A Tutorial on Behavioral Reflection and its Implementation

J. Malenfant, M. Jacques and F.-N. Demers*
Département d'informatique et recherche opérationnelle,
Université de Montréal, Montréal, Québec, CANADA

Abstract

The efficient implementation of behaviorally reflective languages imposes a formidable challenge. By definition, behavioral reflection allows a program to modify, even at run-time, its own code as well as the semantics and the implementation of its own programming language. This late-binding of the language semantics favors interpretive techniques, but compilers are absolutely necessary to make reflective languages efficient and therefore of real interest. The goals of this tutorial are: (1) to give the picture of the state of the art in the efficient implementation of behavioral reflection, (2) to review the main issues in going from interpreter-based to compiler-based implementations and (3) to propose new avenues towards the realization of this objective. Our tutorial is aimed at a large audience of reflective language implementors, in either object-oriented, functional or even logic programming. To make our point widely applicable, we avoid fine-grain technicalities. Rather we emphasize the common denominator of all reflective languages, we propose a clear and general problem statement, and we set up a wide-ranging research agenda.

1 Introduction

In the programming language area, we define reflection as "the ability of a program to manipulate as data something representing the state of the program during its own execution" [BGW93], the mechanism for encoding execution states as data being called *reification*. We further categorize reflective mechanisms in two major forms: *structural* and *behavioral reflection*. The chief interest of this distinction lies in the fact that structural reflection is considered easier to implement. Indeed, languages such as Lisp, Prolog, Smalltalk, and others have included structural reflection mechanisms for a long time.

Behavioral reflection has not been so clearly tackled yet, essentially because it touches aspects governing the semantics of programs. When confronted with behavioral reflection, most language implementors adopt interpretive techniques. Interpreters ease modifications and react to them as soon as they occur, a remarkable advantage in reflection. But to improve the applicability of reflective languages, there is no way around more efficient implementations. Unfortunately, the lack of a precise understanding of the issues involved in this evolution pushes implementors to limit the reflective models. In this tutorial, we resist this temptation. We first propose a comprehensive look at these issues. Next, we suggest new avenues towards more efficient implementation techniques for full behavioral reflection.

Our approach to this problem is based on the recognition that reflection is, in fact, pursuing a long tradition in computer science [Hal93, Binding]: "Broadly speaking, the history of software development is the history of ever late binding time ..." Reflection pushes this idea to its limit by postponing the binding time of at least part of the language

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

syntax, semantics and implementation as well as the program itself, to the run-time. In the extreme case, this possibility challenges our capability to efficiently implement reflective languages (i.e., compile them). But if behavioral reflection indeed includes such an extreme possibility, we claim that most of the time, typical programs will behave in such a way that most of their code can be compiled using standard techniques, or at least techniques that are within the reach of the current research on the implementation of modern dynamic programming languages.

In this sense, we consider that reflective language designers must give as much freedom as possible to programmers in terms of late-binding, but their implementors must use appropriate techniques to extract static computations from programs in order to compile and optimize them prior to run-time. To this end, we introduce the distinction between *static* and *dynamic* behavioral reflection. Coarsely speaking, static behavioral reflection is a restricted form of reflection where we statically know enough information about the reflective computations in a program to compile it.

Our goal is to precisely characterize the static case in order to enable the development of new compiler tools and techniques to tackle static reflection, even in languages allowing dynamic behavioral reflection. Reflective language designers and implementors will then face two possible paths. Indeed, one of them is to provide highly efficient languages with only static behavioral reflection. But we also propose that dynamic behavioral reflection can be introduced, at the price of dynamic compilation and cache techniques similar to the ones already developed in Smalltalk [GR83] and Self [USC+91] for example. The essential motivation behind these techniques is that changes happen infrequently enough that we can compile the code under the assumption that everything is stable, and then pay the price of dynamically recompiling part of it each time a change occurs at run-time.

In the rest of this tutorial, we first recall the basics of reflection in programming languages. In Section 3, we explore the state of the art in the implementation of behavioral reflection. In Section 4, we move on to the issues related to binding time and behavioral reflection; we introduce a crucial distinction between static and dynamic behavioral reflection. In Section 5, we propose new avenues in the implementation of reflective languages based on this distinction. We look at four problems crucial to the current research: how to compilation reflective towers, how to handle introspection and causal connection, how to take into account modifications to the semantics of the language and how to tackle dynamic behavioral reflection. We then conclude and expound some perspectives.

2 Reflective programming languages

Reflective programming languages have existed since the seminal work of Smith in the early eighties, yet the area lacks a widely accepted account of what is reflection and what are reflective languages. Some confusion arises because reflective properties already appear in several existing languages. In this section, we establish our terminolgy for the basic concepts used throughout this paper. Because we are particularly interested in the efficient implementation of reflective programming languages in the large, our definitions are biased towards a better understanding of implementation issues (a companion paper [DM95] provides an historical and comparative overview of reflective concepts and languages), beginning with the following basics about programming languages.

Definition 2.1 *A* **programming language** *is a medium to express computations, which is defined by its syntax and its semantics. An* **implementation** *of a programming language is the realization of its syntax and semantics, which comprises a translator and a run-time system. A* **program** *written in a particular programming language is a syntactically well-formed sequence of symbols that expresses a computation, which semantics is described by mapping its syntactic constructs to formal descriptions using the programming language semantics, and gathering them together to express its computation.*

The effective computation is obtained by translating the program to a suitable low-level sequence

of actions to be executed on a computer in conjunction with the language run-time system. When the translation is done on-line, expression by expression, and interleaved with the actual computation, the implementation is said to be an interpreter for the language; when it is done off-line, all at once, usually prior to the execution, the implementation is said to be a compiler for the language.

Given these basics, an effective or operational definition of reflection in the context of programming languages might well be the following:

Definition 2.2 Reflection *is the integral ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics, or implementation), even at run-time. A programming language is said to be* **reflective** *when it provides its programs with (full) reflection.*

The word integral in the above definition is very important. We insist on it to promote the idea that true reflection imposes no limit on what the program may observe or modify. This is clearly a long term goal. There are even theoretical limits (Gödel incompleteness theorem, paradoxical situations, etc.). But for the time being, we will stick to this definition to avoid including everything in reflection, and considering all languages as reflective. We will come back to this issue shortly.

In order to observe or change something, this thing must be represented in a such a way that the user program can manipulate it. This paves the way to the notion of reification.

Definition 2.3 Reification *is the process by which a user program $\mathcal{P}$ or any aspect of a programming language $\mathcal{L}$, which were implicit in the translated program and the run-time system, are brought to the fore using a representation (data structures, procedures, etc.) expressed in the language $\mathcal{L}$ itself and made available to the program $\mathcal{P}$, which can inspect them as ordinary data. In reflective languages reification data are* **causally connected** *to the related reified information such that a modification to one of them affects the other. Therefore, the reification data is always a faithful representation of the related reified aspect.*

Reification, at least partially, has been experienced in many languages to date: in early Lisps and in current Prologs, programs have been treated as data, although the causal connection has often been left to the responsability of the programmer. In Smalltalk-80, the compiler from the source text to bytecode has been part of the run-time system since the very first implementations of the language. Several other examples exist.

Notice the importance of the capability for the program to handle the reification data. This is in contrast with higher order languages for example, in which entities that were implicit are given a first-class status. First-class entites need only be denotable and used as any other primitive data types of the language: they may be passed as parameters, returned as results, assigned to variables. This is the case of functions and continuations in Scheme for example. But they need not be represented by data structures that can be inspected.

Most of the difficulties in giving a precise definition of what is reflection and what it is not stems from the fact that several existing languages exhibit some reflective behavior or provide reflective mechanisms, a notion that we define as follows:

Definition 2.4 *A* **reflective mechanism** *is any means or tool made available to a program $\mathcal{P}$ written in a language $\mathcal{L}$ that either reifies the code of $\mathcal{P}$ or some aspect of $\mathcal{L}$, or allows $\mathcal{P}$ to perform some reflective computation.*

It is difficult to draw a precise and tight frontier between what is considered as a few appearances of reflective mechanisms and full reflection. Indeed, the above definition is very general. It aims at gathering all expressions of reflective behavior or reflective properties in existing languages and systems. Our goal in introducing these distinctions is both to contrast selective expression of reflective properties from true (fully) reflective languages. On the other hand, we want to reuse the know-how developed around these mechanisms in existing languages and systems. For example, reification of program code has been done in several languages, but this does not mean that these languages were reflective. On the other hand, because reflection needs reified programs, the technology developed in Lisp, Prolog,

Smalltalk and other languages is readily available for reflective languages.

Some reflective mechanisms are now pretty well-understood and their efficient implementation is also well-documented: the aforementioned programs reified as data[1], representation of program entities at run-time, for example using classes and metaclasses as first-class entites in object-oriented programming, etc. We will see later on that a lot of existing techniques provide a very helpful basis for the efficient implementation of reflective languages. In fact, the observation that the well-mastered cases lie much more in the static representation of programs side of reflection lead naturally to a distinction between structural and behavioral reflection.

Definition 2.5 Structural reflection *is concerned with 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.*[2]

Definition 2.6 Behavioral reflection, *on the other hand, is concerned with the ability of the language to provide a complete reification of its own semantics and implementation (processor) as well as a complete reification of the data and implementation of the run-time system.*

Pionnered by Smith [Smi82, Smi84], behavioral reflection has proved, as we noted before, to be much more difficult to implement efficiently than structural reflection. This is essentially because it raises crucial issues related to the execution of programs. First of all, behavioral reflection complicates the run-time model of the language by introducing reflective towers, as we will see shortly. When efficiency, and therefore compiling, come into play, the following questions are raised:

1. How to represent the language semantics in such a way that programs can modify it?
2. How to represent the run-time data in such a way as to provide programs with easy access to them while retaining efficiency?
3. How to compile programs where the semantics of the language can be modified?
4. How to compile programs where the implementation of the run-time system can be modified?

3 Implementing behavioral reflection

In this section, we discuss the fundamental aspects of behavioral reflection from an implementation point of view. We first explain why reflective towers are at the heart of behavioral reflection. Then, we look at the implementation of 3-Lisp and other languages to illustrate the state of the art in the efficient implementation of behavioral reflection.

3.1 Reflective towers

From the above definition, we know that behavioral reflection implies the ability of any program p to observe and modify the data structures actually used to run p itself. This may lead to inconsistencies if updates made by the reflective code come in conflict with updates made by the interpreter. This phenomenon, known as *introspective overlap*, is tackled in Smith's 3-Lisp [Smi82, Smi84] by making a distinction between the interpreter $\mathcal{P}_1$ running p and the interpreter $\mathcal{P}_2$ running the reflective code. Smith then takes the argument to its limit by allowing introspective code in the interpreter $\mathcal{P}_1$, which needs another interpreter $\mathcal{P}_3$ to be run, and so on, giving rise to a potentially infinite number of interpreters. In fact, in his design, the interpreter $\mathcal{P}_i$ is used to run the code of the interpreter $\mathcal{P}_{i-1}$, and so on, with the interpreter $\mathcal{P}_1$ running the base program. This stack of interpreters is called a *reflective tower*, and it is illustrated in Figure 1. Because the levels in the tower need not be based on interpretive techniques, Smith and des Rivières purposely use the term *reflective processor program* (RPP) instead of interpreter.

[1] provided that we are willing to pay the price of having partially interpreted code or a compiler available at run-time and being called from time to time to compile some source code. At first, languages with reified programs were condemned to interpretation. Now implementors either mix compiled and interpreted code, or compile and link the new code on the fly.

[2] For example, in a language providing lists as an abstract data type, structural reflection calls for providing the programs with a complete reification of the implementation of the list ADT, the internal representation and operations of which could be modified.

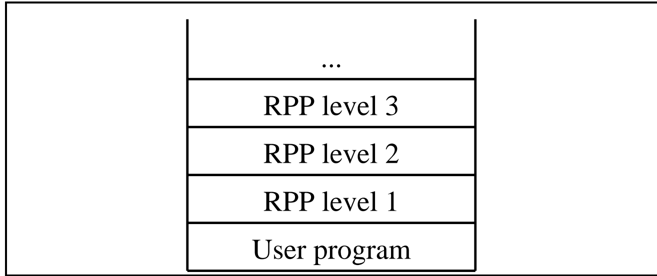

Figure 1: Reflective tower

Although the 3-Lisp reflective tower is potentially infinite, in any single program p and input i, only a finite number of levels n are needed to run the program; this number of levels is called the *degree of introspection* of p. In the same way well-defined recursions never require an infinite number of recursive calls, a well-defined reflective program never uses an infinite number of embedded reflective procedure calls. Hence, given n, we can replace the level $n+1$ interpreter by an ultimate non-reflective machine to run p on i.

To summarize, 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*, procedures with three parameters, the body of which being executed one level up in the tower; upon invocation, a reflective procedure is passed a reification of the argument structure of its call, its current environment and its current continuation.

Because 3-Lisp assumes that all the interpreters are the same (process the same language), a reflective procedure can be called at any level n in the tower, without having to care about the language in which it is written. When invoked at level $n-1$, a reflective procedure is run at level n, i.e., by the level $n+1$ interpreter to avoid the introspective overlap. It is as though the reflective procedure actually inserts lines of code into the interpreter one level up in the tower.

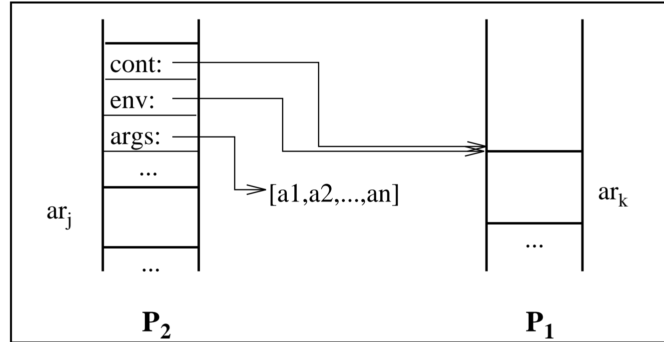

Figure 2: Stacks (continuations & environments) of P_1 and P_2 just after the execution of a reflective procedure call at level 0.

The reflective procedure call is shown in Figure 2. A reflective procedure is called at level 0. Normally, such a procedure call would generates a new activation record ar_{k+1}, but because the procedure is reflective, it is run by level 2, and thus generate a new activation record ar_{j+1} of the RPP at level 2. The reflective procedure is passed a reification of the argument structure of the call, the current environment and the current continuation, as shown by the values in ar_{j+1}[3].

3.2 Ubiquitous reflective towers

Reflective towers have been regarded as mysterious devices, but in fact they appear at the heart of any system exhibiting a form of behavioral reflection (to our current knowledge). For example, two important families of behaviorally reflective object-oriented languages have been proposed. One family is built around the generic function model, where the application of a generic function is reified as a generic function; the CLOS MOP [KRB91] is the quintessence of such systems. The second family is built around the *lookup* $\circ$ *apply* introspective protocol, where a message is decomposed into a lookup and an apply phases, both reified as methods in the language [MDC96]. Interestingly enough, reflective towers appear in both.

Des Rivières [dR90] has brought to the

[3] In fact, we are glossing over the reification in Figure 2. The actual values must take into account the packaging of the environment and the continuation into their reified counterparts. We will come back on this issue below.

fore the existence of a reflective tower in the CLOS MOP through the generic function `apply-generic-function`[4], which describes how a generic function is invoked. The tower appears because when invoked, as a generic function `apply-generic-function` must invoke itself. The potential infinite meta-regression is avoided by topping out to the internal implementation when the method for applying standard generic functions is invoked. In Malenfant et al. [MDC96], we have shown the existence of reflective towers in the *lookup* ∘ *apply* model. In this model, reflective towers appear because *apply* methods are themselves methods and must have their own apply method, and so on. Because they are so ubiquitous in behavioral reflection, the efficient implementation of reflective towers appears as a key issue.

Indeed there are differences between the 3-Lisp approach and the two latter ones, the most important from an implementation point of view being that the 3-Lisp tower is potentially infinite. In the object-oriented models, the towers are finite by construction and the languages always provide explicit rock-bottom entities that stop the tower at some fixed level, perhaps differing from methods to methods. This difference also implies very different philosophies of introspection. Because the interpreters in the 3-Lisp tower are all the same, the end user needs reflective procedures to introduce new meta-level code in the tower. In the object-oriented approach, the apply methods are build by the end user in such a way that they include the necessary meta-level code to perform the envisaged introspection. We therefore distinguish two general approaches: *discrete* and *continuous behavioral reflection.*

Definition 3.1 *Behavioral reflection is said to be* **discrete** *when the reflective computations are initiated at a discrete point by calling a reflective procedure and will only last until the reflective procedure returns. This approach is exemplified by the 3-Lisp approach,*

Definition 3.2 *Behavioral reflection is said to be* **continuous** *when reflective computations are explicitly inserted in the meta-interpreters and can have a continuous effect on the base level computation*[5]*.*

These two approaches favor quite different reflective applications. The discrete approach is good at interrupting the computation at some point, observing its state and taking some punctual action. For example, a reflective procedure can be called by a particular function to gather usage statistics in order to optimize it for the most frequent cases. The continuous approach lends itself to applications where some part of the semantics is modified more or less permanently, like changing the parameter passing mode. Although not exclusive, there are few examples, to our knowledge, of languages mixing both models (Simmons et al. [SJ92] provide such an example). In languages exhibiting discrete behavioral reflection, the interpreters in the tower are almost never modified (except through reflective procedure calls) while in languages exhibiting continuous behavioral reflection, reflective procedures are almost never provided (but they could easily be introduced using reflection).

3.3 The 3-Lisp level-shifting processor

The stack of meta-interpreters in a reflective tower suggests a direct implementation using interpretive techniques. With a loss of an order of magnitude in performance for each level of meta-interpretation, such a naive implementation is totally useless. Early in the development of behavioral reflection, researchers have proposed ways to avoid the levels of meta-interpretation. In fact, two main efforts have succeeded: an operational approach, with Smith's and des Rivières' level-shifting processor (LSP) [dRS84] and a formal approach, with Friedman's and Wand's meta-continuation semantics [WF88].

Both of them apply to the discrete approach and are based on the single-threadedness assumption [DM88]. This assumption says that at any time during the execution, only one of the meta-interpreters needs to be executing, namely the low-

[4] or `compute-discriminating-function`, see [KRB91]

[5] Continuous behavioral reflection bears some similarity to meta-interpreting. The main difference is that in meta-interpreting, the base level program is not aware of the meta-interpreter and cannot change it. In reflection, the program is aware of the meta-level and can change it, even at run-time.

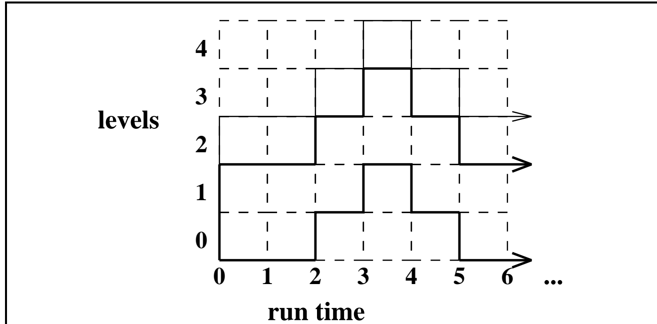

Figure 3: Level-shifting processor

est level which runs user's code, either in the program at level 0 (i.e., the meta-interpreter of level 1) or reflective code at some level i in the tower (i.e., the meta-interpreter of level $i+1$). The single-threadedness assumption relies on the fact that the meta-interpreters above this lowest level need not be executed because they perform an easily predictable computation (the code is known and there is no side-effects).

We now look at 3-Lisp level-shifting processor (LSP), but similar observations can be inferred from the meta-continuation semantics. Using the single-threadedness assumption, a 3-Lisp program p is executed by starting the LSP at level 1 on p, as illustrated in Figure 3. In this figure, the horizontal axis represents the run-time while the vertical axis represents levels in the tower; solid thick lines represent the locus of the computation. The LSP at level 1 is executed by a non-reflective version of itself (the *ultimate machine*) at level 2 (represented by the locus of the solid narrow line). Starting in this configuration is pure speculation. We conjecture that no reflective procedure calls will be made during this run.

When a reflective procedure is invoked at time 2, this conjecture appears to be false and we must conclude that we should have started with at least two levels in the tower running the LSP. This is the case because the body of the reflective must be executed by the level 2. But notice that until this point, the level 2 is the non-reflective version of the LSP, the ultimate machine, which cannot run reflective code. To process the reflective procedure body correctly, the level 2 must be the LSP.

We face two solutions: restart the program from the beginning with a three-level tower with the LSP at level 2 or try to create on the fly the same state of the LSP at level 2 if it had run since the beginning of the program, and to resume the computation from this state. The 3-Lisp LSP adopts the second solution. But, this warm start of level 2 imposes the creation of the values of the level 2 processor registers, namely its current expression, environment and continuation, without having to replay the program. If another reflective procedure is called at time 3, the level 3 LSP must now be created in the expected state we would obtain if it would have been run since time 0, and so on. Notice that when a reflective procedure is run at level n, no code is run at any level below n.

The ingenuity of the 3-Lisp LSP appears in the way it has been written that allows the creation of levels on the fly to be performed. How is this achieved? First notice that code can move up in the tower only when a (reflective) procedure call is made. Hence, this appears at a handful of places in the LSP program. Notice also that the LSP does not explicitly represent the state of I/O streams. In fact, the LSP makes no side-effect, as mentioned before. Hence, the state only contains an expression, an environment and a continuation. When shifting up, these three pieces of state can be *"pulled out of thin air"* because they are independent of the past computation at this level. The LSP does not accumulate state (its primitive procedures always call each others in tail position, so it is effectively a finite state machine) and it always carries the same continuation upon entry to its primitive procedures [dRS84] (in fact, the continuation of the next level up in the tower, which is the initial continuation unless some reflective code has been run at this level earlier in the computation). And this is true regardless of the level of computation, since all levels run the same LSP.

In conclusion, the 3-Lisp LSP gets rid of the levels of meta-interpretation. Des Rivières and Smith even report on an implementation where the 3-Lisp code is incrementally compiled at run-time into byte-codes of the underlying SECD machine, which provides a first compiled implementation of behavioral reflection. However, the above restrictions can hardly be applied to the *lookup* $\circ$ *apply* case. In this latter model, interpreters (or proces-

sors) in the towers are written by the end users for specific purposes, hence flatness (or boredom)[6] of levels doesn't make sense. All the levels are needed to execute methods correctly. It also appears difficult to prevent users from inserting side-effects in their apply methods. Tracing, for example, is one of the well-known applications of reflection that needs side-effects.

3.4 Other languages

Besides small languages, like Brown [WF88] that essentially embody the ideas of the 3-Lisp LSP in a more formal approach, very few attempts have been made to efficiently implement behavioral reflection. Most languages tackling behavioral reflection rely on interpretive techniques, or refrain from reifying the language processor *per se*.

In object-oriented languages, proposals have been made using interpretive techniques [Mae87, Fer89]. The CLOS MOP, on the other hand, reifies the treatment of messages in the form of the generic function invocation protocol, but this protocol essentially addresses the issue of method lookup. The method lookup is usually made by a discriminating function, which returns a funcallable object representing the method found. This funcallable object is merely invoked by a funcall, which refers to the standard semantics. The funcallable object representing a method is obtained by calling `make-method-lambda`. Its standard definition is simply to compile the method lambda definition to obtain a function, but non-standard treatments are also possible [KRB91]. In fact, `make-method-lambda` appears as a potential entry point to a compile-time MOP (see below). The CLOS MOP is now available in most implementations of CLOS, and an implementation is described in Kiczales et al. [KRB91].

4 Static versus dynamic behavioral reflection

Most existing reflective languages do not attack the problem of efficiency in behavioral reflection. We have seen that if 3-Lisp possesses a compiled implementation, this implementation is based on several hard constraints put on the design of its processor. To go further in this direction, we need to answer the following question: what does it mean to compile a behaviorally reflective language? To answer this question, we look at reflection from the point of view of binding times. We then derive the notion of static behavioral reflection related to compilation.

4.1 Reflection: the ultimate flexible

Instead of being a radical departure from current traditions in the evolution of programming languages, we strongly believe that reflective languages are in fact the ultimate result of a long and natural evolution. We concur with Halpern that the pursuit of ever late binding time had a great impact on the history of software development in general but also of programming languages in particular. The underlying principle is that all options should be kept open until the last possible moment.

Definition 4.1 *"***Binding** *means translating an expression in a program into a form immediately interpretable by the machine on which the program is run;* **binding time** *is the moment at which this translation is done." [Hal93, Binding]*

This restrictive definition has the advantage of being very clear. An example of a binding is the one of a variable name to a memory location, which happens at program design time in machine language (and in early programming languages) but has been postponed to compile time or run-time in modern programming languages. Another example is the binding in a procedure call of the procedure name to the address of the code to be run. In procedural languages, this binding is done at compile time while in object-oriented languages it is postponed to the run-time.

In practice, the notion of binding times has evolved towards a much more general one. We now speak about binding a program element to a particular property as the choice of the property among a set of possible properties. The class of binding times includes the design, specification or implementation of the programming language, or the design, specification, implementation, translation or running of

[6] a flat or boring level executes expressions that come solely from the LSP program.

the program. Furthermore, we distinguish formal binding times from actual ones.

Definition 4.2 *A* **formal binding time** *is the latest moment at which a binding can occur in general, while an* **actual binding time** *is the actual moment at which the binding occurs for a particular element of a particular program.*

For example, we have just mentioned the late-binding of procedure names to the address of the code in object-oriented languages. This is a formal binding time. In a particular case, a static analysis of the program code may allow the compiler to bind a particular method call in a program to a specific code address. This is the actual binding time of this particular method call.

The notion of binding time is crucial to understand, compare and contrast the design and implementation of programming languages. Later binding times introduce more flexibility, at the expense of transferring the costs of the bindings towards later stages of processing. The general trend in the evolution of programming languages has been to postpone formal binding times towards the running of programs, but to use more and more sophisticated analyses and implementation techniques to bring actual binding times back to the earlier stages, where the costs of bindings are less harmful. Recently, control-flow analyses have been successfully implemented in Scheme compilers to determine the set of possible lambdas that can be applied at each call site. Likewise, concrete type analyses have been used in compilers for object-oriented languages to determine the set of possible instantiation classes of the receiving object at each method call site, in order to perform the lookup for the actual method at compile time.

Reflective programming languages pursue this tradition of ever late binding times by postponing the binding of almost all elements of programs and languages to the run-time. In this sense, we can speak of an outcome of this long tradition where everything is subject to modification until, and even during run-time. This is clearly the challenge in the efficient implementation of reflective languages. However, in the rest of this paper, we will claim that if formal binding times are postponed to run-time, the research should focus on new tools and implementation techniques aiming at bringing the actual binding times back to compile time. The challenge is still formidable, but the road ahead reveals itself much more clearly.

4.2 Static information is the mainspring of compilation

The notion of statically known information in programs is central to compilers. Compilers serve two main purposes. First, they translate a computation expressed in a high-level source language (or model of computation) into an equivalent computation expressed in a (usually) lower-level language. The lower-level language is usually more efficient because it is directly implemented in hardware. Second, compilers process the static semantics of the computation, that is code that can be executed at compile time because it is known and it processes statically known data (it typically ranges from compile time checks for contextual properties to constant folding).

If the source and the target languages cannot be determined at compile time, compiling itself is indeed pointless. In behavioral reflection, none of them can be considered as known at compile time. In particular, it is perfectly conceivable, and even legitimate to be able to dynamically change the target language. This would be interesting for example when processes are migrated across a network of perhaps statically unpredictable heterogeneous processors, a typical application of behavioral reflection.

A legitimate question to ask is whether compiling behavioral reflection is of any interest. A first answer is efficiency. Unless we want to write programs in some reflective extension of assembler, or to produce hardware implementation of high-level reflective programming languages, acceptable performance can only be achieved by translation of reflective programs into assembly code. A more insightful answer is expressed in the following hypothesis:

Static behavioral reflection hypothesis
In as much as standard languages, by the virtue of their design, provide their programs with enough statically known information to be translated to some target language (i.e., source code, source language syntax and semantics, target language syntax and semantics as well as the transformation between the high-level abstract machine of the source language and a low-level abstract machine written in the target language), languages with static behavioral reflection, by the virtue of their design, provide their programs with enough statically known information to enable their translation to some target non-reflective language.

The major point here is that languages with static behavioral reflection can be both powerful and efficient. Moreover, in a language allowing dynamic behavioral reflection, programs can still exhibit static reflective computations that can be compiled.

The above hypothesis is purposely vague about what exactly we mean by "information" and "known at compile time". Both heavily depend on the language itself. To illustrate the point, consider the following example about functional programming languages. The essential characteristic of functional languages is functions as first-class values, i.e., the ability to create functions at runtime, to pass them as actuals and to return them as results. In early Lisps, functions were represented as lists and could be constructed at runtime. Today, Scheme also provides first-class functions but insists (through its syntax) that their code is known at compile time. Though more restrictive, a large number of applications advocated for functional programming can be implemented in Scheme. Moreover, Lisp compilers have adapted to the presence of functions as lists, although some amount of interpretation may be needed to deal with dynamically constructed functions.

The static behavioral reflection hypothesis adopts a similar idea. And, we propose to explore both the different models of static behavioral reflection and the implementation techniques that could serve to compile programs in static models and to take care of the static reflection that appear in less restrictive (dynamic) models.

4.3 Models of static behavioral reflection

Static behavioral reflection is not entirely new, but it is left implicit in existing models. In a sense, most of the work on the efficient implementation of reflective systems to date revolves around making more things static. We claim that it is important to make this concept explicit. Static models are only beginning to be explored and they will have an important impact on the design of powerful and yet potentially efficient reflective programming languages. To illustrate our point, we look at existing examples of static reflection.

Staticness in the 3-Lisp LSP

The 3-Lisp LSP is crucially based on the fact that the expression, environment and continuation of a level can be statically determined and thus pulled out of thin air dynamically without replaying the program from the beginning. This in turn depends on a lot of information about its processor that must be determined even at language design time, or at least at its implementation time. As mentioned before, it depends on the fact that all of the interpreters are known, and even that they are all the same in order not only to use the same reflective procedures in all levels, but also to compile them.

Compile time MOPs and open compilers

Recently, open compilers [LKRR92] with compile time MOPs [Chi95] have been proposed to provide reflective capabilities limited to modifications that can be expressed as extensions to the language's standard compiler. Open compilers are essentially compilers that are reified and that can be modified from within the language. A compile time MOP is the protocol provided to the user for implementing modifications. Despite their name, compile time MOPs do not imply in our view that everything is done prior to program execution, but rather leaves the door open to dynamic compilation. In the static

Binding-time signature	Implementation tools
baf $apply_S$ $\langle m_S, a_S \rangle$	static execution (side-effects are residuals)
baf $apply_S$ $\langle m_S, a_D \rangle$	static compiler generation + static compilation
baf $apply_S$ $\langle m_D, a_D \rangle$	static compiler generation + dynamic compilation
baf $apply_D$ $\langle m_D, a_D \rangle$	dynamic compiler generation + dynamic compilation

Figure 4: Binding times versus compiler technology

behavioral reflection perspective though, they must be limited to static modifications.

Partial evaluation based models

Another approach that has been suggested to implement behavioral reflection efficiently is partial evaluation based systems. Partial evaluation is a technique for automatic program specialization that has been developed essentially around the implementation of the first two Futamura projections [JGS93]:

$$\begin{aligned} \texttt{target} &= [\![\texttt{pe}]\!]\ \texttt{int}\ \texttt{p} \\ \texttt{compiler} &= [\![\texttt{pe}]\!]\ \texttt{pe}\ \texttt{int} \end{aligned}$$

The first projection says that partially evaluating an interpreter with respect to a known program "compiles" the program by removing the level of interpretation. The second says that partially evaluating the partial evaluator itself with respect to a specific interpreter yields a "compiler". In the reflection perspective, partial evaluation can provide the effect of compilation when the interpreter and the method are known. If only the interpreter is known, applying the second projection yields in theory a compiler that could be applied at run-time to compile methods. Explored by Danvy [Dan88], the link between partial evaluation and reflection has been exploited by a few researchers with more or less success (see §5.4).

4.4 Characterizing staticness in the *lookup* ∘ *apply* model

To help clarify things, we now characterize static behavioral reflection in the *lookup* ∘ *apply* model. Coarsely speaking, we have made explicit the *lookup* ∘ *apply* reflective towers by the following equation [MDC96]:

$$\textbf{baf}\ \langle apply_n, \ldots \langle apply_i, \ldots \langle apply_1, \langle m, a \rangle \rangle \ldots \rangle \ldots \rangle$$

which says that the basic apply function from the implementation (**baf**) executes the apply method at level n ($apply_n$), which in turn executes the apply method at level $n-1$ and so on. The level 1 apply method ($apply_1$) executes the method m on the arguments a. Consider the two-level tower:

$$\textbf{baf}\ apply\ \langle m, a \rangle$$

In this equation, each of *apply*, m or a can be either static or dynamic, but if the *apply* is dynamic, there is little interest in knowing if m or a is static. There are two major ways to know that something is static in a program: syntactically or by a binding-time analysis. For example, the syntax of applications in first-order functional languages forces the applied function to be known at compile-time. In higher-order languages, the applied function is the result of an expression, but a closure analysis [JGS93, §15.2] can show that, in particular cases, the result of this expression is in fact known at compile time. In general, syntactic staticness is much easier to deal with than the one brought to the fore by analysis, but it is also more restrictive.

Figure 4 illustrates the different interesting combinations of static and dynamic values in the above equation and gives a quite general "least upper bound" on the necessary compiler technology in each case. When everything is known statically, the result of the whole computation can be obtained at compile time, but this ideal case is indeed very rare. When both the interpreter (`baf`) and the method are known, we can partially evaluate the interpreter with respect to the method or statically generate a compiler from the interpreter and compile the method. In the third case, where only the interpreter is known, we can generate the compiler statically, but this compiler will have to be used

at run-time to compile methods. In the last and most challenging case, the interpreter is not known until run-time, which means that even the generation of a compiler must be done at run-time. They are least upper bounds because specific apply methods may necessitate much simpler implementation techniques. These observations scale up to the more general case of n-level reflective towers.

Digression on staticness and reflective towers

Notice the big difference, with respect to binding times, between 3-Lisp reflective towers and *lookup* ∘ *apply* ones. Although 3-Lisp reflective processors are all the same, and known prior to execution, the actual code run at each level is not completely known until run-time because reflective procedures can effectively add code into higher-level interpreters. In order to be known at compile time, not only the code of 3-Lisp interpreters must be static (which they are, by design), but the reflective procedures that may be called within them must also be determined statically. In the *lookup* ∘ *apply* case, there is no reflective procedure, so an interpreter is known as soon as its code is known.

5 Research directions

In this section, we first look at emerging techniques from the implementation of current advanced programming languages. We then suggest research directions to apply these techniques to the efficient implementation of behavioral reflection.

5.1 Catalogue of emerging techniques

The 3-Lisp LSP is now ten years old. Since then, a lot of exciting tools and techniques have emerged from the research on the implementation of reflective languages, but also dynamic languages such as Prolog, Common Lisp, Scheme, ML, CLOS, Smalltalk, Self, and others[7]. Several of them appear necessary for the efficient implementation of reflective languages, including:

- **Dynamic compilation**: in languages like Smalltalk [DS84] and Self [CUL89], it is now routine to compile methods at run-time. In both of these languages, methods are first compiled from the source language text to byte-codes at creation time. On demand, at call time, the byte-codes are compiled to native code to be run by the underlying processor. The native code is cached to be reused from invocation to invocation. Although run-time code generation has been in use since the 1940s, few language implementations, besides the dynamic languages Smalltalk and Self (and 3-Lisp!), are using this technique. Although it has fallen from favor because practices and technology changed, implementors now see it as a way to improve the performance of applications through the use of run-time information. *"The crux of the argument is that, although it costs something to run the compiler at run-time, run-time code generation can sometimes produce code that is enough faster to pay back the dynamic compile costs"* [KEH91]. Engler and Proebsting reports the construction of a machine-independent run-time code generator called DCG [EP94] while Leone and Lee apply run-time code generation to the optimization of ML [LL95].

- **Reified compilers**: in Smalltalk, the compiler from the source language to byte-codes is written in Smalltalk and available to the users. Although scarcely documented and publicized, it is perfectly possible, and even legitimate, to modify this compiler [Riv96]. Methods also have a field recording their *preferred compiler*, which may thus differ from the standard one. Open compilers [LKRR92] propose a methodical development of this technique (see §4.3).

- **Adaptative run-time systems**: families of compilers often share an important part of their run-time systems [Web92]. This kind of reuse does not only simplify the maintenance of compilers, it also paves the way to run-time adaptiveness of compilers to applications.

[7] Dynamic languages such as CLOS and Smalltalk now enjoy some fortune in the software industry, so more and more efficient commercial implementations are available. The technology developed by firms becomes an important factor of competitiveness, so it unfortunately remains secret. This is particularly the case for the CLOS MOP but also for Smalltalk.

Another example of adaptative run-time systems is the possiblity to choose among several alternative implementations for programming languages ADTs depending on profile data [Sam92].

- **First-class continuations and call/cc**: Scheme [CR91] provides first-class continuations, so a lot of effort has been directed towards their efficient implementation and the one of the related call/cc. Combined with Smalltalk reified stack frames [Riv96], the Scheme first-class continuations approach will serve as a strong basis for this crucial part of behavioral reflection dealing with control issues.

- **Partial evaluation**: Partial evaluation is a fairly general tool that specializes functions (programs) given a known subset of their actual parameters [JGS93]. The specialization engine of partial evaluation is essentially based on unfolding function calls. Applications of partial evaluation to reflective languages have been reported by Danvy [Dan88], Ruf [Ruf93] and more recently by Masuhara et al. [MMAY95] (see §4.3).

- **Binding-time analysis**: crucial to partial evaluation is the ability to statically determine the binding time of the value of each expression in a program, and to segregate among those known at compile time (static) from those known only at run-time (dynamic). In functional programming, good binding time analyses (BTA) [Con90] have been developed. The result of these analyses also enables various optimizations.

- **Other static analyses**: a lot of interest is currently raised by static analyses in general. Besides the constraint propagation approach developed in standard compiler technique and for object-oriented languages, abstract interpretation is now rapidly developing in functional and logic programming. Analyses based on type inference techniques are also more and more widely used [PS94].

- **Compiler generation**: automatic compiler generation is an important research theme since almost two decades now (look at [Lee89] for a survey). Despite the problems we still face, recent developments make us believe that it will play a substantial role in the future of reflective languages.

- **Monads**: introduced by Moggi [Mog89, Mog91] and popularized by Wadler [Wad90, Wad92], monads are categorical constructions that have been advocated for the modular extension of interpreters or denotational semantics. As descriptions of programming language semantics, monads are currently being investigated as a means to describe different programming language features and compose them in a general framework [LHJ95, Ste94, JD93, Fil94]. They are also considered for the automatic generation of efficient compilers [LH95].

5.2 Problem 1: Compilation of reflective towers

In the general case, as we have said before, interpreters in reflective towers implement complex levels of meta-interpretation. Each of them is actually a complete specification of a programming language that is used to implement the program of the level below. To get rid of them, an obvious solution is to provide a compiler that implements the corresponding language.

If we enlarge the scope of this observation to the whole reflective tower, we get an interesting peephole effect. For each level n in the reflective tower, the corresponding interpreter is a processor for the language $\mathcal{L}_{n-1}$. Hence, each level n can equally be implemented as a compiler from the language $\mathcal{L}_{n-1}$ to the language $\mathcal{L}_n$, itself implemented by the processor of level $n+1$. The whole tower can therefore be viewed as a multistage compiler from the language $\mathcal{L}_1$ in which the user program is written in the machine language in which the ultimate processor or the basic apply method is written.

The peephole effect suggests looking at three levels at a time, as through a peephole, and observe a compiling relationship between the language in which the level $n-1$ is written, implemented by

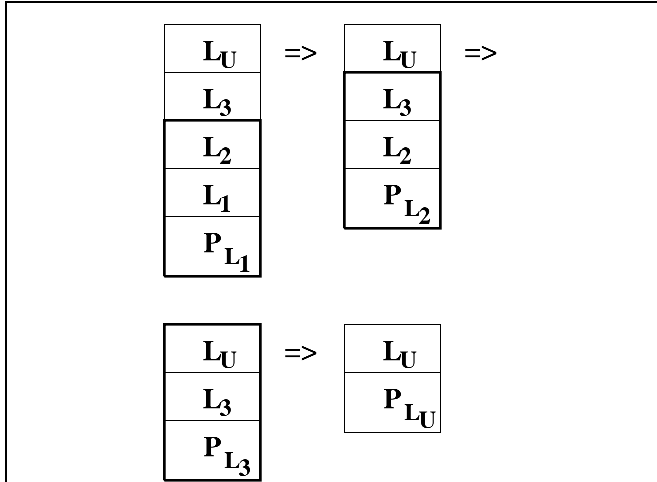

Figure 5: Peephole effect.

the level n interpreter itself written in $\mathcal{L}_{n-1}$. By moving the peephole up in the reflective tower, we can reveal a sequence of compilations transforming the program of level 0 into a program of level 1 and so on.

Hence, a reflective tower can be represented as a multistage compiler, which are more and more pervasive in high-level programming language implementations. The process is illustrated in Figure 5. In this example, a program **P** is run on a four-level tower. **P** is written in the language $\mathbf{L_1}$ implemented by the level 1 processor. The level 1 processor is written in the language $\mathbf{L_2}$, etc. By a series of compilations, **P** is transformed into a program written in $\mathbf{L_U}$, the language of the ultimate processor, which is directly implemented in hardware.

This kind of architecture is more and more common. We often see compilers producing intermediate programs in C that are then compiled using a C compiler. Serrano and Weiss [SW94] have even implemented an efficient CAML compiler by compiling from CAML to Scheme, then from Scheme to C and finally from C to machine language. This is exactly what we would like to do with reflective towers, modulo introspection which makes the process more complex.

5.3 Problem 2: Handling introspection and causal connection

Introspection behaves very differently in 3-Lisp like reflective towers and *lookup* ∘ *apply* ones. In 3-Lisp towers, reflective procedures should be written in the language of the level where they are applied[8] but their body will ultimately be executed at some higher-level, perhaps depending on the inputs to the program. In *lookup* ∘ *apply* ones, the introspective code is directly inserted in their level of application and is therefore written in the language of this level.

Compilation implies that the data structures and the code of one level will be progressively transformed, in a multistage process, into data structures of the lowest (machine) level. For example, in a Scheme to C compiler, Scheme environments, a latent concept at the Scheme level (but it would be explicit in a reflective language) are first transformed into C data structures and then into machine language ones. Danvy and Malmkjaer [DM88] indirectly observe this when they show that a value at level n has to be represented by some value of the level $n+1$ processor, which in turn must be represented by a value of the level $n+2$ processor, and so on. This means that when introspective code written in Scheme or in C, respectively accessing reified versions of the internal data structures, i.e., data structures in Scheme and in C, is compiled in machine language, it must access the corresponding machine language data structures.

Compiling usually means a loss of information. When a program is compiled, variable names are mapped to locations in memory and the names themselves disappear. Hence, a reflective computation written at the Scheme level to access the reified environment, must be compiled into machine code that accesses the real environment. This leads us to the following observation about the relationship between reflection, reification and compilation:

$$\begin{array}{rcl} \textit{Reflection} & = & \textit{Compilation} + \textit{Information} \\ \textit{Reification} & = & \textit{Information} + \textit{Decompilation} \end{array}$$

[8] This is not the common assumption when all the levels implement the same language. However, in a tower where each level implements a different language, it is not possible to have reflective procedures written in all of these languages. It appears more convenient to write them in the language of the level where they are applied.

For the same piece of data, reflection and reification are mediating between a higher-level representation accessible in terms of the higher-level language, and a lower-level one, which is accessible in the terms of the lower-level language. For example, the piece of data may be an environment represented as an a-list in Scheme but as an activation record in the lower-level representation. The a-list in Scheme maps variable names to values, while the activation record maps offsets to values. The first equation essentially says that when you reflect something, or install it in the lower-level, you must compile it but some information will be lost in the process. In the environment example, the variable names will be lost. The second equation essentially says that when you reify something, you must decompile a lower-level representation to create a higher-level one, but this process will need some information. Again in the environment example, you will need to supply the mapping between the variable names and the offsets in the activation record to return a complete a-list.

In fact, the need for this kind of information appears in several existing situations:

1. This kind of information is needed for source level debugging (the symbol table). This is not surprising: in source level debuggers, we have requests made in the source level model of computation that must be mapped onto the low-level model in which the program is run. These requests are very similar to reflective expressions.

2. It is also needed to manage Smalltalk reified activation records where a mapping between machine code addresses and the corresponding positions in the byte-codes must be kept for debugging purposes but also to transform them on demand from their native representation to an object one [DS84].

3. It is even more apparent in the debugging of heavily optimized code where it is difficult to keep track of the infromation because of code movement or even code elimination, among other things [BHS92].

4. Finally, it is also needed to decompile code on demand, a capability that exists in several systems like Smalltalk [DS84] and Self [HCU92].

In the static behavioral reflection case, the compiler would have this information at hand and would use it to compile the introspective code. In the dynamic behavioral reflection case, this information would have to be kept around at run-time to process the introspective code correctly.

5.4 Problem 3: Modifications to the semantics

The point of behavioral reflection is to modify the syntax, the semantics and the implementation of the language. Compiler extensions and automatic compiler generation appear as tools of choice to directly achieve such modifications. A modification to the syntax is achieved by modifying the lexer and the parser. Modifications to the semantics are implemented by modifications to the code generator. The run-time system can also be modified to match the needs of an application. Interestingly, in reflection the semantics of the language is incrementally modified, a process which suggests incremental compiler generation.

Compiler extensions are exemplified by the open compiler approach, which provides exactly the kind of incremental generation needed for reflection. Compared to automatic compiler generation, it appears as a proven approach, but forces the reflective programmers to construct modifications in terms of a complex compiling process. Automatic compiler generators start from a specification of the language that may appear more appropriate. Unfortunately, they produce less efficient compilers, and the specification they use may be as difficult to modify as the compiler itself. An interesting question is whether or not the fact that reflection incrementally modifies specifications for which there is already an existing compiler can help to construct automatic incremental compiler generators that would produce better compilers. In this context, it is interesting to assess the relative merits of the different formalisms of specifications in a reflection perspective.

We propose the following criteria of a good formalism for reflective purposes, which seem to catch the core issues:

1. The formalism must be easy to represent in the language, easy to understand and easy to modify, since this is the final end of reflection.
2. The formalism must lend itself to efficient compiler generation.
3. The formalism should make it possible to perform run-time modifications to be adaptable to dynamic behavioral reflection.

Besides compile time MOPs, which are rather new, the need to represent the formalism in the language as well as the need to easily provide causal connection have advocated the use of interpreters as formalism to specify the languages. Interpreters seem easy to represent, to understand and to modify. From a compilation point of view, interpreters can be transformed into compilers by partial evaluation as we have seen before. Unfortunately, this is actually difficult to achieve.

Part of the problem comes from the relative youth of the technique. Our own experiments [Dem94] have shown that currently available partial evaluators are often a little too fragile for production use. We have noticed that it takes great care to write interpreters and programs that perform well under partial evaluation. A deep knowledge of partial evaluation and of the particular partial evaluator under use is needed to achieve the goal of compiling away the levels of meta-interpretation. Other recent experiments, such as the one by Masuhara et al. [MMAY95], do not invalidate these remarks.

Part of the problem also comes from immoderate expectations put in partial evaluation. Partial evaluation performs a kind of compilation but Ruf [Ruf93] has pointed out that programs resulting from partial evaluation essentially stick to the implementation choices made by the interpreter. For example, if the interpreter uses a-lists to implement environments, partially evaluating it with respect to a program will yield a "compiled" program that still uses a-lists for its environments. Hence, unless a very precise model of the underlying machine leaks out to the interpreter, the "compiled" program will not achieve good performance. These remarks are corroborated by the study of Lee [Lee89] on automatic compiler generation from formal semantics. Unfortunately, if the interpreter deals with so many details, it may force reflective programmers to express their modifications to the semantics and implementation of the language in such a low-level model that most of them would probably be put off by this perspective.

Compiler generators have also been developed around formal semantics like denotational (see [Lee89]), action [Mos92, BMW92] and monadic semantics [LH95]. Denotational semantics is universally acknowledged as too monolithic to be extended easily. Action semantics has been developed in reaction to the monolithism of denotational semantics, but Liang and Hudak [LH95] argue that the theory of reasoning about programs in the action formalism is still weak and this may hamper program transformations, and therefore optimizations. Monads have attracted a lot of attention since the beginning of the nineties because of their powerful structuring capabilities, which have been used to develop modular interpreters where language features are represented as monads extending a vanilla monadic interpreter. Interestingly, the possibility for combining already existing monads to obtain an interpreter with mixed capabilities is also rapidly evolving [JD93, Ste94, LHJ95]. Finally, the most recent work in this domain tries to use monadic semantics to implement efficient compiler generators. Although still depending on partial evaluation, this approach seems to enable the development of specific program transformations [LHJ95, LH95], which, as we have argued, appear mandatory for the time being to obtain efficient compilers.

In our view, modular interpreters and monads, in conjunction with the idea of incremental compiler generation, are currently the most promising avenues to provide extensibility to reflective languages at an affordable cost in terms of performance.

5.5 Problem 4: Tackling dynamic behavioral reflection

Dynamic behavioral reflection happens when modifications depends directly or indirectly upon unknown inputs to the program. It doesn't mean however that no compilation can occur. Dynamic compilation techniques are already used in several languages, such as Smalltalk and Self. The problem, as stated earlier, is that the compilers will have to be generated dynamically, currently a major draw-

back. This fact leads to the following observation. In the same way standard languages face a compromise between the cost of compiling versus interpreting, which depends on the number of times a portion of code is executed, languages with dynamic behavioral reflection face a compromise between compiling versus interpreting a portion of code that depends on the number of times it is executed between successive reflective modifications that renders its recompilation mandatory.

In this perspective, we can foresee systems that mix interpretation and compilation, where it would be possible to compile part of the application and suspend the compilation when the necessary information will be known only at run-time. Suspended compilation will then be incrementally performed during run-time, perhaps depending on heuristics balancing the expected gain in performance versus the expected cost of performing this compilation.

Obviously, there is a compromise between the added flexibility that is provided by dynamic behavioral reflection, and performance. If it currently appears as pure fantasy for most applications, some of them may already prefer this kind of flexibility. For example, applications running for weeks on networks of heterogeneous computers may put a higher preference on the possibility to generate a new compiler to migrate processes towards newly added nodes than pure performance. Also, as incremental compiler generation will enhance, the balance between flexibility and performance will militate in favor of more practical applications.

6 Conclusion and perspectives

Because behavioral reflection implies the possibility of modifying the language semantics at run-time, most of the implementations tend to rely on interpretive techniques, which are easier to implement and by design more reactive to modifications. However, compiling is mandatory to get efficient languages and therefore make reflection of real interest. In this tutorial, we have proposed a new comprehensive approach to the efficient implementation of behavioral reflection based on the notion of binding times. We have introduced the distinction between static and dynamic behavioral reflection. We have argued that the static approach, while not covering all possible applications of behavioral reflection, is very powerful. It enables the compilation of reflective programs, sometimes using the available compiler techniques for modern dynamic languages.

In a more general setting, part of behavioral reflection is captured by the capability of extending the language's standard compiler. Compilers can either be constructed by hand or generated from some specification of the language syntax and semantics. Modifications can then be made either directly in the code of the compiler or by altering the specification and automatically generating a new compiler. In the field of compiler generation, the kind of specifications that have been used ranges from formal semantics (denotational, action or modular monadic semantics) to interpreters, either meta-circular or written in a low-level (assembly) language. The compiler itself can also be viewed as a low-level specification of the language semantics.

Unfortunately, a tension exists between high-level specifications, that eases the modifications, and the implementation, which needs a low-level (assembler) model to be run efficiently. The two can hardly coincide because low-level models are too detailed and tend not to be modular in terms of modifications. A modification made in low-level specification terms tends to spread all over the specification because of the coupling generally observed in these models. Open compilers tend to exhibit this kind of problem, but an interesting approach is to help the user in making modifications by providing default decisions, like in the Scheme open compiler of Lamping et al. [LKRR92].

An alternative is to use high-level models oriented towards the user but to automatically propagate the modifications from the high-level specification to the low-level model. We have assessed this approach in the reflection perspective and we have argued that incremental generation of compilers, either from extensible interpreters or from modular monadic semantics, currently appears as the most promising avenues.

Finally, we hope that the new approach to behavioral reflection as well as the thoughts expressed in this tutorial will help clarify the issues raised by its efficient implementation and that they will provide new research avenues to be explored by the

reflective language community in the near future.

7 Acknowledgments

The authors wish to thank Charles Consel for fruitful discussions on the monitoring semantics that guided us to the idea of static behavioral reflection.

References

[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.

[BHS92] Gary Brooks, Gilbert J. Hansen, and Steve Simmons. A new approach to debugging optimized code. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 1–11. *SIGPLAN Notices,* 27(7), July 1992.

[BMW92] D.F. Brown, H. Moura, and D.A. Watt. ACTRESS: An Action Semantics Directed Compiler Generator. In *Proceedings of the 4th International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 95–109. Springer-Verlag, 1992.

[Chi95] S. Chiba. A Metaobject Protocol for C++. *Proceedings of OOPSLA'95, ACM Sigplan Notices*, 30(10):300–315, October 1995.

[Con90] C. Consel. Binding Time Analysis for Higher Order Untyped Functional Languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 264–272, June 1990.

[CR91] W. Clinger and J. Rees, editors. *Revised⁴ Report on the Algorithmic Language Scheme*, November 1991.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of Self, a Dynamically-typed Object-Oriented Language Based on Prototypes. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10):49–70, October 1989.

[Dan88] O. Danvy. Across the Bridge between Reflection and Partial Evaluation. In D. Bjorner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation, IFIP*. Elsevier Science Publishers (North-Holland), 1988.

[Dem94] F.-N. Demers. Réflexion de comportement et évaluation partielle en Prolog. Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, 1994. Rapport technique #956.

[DM88] O. Danvy and K. Malmkjaer. Intensions and Extensions in a Reflective Tower. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 327–341, 1988.

[DM95] F.-N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, August 1995.

[dR90] J. des Rivières. The Secret Tower of CLOS. In *Informal Proceedings of the First Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP'90*, October 1990.

[dRS84] J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 331–347, August 1984.

[DS84] L.P. Deutsch and A.M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the ACM Symposium on Principles of Programming Languages '84*, pages 297–302. ACM Press, January 1984.

[EP94] Dawson R. Engler and Todd A Proebsting. DCG: An Efficient, Retargetable Dynamic Code Generator. In *Sixth International Conference on Architectural Support for Programming Languages a nd Operating Systems (ASPLOS-VI)*, pages 263–273, October 1994.

[Fer89] J. Ferber. Computational Reflection in Class Based Object-Oriented Languages. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10):317–326, October 1989.

[Fil94] Andrzej Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.

[FJ89] B. Foote and R. E. Johnson. Reflective Facilities in Smalltalk-80. *Proceedings of OOPSLA'89, ACM Sigplan Notices*, 24(10):327–335, October 1989.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 – The Language and its Implementation.* Addison-Wesley, 1983.

[Hal93] M. Halpern. Binding. In A. Ralston and E.D. Reilly, editors, *Encyclopedia of Computer Science*, page 125. Chapman & Hall, third edition, 1993.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 32–43. *SIGPLAN Notices,* 27(7), July 1992.

[JD93] Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report YALEU/DCS/RR-1004, Department of Computer Science, Yale University, December 1993.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[JM95] M. Jacques and J. Malenfant. ProtoReflex: un langage à prototypes avec réflexion de comportement. In A. Napoli, editor, *Actes de la conférence Langages et Modèles à Objets, LMO'95*, pages 75–91. INRIA-Lorraine, October 1995.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report 91-11-04, Department of Computer Science and Enginnering, University of Washington, November 1991.

[KHC91] A. Kishon, P. Hudak, and C. Consel. Monitoring Semantics: A Formal Framework for Specifying, Implementing and Reasoning about Execution Monitors. *Proceedings of PLDI'91, ACM Sigplan Notices*, 26(6):338–352, June 1991.

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

[Lee89] P. Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[LH95] Sheng Liang and Paul Hudak. Modular Denotational Semantics for Compiler Construction. Available by anonymous ftp from nebula.cs.yale.edu:/pub/yale-fp/papers/mod-sem-draft.ps.Z, September 1995.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.

[LKRR92] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An Architecture for an Open Compiler. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*, pages 95–106. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.

[LL95] Mark Leone and Peter Lee. Optimizing ML with Run-Time Code Generation. Technical Report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1995.

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

[MDC96] J. Malenfant, C. Dony, and P. Cointe. A Semantics of Introspection in a Reflective Prototype-Based Language. to appear in the journal Lisp and Symbolic Computation, 1996.

[MMAY95] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages using Partial Evaluation. *Proceedings of OOPSLA'95, ACM Sigplan Notices*, 30(10):300–315, October 1995.

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

[Mog89] E Moggi. Computational lambda-calculus and monads. In *Proceedings of the Logic in Computer Science Conference*, 89.

[Mos92] P.D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[PS94] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems.* John-Wiley & Sons, 1994.

[Riv96] F. Rivard. Smalltalk: a reflective language. In *Proceedings of the First International Conference on Computational Reflection, Reflection'96*, April 1996.

[Ruf93] E. Ruf. Partial Evaluation in Reflective System Implementations. In *Informal Proceedings of the Third Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA'93*, October 1993.

[Sam92] A.D. Samples. Compiler Implementation of ADTs Using Profile Data. In *Proceedings of the 4th Int'l Conference on Compiler Construction, CC'92*, volume 641 of *LNCS*, pages 73–87. Springer-Verlag, October 1992.

[SJ92] J. Wiseman Simmons II and S. Jefferson. Language Extensions via First-Class Interpreters. In A. Yonezawa and B. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Meta-Level Architecture*, pages 59–59. RISE (Japan), ACM Sigplan, JSSST, IPSJ, November 1992.

[Smi82] B.C. Smith. Reflection and Semantics in a Procedural Language. Technical Report 272, MIT Laboratory for Computer Science, 1982.

[Smi84] B.C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1984.

[Ste94] Guy L. Steele, Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 472–492, Portland, Oregon, January 1994.

[SW94] M. Serrano and P. Weiss. 1+1=1: An Optimizing CAML Compiler. In *Record of the 1994 ACM Sigplan Workshop on ML and its Applications*, pages 101–111, June 1994.

[USC+91] D. Ungar, R. Smith, C. Chambers, B.-W. Chang, and U. Hölzle. Special Issue on the Self programming language. *Lisp and Symbolic Computation*, (4), 1991.

[Wad90] Philip Wadler. Comprehending monads. In *1990 ACM Conference on Lisp and Functional Programming*, pages 61–78. ACM, ACM Press, June 1990.

[Wad92] P. L. Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM*, pages 1–14, 1992.

[Web92] C. Weber. Creation of a Family of Compilers and Runtime Environments by Combining Reusable Components. In *Proceedings of the 4th Int'l Conference on Compiler Construction, CC'92*, volume 641 of *LNCS*, pages 110–124. Springer-Verlag, October 1992.

[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.