote mechanism completely uniform (all meta-objects are terms) and most powerful. This quote mechanism is used in the reflective systems metaProlog of Bowen (and a similar language G_{CP} [Chr90] which is not said to be reflective even if it has some reflective features) and in some of the set of 3-Prolog languages. However the ground representation is not efficient (not until now maybe in a near future [HL94] by using efficient concurrency and partial evaluation [Gal93]) because in the `Demo` definition, all the substitution procedure must be simulated, a process which is very cosuming in time and space. For this reason, most reflective languages uses a quote mechanism less powerful but much more efficient (for example by using non-ground representation). This is the case of the systems R-Prolog* (it uses lists as meta-level representation for reflective objects), Reflective Prolog, CPU, ALPES-IProlog and some simple 3-Prolog languages.

In some languages, the quote mechanism is also used to create an explicit distinction (as two types) between object-level terms and meta-level terms. This is the case for the languages R-Prolog*, metaProlog and some 3-Prolog languages. In metaProlog, the distinction is made in a similar way than in Gödel (where all terms are statically typed) by considering the term used for representing object-level variables as special and unique.

3.2.2 Reflective mechanism

It has to be noted that all but one reflective languages use **explicit reflective mechanism** with reflective procedures. These procedures (called **reflective predicates**

[2] In this paper, we have written several different reflective languages with different degrees of reification: 3-Prolog$_P$ reifies the object-level program clauses (and possibly `Demo`), 3-Prolog$_U$ reifies the substitutions made by unification, 3-Prolog$_R$ reifies the resolvent and 3-Prolog* reifies all the different previous structures and implements an infinite reflective tower as in 3-Lisp.

in R-Prolog* and 3-Prolog) could be specified explicitly in the program (by goals). Or some predicates are said to be reflective because they contain in their definition some explicit use of upward and downward reflection which are specific predicates creating causal connection and access to reflective structures. This is the case of the languages CPU and ALPES-IProlog. Only one language (Reflective Prolog [CL89]) uses an **implicit reflective mechanism** which is made by defining reflective operations as metalevel definitions and taking care of reflective calls by an extended resolution procedure.

3.2.3 Reification and reflection

The last issue most important for reflective languages is causal connection. In logic programming, this mechanism is implemented in two different ways: by using special predicates or by using specific parameters to reflective predicates. The former creates an **indirect causal connection** and the latter, a **direct** one. An indirect causal connection is a transfert in two directions (usually up and down) of reflective information made at different time of execution. A direct one is rather about at the same moment. In most logical reflective languages, indirect causal connection is used because it is easier to implement. For example, in CPU [LMN91], the reflection mechanism triggers the computation from the object-level to the meta-level domain by **upward reflection** (a predicate `reflect_up`) and vice versa **downward reflection** (a predicate `reflect_down`). These two mechanisms are not executed at the same time as they should be done virtually: upward reflection is made first (to "read" reflective structures) and later in the execution, downward reflection is made (to "write" them back). Comparable indirect causal connection mechanisms are used in metaProlog, ALPES-IProlog and Reflective Prolog. In one way indirect causal connection could be compared with reflection and reification mechanisms used in functional programming.

The direct causal connection is used in R-Prolog* and the different languages 3-Prolog. It works as follows: reflective predicates receive additional parameters containing the different reflective information (down) and other additional parameters (up) which represent reflective information after modification (by reflective predicates execution).

3.3 Object-oriented programming

Two major trends single out in the history of object-oriented programming. The first, examplified by Simula-67, C++, Eiffel and Beta, is shaped by software engineering and modularity principles. It led to typed languages and is characterized by a relatively rigid object model alien to reflective concepts. The second trend, born in the world of untyped and already partly structurally reflective languages. It led to much more flexible languages where classes are typically treated as first-class entities. This trend is examplified by Smalltalk as well as object-oriented extensions of Lisp: Flavors, Loops, Ceyx, and CLOS. This second trend is playing a major role in reflection.

3.3.1 Structural reflection in OOP

The evolution of flexible object-oriented languages has been first marked by a quest for the right metaclass/class/instance model that led to a full notion of structural reflection in what concerns the object model of the language. In summary, the history stands as follows. Smalltalk-72 already included the idea that everything should be an object, even a class. The class introduces structural reflection in the sense that it describes the structure of an object. Smalltalk-72 made classes first-class entities, but yet they were considered as instances of themselves. Besides including inheritance, Smalltalk-76 introduced metaclasses, classes of classes. Note that in OOP, each object has a class, which is called its instantiation class. As an object, the Smalltalk-76 class has its own instantiation class called its metaclass. Smalltalk-80 pursued this idea, but imposed some important limits on metaclasses [Coi87]:

- metaclasses do not have names and they are treated by the system in such a way to hide them as much as possible from the end-user.
- all metaclasses are instances of the same class `Metaclass`, hence the number of metalevels is fixed and metalinks cannot be created indefinitely

Loops [BKK+86] propose a meta-architecture similar to the one of Smalltalk, except that it allows metaclasses to be created explicitly. On the other hand, metaclasses are still distinct from classes and a maximum number of metalinks is still fixed. The ObjVLisp model [Coi87] has been proposed to unify metaclasses and classes and to allow an indefinite number of metalinks to be created. After all, the only difference between classes and metaclasses is that the latter's instances can themselves have instances (i.e. they are classes). The ObjVlisp model is minimal, since it is based on two classes only, `Object` and `Class`. It also solves the potential infinite metaregression by making `Class` its own instance. ObjVLisp is a model that has been applied first to an object system written in Lisp, but it has also been applied to Scheme, to Smalltalk (Classtalk [BC89]), and to an object-oriented extension of Prolog (ObjVProlog [MLV89, MLV90]).

3.3.2 Towards full reflection

Unsurprisingly, Smalltalk is one of the most reflective language to date, unfortunately mostly ignoring itself as such. With its "object-to-the-bottom" principle, Smalltalk pionnered a design principle basic to reflection, which is that everything should be expressed in the terms of the language itself. On the other hand, Smalltalk was not designed with the aim of making this organization so manifest in the language to incite people to use and modify it, a goal pursued by reflection. For reflection practionners, it became obvious that a static description of the language in terms of classes and objects, although necessary to properly organize things, was not enough to enable a full fledged reflective programming paradigm. We need a manifest description of the protocols activating the different objects in order to execute the program. The Metaobject Protocol of CLOS [KRB91, GWB91] tries to achieve exactly that. It does not only describe the objects involved in the representation of the computational process, it also exhibits the protocols responsible for the actual execution of the program.

3.3.3 Behavioral reflection in OOP

As we have seen, a behavioral reflection model must allow programs to intervene in the current execution in order to execute some reflective code that will analyse or modify the course of events. In functional programming, 3-Lisp invented reflective procedures for that. The vantage points where a program can reflect in 3-Lisp is at procedure invocation. In OOP, procedure invocation is traded for message passing. Hence, it is natural to use message passing as vantage points where to reflect, and to use methods to represent the reflective code. The execution of a message is divided in two

phases: a lookup phase to find the method that corresponds to the message's selector, and the apply phase which actually executes that method. Behavioral reflection in OOP has been implemented mainly by giving the user the control over these message passing mechanisms. This has been done in two different ways:

- In traditional OOP languages such as Smalltalk, the message passing mechanisms has been reified as: a lookup and an apply method. These methods are typically held by a meta-object to which is linked the object. Every message sent to an object `o` is then transformed into a sequence of two messages: a lookup message sent to the meta-object of `o` that yields a method to which an apply message is sent to invoke it. Reflective code can be implemented by creating subclasses of standard meta-objects and redefining the lookup method and by creating subclasses of the standard class describing methods and redefining the apply method.
- In OOP languages based on the notion of generic function, such as CLOS, behavioral reflection has been introduced by reifying the generic function invocation protocol, making `apply-generic-function` itself a generic function. Reflective code can be implemented by defining subclasses of the standard class describing generic functions and by redefining the method `apply-generic-function` for these new classes.

3.4 Comparison

In the following, we initiate a comparison between the current application of reflection to functional, logic and object-oriented programming languages.

3.4.1 Structural reflection

Structural reflection in functional and logic programming have concentrated on the issues of quoting as well as amalgamation of language and metalanguage. Lisp is not an amalgamating language in the sense of Bowen [Bow82] because he requires an explicit representation of the language's processor. But the Lisp quote construct does amalgamate language and metalanguage: it uses a term of the base language to represent itself at the metalevel, albeit in a quoted form. The work of Smith on 2-Lisp and Muller on M-Lisp have focused on non-amalgamating representations, distinguishing between levels of designation that are mediated by explicit conversion operations UP and DOWN. In logic programming, amalgamation is intimately linked to the problem of logic variables. When an amalgamation is used, the deduction at the metalevel becomes fragile. If the language allows the terms to be looked at, nothing prevent the metalevel from unifying base level variables to metalevel terms, which would result in an absurd deduction (or even paradoxes, see below). To rule out this eventuality, some languages have adopted a type system that prevents the inappropriate mixing of metalevel and base level entities.

It is worth noticing that quoting, or converting through UP in 2-Lisp of M-Lisp, acts as if a piece of code is translated into a metalevel representation. This metalevel representation may be inspected or modified, and then translated back into base level code. However, at any time, a particular piece of code exists only in one representation, either the base level or the metalevel one. Hence, there is no need for something like a causal connection link between two different representation. In functional programming, the manipulation of functions as list have been abandoned a long time ago. In Prolog, the manipulation of (interpreted) code is still used[3], but again only one representation at a time exists. If a modification has to be made to a clause, the program first seizes it as a term, modifies it without connection to the actual program, and then reinstalls the new clause.

In OOP, structural reflection has been dominated by the quest for a complete and minimal metalevel model to represent instances, classes and metaclasses. Classes are used to reason about the structure of their instances, but there is no distinct representation of an instance when it is refered to at the metalevel. In this sense, OOP does also amalgamate language and metalanguage. The quote construct of Lisp, associated with the use of lists to represent the base level term, corresponds to the object's identity and its class as first-call entity. In Smalltalk, having access to the list of instance variables declared in the class allows the program to access the object as a vector: we can compute the index of the instance variable and send the object messages like `instVarAt:` and `instVarAt:Put:`. Therefore, instead of having two representations, we have two different protocol to access the same information: the standard one at the base level (which usually encapsulates the state of the object) and one at the metalevel (which sees the object as a vector of values). Compared to the quote construct, the OOP provides two points of view (base and meta) on the same piece of information, which are obviously causally connected.

OOP languages like Smalltalk and CLOS have also included an extensive representation of programs with objects. In Smalltalk, methods are represented by instances of the class `CompiledMethod`. These objects do not store the source code itself but rather a bytecode representation. The source code is easily accessible (the browser uses it directly) because the instance of `CompiledMethod` stores a pointer into the source file where the corresponding method is defined. Note that a change to a previously compiled method must be explicitly recompiled to be taken into account by Smalltalk; there is no causal connection between the textual representation and the bytecode representation. There is a causal connection between the bytecode representation and the physical's processor code actually run by Smalltalk, but this is a different story that goes beyond the present paper. CLOS follows a similar patterns with its generic functions and methods implemented as CLOS objects.

3.4.2 Behavioral reflection

Behavioral reflection has been divided into access to the language's processor on one hand, and access to the actual data used to run the program on the other hand. We now look at these two issues in turn.

Ideally, all reflective languages should provide the programs with a complete reification of the language's processor. A representation of the processor should be given in terms of the language itself, which could be modified by the program to be adapted to its own execution. This ideal has been tried in languages of the three paradigms without great success, mainly because of a lack of efficiency (or lack of suitable implementation techniques to render it efficient). For example, 3-Lisp does not reify the processor itself. Its functional successors have essentially followed the same trend, as well as the object-oriented ones except some attempts around the reification of apply methods. In logic programming, on the other hand, the reification of the derivability relation has been the focus of several works. But by now, very few still

[3]Mainly because asserts and retracts are the only means to memorize results across backtracking.

work on it, mainly for efficiency reasons.[4]

Admittedly, making the language processor fully accessible is a formidable challenge to our current implementation technology. An interesting approach recently explored is to provide the users with an open compiler through they could introduce modifications to the language's semantics [LKRR92]. This approach has similar properties as the metaprogramming approach using metainterpreters: modifications can be done once before the program execution, so they cannot depend upon the current inputs or intermediate results. In Smalltalk, the compiler is part of the program run-time system, so it is theoretically possible to recompile part of the program at run-time, thus obviating the previous remark.

Several languages have been proposed with a less powerful model of behavioral reflection, yet giving the programs access to the processor's run-time data. In functional programming, 3-Lisp have popularized the reification of the standard functional processor's registers containing the current expression (actually the argument structure), environment and continuation. A reflective procedure is passed these informations upon invocation. Reflective procedures are a weaker means to modify the language processors since their semantics can be viewed as actually inserting new lines of code into the processor itself. The same kind of approach has been tried successfully in the logic programming paradigm.

Moreover, two alternatives have been tried as far as the pair of opposite operations reification-reflection is concerned. Automatic pairing consists to have reflective procedure receiving the run-time data as parameters and to reinstall them automatically when the reflective procedure returns control to the interpreter. The other alternative, the manual one, consists to provide two primitives, one that reads the data and returns it in a reified form, and the second that takes a reified representation of the data and reinstalls it into the processor's internal data structure.

A last point concerning the comparison between logic, functional and object-oriented behavioral reflection concerns the presence or absence of reflective towers. Pionnered by 3-Lisp, it was not obvious that reflective towers were involved in other forms of behavioral reflection, such as the one implemented in OOP languages. Interestingly, des Rivières [dR90] has shown that the CLOS MOP includes in fact a reflective tower hidden within its reification of the generic function invocation protocol. A little later, Malenfant et al. [MDC94] have shown a similar result for languages using a more traditional *lookup∘apply* reflective protocol also give rise to reflective towers. Hence, reflective towers seem to be intimately linked to the current model of behavioral reflection, no matter what is the underlying programming paradigm.

3.4.3 Methodology of reflective computation

Ideally, a reflective language should support a methodology of reflective computation giving its users as much flexibility as possible. It should be possible to modify the behavior of some construct for the whole execution of a program, or for short period of (execution) time. It should also be possible to modify the behavior of some construct for all the program only for some of its subparts. None of the language currently proposed achieves this level of flexibility.

In functional programming and logic programming, the main efforts revolved around reflective procedure that are called at some point during the computation. Upon the end of this reflective computation, the base level is restarted perhaps in a new state resulting from the reflective computation itself. All the meta-interpreters in the towers are the same and stay unchanged during the whole execution. As a result, the kind of reflective computation of these models have been restricted to punctual intervention during the execution. 3-Prolog [Dem94] is a notable exception to this since it allows the program to specify that some reflective predicates must be called repeatedly, without having to explicitly insert calls to them within the program code. A repeated reflective predicate will be called between each reduction of the 3-Prolog metalevel solver. No other language based on reflective procedures considered this kind of behavior.

In OOP, the main efforts revolved around local reflection. In the *lookup∘apply* approach, lookup methods are redefined locally for some objects only while apply methods are redefined locally for some methods only. Similarly, in the reified generic function approach, the invocation protocol is changed locally, in a per generic function way. As a result, the kind of reflective computation you can do is also restricted; it is awkward to change one aspect of the implementation for all the objects. An open research problem is to decide what is the scope of a change in the lookup and apply methods and reflective computation in general [Fer89]. Should they be attached to individual objects, to classes (and apply to all their instances), to metaclasses (and apply to the instances of all its class instances) to individual messages (and apply only to one call)?

As noted previously, the ideal complete reification of language's processor has stayed out of reach mainly for efficiency reasons. Consequently, modifications spawning over the whole program execution are difficult to implement. With the reified compiler approach, it becomes possible to modify the semantics of the language in a per construct basis. However, very few such experiences have been reported yet.

4 Formal notions for Reflection

We discuss here of formal notions used to characterize reflection in programming languages of the three different paradigm, preceded by a historical discussion of reflection in logic freely inspired by [Per85, Per88], [Wey80], [Bow82], [Fef62] and [Cos90].

4.1 Historical discussion

Gottlieb Frege developed the first formal quantificational logic over a period of more than two decades culminating in 1903. The idea was to have a *universal* language for logic. For Frege, an object c and the properties P it may have were all objects to be reasoned about in the same way, i.e., with the same basic rules and notations. Frege had certain comprehension axioms that specifically created object-notations "P" for P, and stated that sentences using properties as predicates could be equivalently rephrased using properties as objects. These axioms in effect state a relationship between a name and what it names [Per85]: $\text{Has}(c, \text{“}P\text{”}) \leftrightarrow P(c)$ or equivalently, but closer to Frege's notation: $c \in \{x|P(x)\} \leftrightarrow P(c)$.

In the same year Bertrand Russell showed that Frege's system was inconsistent. Russell then proposed that objects be arranged in a hierarchy with different notations and rules, thus avoiding the possibility of self-reference that led to the inconsistency in Frege's system. The resulting "typed" (analogy could be made with the notion of reflective tower which is also a way to avoid inconsistency in programming languages)

[4] Some people still work on making the language Gödel (which reifies the full derivability relation [HL94]) as efficient as Prolog by using partial evaluation and parallel implementation techniques. Unfortunately this is not achieved yet.

has as its first level of notations precisely that of Frege, but without the damaging axioms that created objects out of properties at the first level. For Russell, properties of first-level objects are to be viewed as second-level objects. For this reason we refer to first-order logic and higher-order logics. However as we know many significant concepts cannot be expressed at all with levels. The original simplicity and plausibility of Frege's approach has then continued to attract interest, and much of modern logic has been motivated by efforts to revise it to preserve its desirable features while removing inconsistency. But what seems to be needed is an avoidance of separated levels altogether, so that all concepts are treated at the same (first) level. In fact it appears reasonable to allow a certain number of levels to be collapsed into first-order logic, and leave the rest out. It would not be desirable to collapse all levels into first-order logic because it is just what causes Russell's paradox. The main problem was thus addressed in logic as the need to find a way to collapse all levels into one without contradiction, i.e. the need to have a self-referential or universal language. In [Per85], it has been said that quotation seems necessarily involved at some point if we are to have a self-describing language and to treat logical terms as some sort of first-class entities.

In a same way, other reflective notions (other than the ones going back to famous Gödel incompleteness theorems in mathematical logic) have been introduced related to symbolic logic in the sixties. The main idea was concerned with the "reflection principles" mentioned earlier. The concept of reflection principle was introduced by Feferman [Fef62] where it was intended as "the description of a procedure for adding to any set of axioms A certain new axioms whose validity follows from the validity of the axioms A and which formally express, in the language of A, evident consequences of the assumption that all the theorems of A are valid". Later in [Kre68], different reflection principles are discussed for establishing the complexity of different axiomatic systems. In particular, Kreisel says:

> "Turning now to uses of reflection principles we recall two from the literature. First given a system S that has been recognized to be sound, reflection principles provide a systematic method for constructing stronger systems [...]. Second, they provide a method for comparing the *strength* of formal systems. Thus, if $S \subseteq U$ (where U is another system) and if the reflection principle R_S[5] can be proved in U then U has more theorems than S. What makes reflection principles useful is that they have a clear intuitive meaning, and so, if such a principle is provable in U, we have a good chance of finding a proof."

4.2 Logic for computation

A more pragmatic form of reflection principles, based on a different naming mechanism, was proposed by Weyhrauch [Wey80] and then in [Bow82]. We have already discussed the reflective principles presented in the latter. In practice, this approach leads to explicit reflection, that is, explicit calls to `Demo` must appear in clauses to specify that a goal should be attempted at the metalevel. These results remind clearly the need of meta-programming facilities to amalgamate object-level with some meta-level languages.

In Perlis [Per85], a theory of "quotation" and "unquotation" is presented to refer to certain statements as true and false. The author concentrates on the importance of the roles of truth and self-reference in commonsense reasoning. In his following article [Per88], he has showed that modal logics are on no firmer ground than first-order ones when equally endowed with substitutive self-reference. Thus his work has tried to understand the relationship between self-reference and modal logic.

[5] A suitable one that is called *local reflection principle for S* by the author.

4.2.1 Declarative semantics in the logical style

The next attempts to formalize of reflection are very close to the classical Horn-clause language semantics [Llo87]. In [Cos90], the declarative and procedural semantics of the Reflective Prolog [CL89], which, on the basis of a naming mechanism (that allows the representation of terms and atomic formulas), makes a different use of amagamating languages' reflection, both procedurally and semantically. Declarative semantics is defined in a model-theoretic fashion, providing a least model semantics based on the definition of a concept of *Reflective Herbrand Model* of a theory. The least reflective Herbrand model is then characterized as the least fixpoint of a suitable mapping, in order to provide a link between the declarative and procedural semantics of a program. Derivation by resolution is extended to include forms of implicit reflection to shift between levels. Extended resolution is proved sound and complete with respect to the least reflective Herbrand model of a program.

In [Sug90], a declarative semantics of R-Prolog* is discussed. Because computational reflection is a sort of procedural notion, the usual declarative semantics given as logical consequence of programs cannot be adopted. In order to incorporate a procedural aspect of reflective computation, the notion of interpretations and models is extended based on the equivalence classes of syntactic objects. They prove soundness and completeness of their R-refutation mechanism with respect to the declarative semantics defined. However, that reflective operations in R-Prolog* could lead to somewhat dangerous situations which are not studied semantically.

In [Chr90], a declarative semantics of a meta-programming language (called G_{CP}) concerned with metagoals is proposed. Although this language does not say offering computational reflection, it has a good structural reflection by using a kind of ground representation and also it is explicitly compared with metaProlog [Bow85]. The development of the semantics presented is quite traditional [Llo87] except that the Herbrand interpretations defined are said to be concerned with metagoals instead of with goals. It seems however that it is similar to the analysis of meta-programs made in [HL89] and [HL94].

4.3 Semantics in the functional style

In Section 3.1, we have presented the work of Friedman and Wand with the development of the different versions of Brown in response to 3-Lisp. In fact, as mentioned earlier, Wand and Friedman tried to settle a denotational account of reflective towers [FW84], which culminated with [WF88]. Smith has defended for a long time that the denotational framework was not the right tool to approach the problem. Although he essentially argued that denotational semantics was forcing the distinction in levels of designation to disappear, and thus evacuating the essence of reflection, the final word came from the observation that the compositionality principle of denotational semantics is irremediably impaired by reflection [DM88]. Hence, the denotational semantics approach, even though it has been helpful to understand the nature of reflection, cannot fully express its semantics. New approaches must be sought to reach the goal.

4.4 Rewriting systems

Recently, Friedman and Mendhekar [MF93] as well as Malenfant, et al. [MDC94] independently applied the theory of rewrite systems to this end, with encouraging results. The goal of Friedman and Mendhekar is to develop a programming logic for reflective languages. They introduce a reflective extension of the λ_v-calculus and provide it with a simple operational semantics where reflective operations are based on the underlying rewrite systems. They get a reflective language that includes an infinite tower model similar to the one described by Smith. An equational logic from this semantics is developped. However the resulting logic is shown to be weak because of the reflective properties.

Malenfant, Dony and Cointe have formalized the use of the *lookup∘apply* reflective protocol using the theory of *priority rewrite systems*. In these systems, each rewrite rule is assigned a priority and the reduction process garantees that among all rules applicable to the reduction of a redex, the one with higher priority will be effectively used. The authors have exhibited a mapping that, applied at each message sending point, transforms the reflective object-oriented program into a rewrite system, which is shown to be terminating. The theory applies only to the reflective protocol itself, not to the rest of the computation (method execution), but a complete rewriting semantics can be envisaged.

4.5 New promising approaches

Over the last few years, the categorical approach has gained considerable acceptance especially in the lazy functional programming world. Originally proposed by Moggi as a convenient framework for structuring the semantics of languages [Mog89] [Mog91], they were popularized by Wadler [Wad90] [Wad92] and others as a technique for structuring functional programs. In fact, monads can be used to give the semantics of various computational effects such as state, exceptions, or I/O in applicative programming languages. Would it be possible to give a monadic semantics of computational reflection? Filinski [Fil94] says:

> "The correspondence principle can be embodied in an introspective language extension which could be called **monadic reflection** by analogy to computational reflection, given by two operators:
>
> $$\frac{\Gamma \vdash E : T\alpha}{\Gamma \vdash \mu(E) : \alpha} \quad \text{and} \quad \frac{\Gamma \vdash E : \alpha}{\Gamma \vdash [E] : T\alpha}$$
>
> For any $E : T\alpha$, $\mu(E)$ *reflects* the value of E as an "effectful" computation of type α. Conversely, given a general computation $E : \alpha$, $[E]$ *reifies* it as the corresponding "effect-free" value of type $T\alpha$."

Note that the two operators mentioned are quite comparable to the reflection principles exposed in the earlier Section 3.2 on logic programming.

Recently a lot of research is made to introduce new formal calculi based on a categorical semantics. In [Mog91], it is said that "the logical approach (to express these new calculies compared with the more commun operational and denotational approach) gives a consequence relation $\vdash$, namely $Ax \vdash A$ iff the formula A is true in all models of the set of formulas Ax, which can deal with different programming languages (e.g. functional, imperative, non-deterministic) in a rather *uniform* way, by simply changing the set of axioms Ax, and possibly extending the language with new constants". This tendency to use a logical approach based on categorical notions seems to be quite promising to study programming languages and particularly computational reflection . Some people [HD94] have already tried to reason in the new logical frame proposed by Moggi [Mog91] and they say about it the following lines that corroborates in a way with the idea that monads could be used to model computational reflection.

> "Moggi's framework seems to provide a solid basis for studying both the relation between implicit and explicit representations of control and the relation between implicit and explicit representations of state, in a typed setting."

However much more research should be done to find the right relation between monadic reflection and computational reflection.

Finally, Cartwright and Felleisen [CF94] introduce a new format for denotational language specifications, **extended direct semantics**, that accommodates orthogonal extensions (as reflective mechanism) of a language without changing the denotations of existing phrases. It is not yet clear how the new approach could be use to formalize reflective languages. Further research must be done.

5 Conclusion

In this paper, we have described the different approaches to reflection in three major programming paradigms: logic, functional and object-oriented programming. We have compared them and pointed out similarities as well as fundamental differences. We have also defined the main concepts of computational reflection and presented some historical development of the idea of reflection in logic, as well as recent attempts at formalizing the concept of reflection.

It is our hope that this comparison, albeit incomplete, will help building bridges among the different research communities. If it encourages them to share their respective accomplishments in order to cross-fertilize their work, our goal will be fully achieved.

References

[AMY93] K. Asai, S. Matsuoka, and A. Yonezawa. Duplication and Partial Evaluation to Implement Reflective Languages. In [WRM93]

[AR89] H. Abramson and M.H. Rogers, eds. *Metaprogramming in Logic Programming.* MIT Press, 1989.

[Baw88] A. Bawden. Reification without Evaluation. In *Proc. of ACM L&FP'88*, pages 342–351, 1988.

[BC89] J.-P. Briot and P. Cointe. Programming with Explicit Metaclasses in Smalltalk-80. *Proc. of OOPSLA'89, ACM Sigplan Not.*, 24(10):419–431, Oct. 1989.

[BCL+88] M. Bugliesi, M. Cavalieri, E. Lamma, P. Mello, A. Natali, and F. Russo. Flexibility and efficiency in a prolog programming environment: Exploiting metaprogramming and partial evaluation. In *Proc. of the 5th ESPRIT Conf., Brussels*, 1988.

[BGW93] D.G. Bobrow, R.P. Gabriel, and J.L. White. CLOS in Context — The Shape of the Design Space. In A. Paepcke, editor, *Object-Oriented Programming — The CLOS Perspective*, chapter 2. MIT Press, 1993.

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. ComonLoops: Merging Lisp and Object-Oriented Programming. *Proc. of OOPSLA'86, ACM Sigplan Not.*, 21(11):17–29, Nov. 1986.

[Bow82] K. A. Bowen. *Logic Programming*, chapter Amalgamating Language and Metalanguage in Logic Programming. Academic press, 1982.

[Bow85] K. A. Bowen. Meta-Level Programming and Knowledge Representation. *New Generation Computing*, 3:359–383, 1985.

[Bru90] M. Bruynooghe, editor. *Proceedings of the Second Workshop on Meta-programming in Logic*. K.U. Leuven, Department of Comp. Sc., April 1990.

[CF94] R. Cartwright and M. Felleisen. Extensible denotational language sepecifications. In *Proc. of TACS'94*, vol. 789. Springer-Verlag, LNSC, April 1994.

[Chr90] H. Christiansen. Declarative Semantics of a Metaprogramming Language. In [Bru90], pages 159–168, April 1990.

[CL89] S. Costantini and G. A. Lanzarone. A metalogic programming language. In *Proc. of the 6th Int. Conference in Logic Prog. Workshop*, 1989.

[Coi87] P. Cointe. Metaclasses are First Class: the ObjVLisp Model. *Proc. of OOPSLA'87, ACM Sigplan Not.*, 22(12):156–167, Dec. 1987.

[Cos90] S. Costantini. Semantics of a metalogic programming language. *Int. Journal on Foundations of Computer Science*, 1(3), 1990.

[Dem94] F.-N. Demers. Réflexion de comportement et évaluation partielle en Prolog. Master's thesis, U. de Montréal, DIRO, Avril 1994. Tech. report no. 956.

[DM88] O. Danvy and K. Malmkjaer. Intensions and Extensions in a Reflective Tower. In *Proc. of ACM L&FP'88*, pages 327–341, 1988.

[dR90] J. des Rivières. The Secret Tower of CLOS. In [WRM90].

[dRS84] J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proc. of ACM L&FP'84*, pages 331–347, 1984.

[Fef62] S. Feferman. Transfinite Recursive Progressions of Axiomatic Theories. *Journal of Symbolic Logic*, 27:259–316, 1962.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. *Proc. of OOPSLA'89, ACM Sigplan Not.*, 24(10):317–326, Oct. 1989.

[Fil94] A. Filinski. Representing monads. In *Proc. of ACM POPL'94*, pages 446–457, 1994.

[FW84] D.P. Friedman and M. Wand. Reification: Reflection without Metaphysics. In *Proc. of ACM L&FP'84*, pages 348–355, 1984.

[Gal93] J. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98, June 1993.

[GWB91] R.P. Gabriel, J.L. White, and D.G. Bobrow. CLOS: Integrating Object-Oriented and Functional Programming. *Comm. of the ACM*, 34(9):29–38, 1991.

[HD94] J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Proc. of ACM POPL'94*, pages 458–471, 1994.

[HL89] P. M. Hill and J. W. Lloyd. Analysis of metaprograms. In [AR89].

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[JF92] S. Jefferson and D.P. Friedman. A Simple Reflective Interpreter. In [YS92], pages 48–58, 1992.

[KRB91] G. Kiczales, J. Des Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kre68] G. Kreisel. Reflection principles and their use for establishing the complexity of axiomatic systems. In *Zeitschrift Fur Mathematische Logik und Grundlagen der Mathematik*, vol. 14, pages 97–142. 1968.

[LKRR92] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An Architecture for an Open Compiler. In [YS92], pages 95–106, 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.

[LMN91] E. Lamma, P. Mello, and A. Natali. Reflection mechanisms for combining prolog databases. In *Software Practice and Experience, 21(6)*, pages 603–624, 1991.

[Mae87] P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987.

[MDC94] J. Malenfant, C. Dony, and P. Cointe. A Semantics of Introspection in a Reflective Prototype-Based Language. to appear in Lisp and Symbolic Computation.

[MET94] *Proc. of the fourth Workshop on Meta-programming in Logic*. Springer-Verlag, LNCS 883, 1994.

[MF93] A. Mendhekar and D.P. Friedman. Towards a Theory of Reflective Programming Languages. In [WRM93].

[MLV89] J. Malenfant, G. Lapalme, and J. Vaucher. ObjVProlog: Metaclasses in Logic. In *Proc. of ECOOP'89*, pages 257–269. Cambridge U. Press, July 1989.

[MLV90] J. Malenfant, G. Lapalme, and J. Vaucher. Metaclasses for Metaprogramming in Logic. In [Bru90], pages 257–271, April 1990.

[MN88] P. Maes and D. Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[Mog91] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1), 1991.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proc. of LICS'89*.

[Mul92] R. Muller. M-LISP: A Representation Independent Dialect of LISP with Reduction Semantics. *ACM TOPLAS*, 14(4):589–615, Oct. 1992.

[Per85] D. Perlis. Languages with self-reference I: Foundations (or: We can have everything in first-order logic!). *Artificial Intelligence*, 25:301–322, 1985.

[Per88] D. Perlis. Languages with self-reference II: Knowledge, Belief, and Modality. *Artificial Intelligence*, 34:179–212, 1988.

[Pet92] A. Pettorossi, editor. *Proceedings of the Third Workshop on Meta-programming in Logic*. Springer-Verlag, LNCS 649, April 1992.

[Smi82] B.C. Smith. Reflection and Semantics in a Procedural Language. Tech. Report 272, MIT, 1982.

[Smi84] B.C. Smith. Reflection and Semantics in Lisp. In *Proc. of ACM POPL'84*, pages 23–35, 1984.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Sug90] H. Sugano. Meta and reflective computation in logic programs and its semantics. In [Bru90].

[Wad90] P. L. Wadler. Comprehending monads. In *Proc. of ACM L&FP'90*, pages 61–78, 1990.

[Wad92] P. L. Wadler. The essence of functional programming. In *Proc. of ACM POPL'92*, pages 1–14, 1992.

[Wey80] R.W. Weyhrauch. Prolegomena to a Theory of Mechanized Formal Reasoning. *Artificial Intelligence*, 13(1,2), 1980.

[WF86] M. Wand and D. P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proc. of ACM L&FP'86*, 1986.

[WF88] M. Wand and D. P. Friedman. The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988.

[WRM90] *Proc. of the First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, Oct. 1990.

[WRM91] *Proc. of the Second Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'91*, Oct. 1991.

[WRM93] *Proc. of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93*, Oct. 1993.

[YS92] A. Yonezawa and B. Smith, editors. *Proc. of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*, Nov. 1992.