

# Safe Reflection Through Polymorphism

Toon Verwaest      Lukas Renggli  
Software Composition Group  
University of Bern, Switzerland  
<http://scg.unibe.ch/>

## ABSTRACT

Code executed in a fully reflective system switches back and forth between application and interpreter code. These two states can be seen as contexts in which an expression is evaluated. Current language implementations obtain reflective capabilities by exposing objects to the interpreter. However, in doing so these systems break the encapsulation of the application objects. In this paper we propose safe reflection through polymorphism, *i.e.* by unifying the interface and ensuring the encapsulation of objects from both the interpreter and application context. We demonstrate a *homogeneous system* that defines the execution semantics in terms of itself, thus enforcing that encapsulation is not broken.

## Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming; D.3.3 [Programming Languages]: Language Constructs and Features

## General Terms

Design, Languages

## Keywords

Contexts, Reflection, Encapsulation, Virtual Machines

## 1. INTRODUCTION

Programming languages define high-level views over the execution semantics of a host system. Often these abstraction layers completely hide the internal semantics, and thus make it hard for application code to cross the barrier put up by the programming language between the high-level model and the low-level execution engine.

Crossing this barrier is especially important for building new types of languages. Existing language implementations might not always rely on the same assumptions as new languages, making it tedious for the new language to work

around those of the host system. For example, to add backtracking support to Smalltalk the designer has to manually realign Smalltalk's stack frames since this is the only way the execution semantics of Smalltalk can be changed.

Even more important is that the effort needed to work around decisions in a host system often imposes an overhead on the performance of the new language. An example of such a situation is where functional languages are implemented on top of the Java Virtual Machine. The JVM assumes that stack frames are needed for each call, while functional programming languages rely on recursion and thus require tail-call optimization.

Current mainstream interpreters internally consider the application code as data. By directly accessing this data to decide on how to proceed with the interpretation, they however break the encapsulation of the application. When an interpreter becomes more reflective applications can harm their own runtime by breaking the assumptions made by the interpreter.

In this paper we propose a bottom-up approach to reflection. By building a *homogeneous system*, *i.e.* defining a programming language's execution semantics in terms of itself, we ensure that encapsulation is not broken. Encapsulation enables reusability [2], thus the same interpreter can be used for different languages. To bootstrap the system, circular dependencies are broken by introducing objects that know how to perform the required low-level evaluation. We uniformly impose the same strong encapsulation upon all objects of the system. Interpretation and application contexts communicate with each other using the same mechanisms.

## 2. THE ENCAPSULATION PROBLEM

Maes [7] stated that for a programming language to be reflective, it has to guarantee causal connection between data representing (aspects of) the interpreter and the interpreter they represent. Current mainstream languages take a top-down approach to adding reflection. They start from being totally non-reflective and add reflective capabilities step by step by gradually adding application-level objects to the interpreter-level objects. This process gives language developers access to certain parts of the interpreter. However, this also implies there are two representations of the running interpreter and the objects it executes, one for the application level and one for the interpreter level. To ensure causal connection, a system to synchronize the two levels must be put in place.

Reflective languages typically allow applications to communicate with the interpreter through two main mecha-

nisms. In places where a specific type of object is expected, the system normally allows other objects to be passed in that

1. follow a certain meta-object protocol, or
2. that conform to a predefined memory layout.

The first kind of reflection is typically used by object-oriented systems with support for first-class functions. The code below is taken from PyPy [9], an object-oriented Python interpreter written in itself<sup>1</sup>.

```
def get_and_call_args(space, w_descr, w_obj, args):
    descr = space.interpclass_w(w_descr)
    # a special case for performance and
    # to avoid infinite recursion
    if type(descr) is Function:
        return descr.call_obj_args(w_obj, args)
    else:
        w_impl = space.get(w_descr, w_obj)
        return space.call_args(w_impl, args)
```

The code assumes that there are two types of functions: (1) *native functions* that are evaluated at the interpreter-level using `call_obj_args`, and (2) user defined *function objects* that are evaluated at the application-level using `call_args`.

This approach breaks the encapsulation of both interpreter and application level function objects. The interpreter directly accesses the internals of the function objects to test their type and to evaluate them either at the interpreter or application level. The interpreter uses the low-level abstractions of the application as data, but not the real abstractions provided by the application. This forces the language designer to be careful when implementing the class for native function objects: it is expected that the application-level `call_arg` is equally valid behaviour for the native function objects.

The second kind of reflection can for example be seen in Squeak [5], an open-source Smalltalk implementation. Squeak is a highly reflective system allowing developers to use any object as a class if the object follows a certain memory layout: the first slot must be a reference to the superclass, the second slot must be a reference to a dictionary of methods, and the third slot must contain an integer encoding various properties of the class such as the size of instances. Failing to correctly initialize one of these slots might cause the VM to crash.

In both cases we encounter a violation of the encapsulation of the respective objects. The duality in the representation causes problems that arise by not automatically forcing conformity with both representations. Even worse, the interpreter-level API of application-level objects is abused, possibly even from the application-level, to go around the encapsulation designed to protect objects from the outside world.

### 3. ENFORCING ENCAPSULATION

To preserve encapsulation across the meta-barrier, *i.e.* between code living in the interpreter context and code living in the application context, the interface between both types

<sup>1</sup>The code excerpt is taken from revision 65525 of <http://codespeak.net/svn/pypy/trunk/pypy/objspace/descroperation.py>, lines 74–81

of code has to be unified. Code from both contexts communicates through this unified interface. By providing a common reflective interface, encapsulation only has to be ensured at a single place. The language becomes reflective only through the meta-object protocol of the interpreter.

As an example of a unified view across the meta-barrier, we discuss our implementation of an object-oriented language built on top of Scheme called SCHEMETALK<sup>2</sup>. This new language combines the syntax of Scheme with message passing semantics of Smalltalk. Our prototype implementation uses closures to capture the state of objects. The following code shows how a class is defined in SCHEMETALK.

```
(define-class Person
  :superclass Object
  :instvars email
  :methods
  (setEmail! (arg) (self 'set-email! arg))
  (getEmail () (self 'get-email)))
```

Sending a message to an object in SCHEMETALK works by executing the lambda representing the object with a symbol identifying the method to execute and the arguments that should be passed to the method.

```
> (define john (Person 'new))
; sets John's email
> (john 'setEmail! "john@doe.com")
; retrieves the email
> (john 'getEmail)
"john@doe.com"
```

While our language has the same syntax as normal Scheme code, it is important to notice that SCHEMETALK is an object-oriented system. The code written in terms of SCHEMETALK belongs to the application context. All other types of Scheme code conceptually live in the interpreter context. The following is an example of Scheme code in the interpreter context:

```
(+ 39 2 1)
```

The interfaces provided by SCHEMETALK objects are the same as those provided by Scheme closures. Scheme closures are non-reflective thus the encapsulation of objects is guaranteed. In languages like C that do not provide a fail-safe encapsulation mechanism, we require a slightly different approach for new structures. Rather than letting the host-language ensure that no encapsulation can be broken, all the code written in the host-language must follow a strict discipline: from within an object only the object itself may be accessed directly. All other accesses must go over the provided abstractions.

Sending a message to an object in SCHEMETALK results in a lookup in the class hierarchy. Once a method object is found, the system sends the message `'execute` to the method object, with the given arguments. The class of a method is implemented using the same infrastructure as our previous model class:

```
; Application context
(define-class Method
  :superclass Object
  :instvars interp-code
  :methods
```

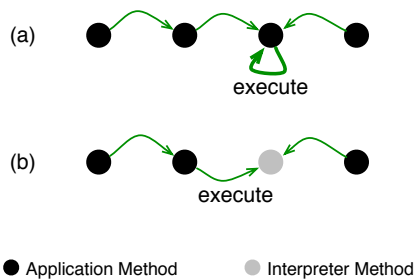
<sup>2</sup>The implementation along with documentation can be downloaded from <http://scg.unibe.ch/research/schemetalk>.

```

(initialize (interp-code)
  (self 'set-interp-code! interp-code))
(execute args
  (apply (self 'get-interp-code) args)))
; Interpreter context
(define (create-object class layout)
  (let ((instvars (create-instvars layout)))
    (define (self msg . args)
      (or (find-instvar instvars msg)
          (let ((method (class 'lookup msg)))
              (method 'execute args))))
      self))

```

Even if `self` is an object of the execution engine itself, it is defined using the concepts of the message send described in the application context. Furthermore, the code that defines the semantics for method execution itself depends on the semantics of the method execution.



**Figure 1: Method invocations.** (a) If a language is written in itself, at some point circular dependencies are imminent. (b) Applying polymorphism to the method invocation itself solves the problem.

As we explained in Section 2, in traditional systems this circular dependency is broken by not directly relying on objects in the application context. Methods would not be real application objects but rather specially tagged interpreter objects. The interpreter then checks if the looked up method is really an object internal to the interpreter, and if it is, the interpreter natively executes its code. Reflective interpreters would allow applications to insert custom types of methods by falling back to normal message sends in the case that the retrieved object was not an interpreter-level object.

This way of building a system is not object-oriented. In a fully object-oriented system the different types of behaviour would be decided based on the polymorphic behaviour of the retrieved object. Instead the previously described way of interpreting breaks the encapsulation of the object by directly checking its runtime type.

To break the circular dependency in an object-oriented fashion as shown in Figure 1, the VM must ensure that objects from the application context support the same interface as objects from the interpretation context. This avoids the need for the VM to rely on type information to know how to execute code.

In Scheme we can easily build code in the interpreter context which uses the same interface as `SCHEMETALK` objects. This concept is well-known as *dispatch objects* [1]. Dispatch-objects introduce object-orientation to Scheme by adding objects which directly understand a set of messages, *i.e.* they do not rely on other objects for method invocation.

```

(define (method-class interp-code)
  (letrec ((self (lambda (msg . args)
                   (case msg
                     ((execute) (apply interp-code args))
                     (else
                      ; Remember that Method is the class
                      ; for methods written in SchemeTalk.
                      (let ((method (Method 'lookup msg)))
                          (method 'execute args)))))))
          self))

```

Dispatch-objects as shown in the previous code allow the interpreter to generate objects which directly understand a very limited number of messages. In this example, methods only understand `execute` directly, *i.e.* without any need for further evaluation in the interpreter or application context. In the alternative path, whenever the message is not `execute`, the normal lookup pattern is taken. By relying on dispatch objects we have successfully broken the circular dependency circle while still allowing the objects to receive all sorts of messages which are defined in the class. In effect we have moved the test which the interpreter needs to decide how to execute the code, from outside the method object to the inside of the object itself.

In contrast to traditional reflective systems our implementation is safe by design. Since the interface of interpreter- and application-level objects is unified, applications can directly communicate with the interpreter's objects through the same interface as other objects. This avoids duality and related synchronization problems. Secondly, since objects never break the encapsulation of other objects, the interpreter-level objects cannot accidentally read raw memory by making wrong assumptions about the objects it is handling. Properly implemented encapsulation enforces the interpreter to handle all objects safely.

## 4. RELATED WORK

Lorenz *et al.* also identified an encapsulation problem related to reflection [6]. They demonstrate that Java applications often rely on interfaces as the sole source for meta-information. This limits the reuse and makes it difficult to re-target applications to use a different source of meta-information. By providing *pluggable reflection* the actual source of meta-information can be plugged in at compile- or load-time. While they identify and solve the encapsulation problem on the application-level, their approach does not enforce the interpreter to follow the same strategy. As presented in our paper, we enforce a pluggable interpretation strategy at application and interpretation level.

Bracha *et al.* argue that meta-level functionality should be implemented separately from the base-level functionality, using objects known as *mirrors* [3]. They identify three design principles for reflection: *encapsulation*, *stratification* and *structural correspondence*. Stratification means that the meta-level functionality should be separated from the base functionality and structural correspondence ensures that the meta-level functionality corresponds to the structure of the language they manipulate. With our approach there is no need for stratification, as the complete system including the interpretation layer is implemented in terms of encapsulated entities.

Gybels *et al.* combine reflection and inter-language meta-programming into inter-language reflection [4]. They argue that an unification between interpreter-level representations

of the different languages is required. To unify the interfaces between objects they propose to wrap the internal representations. This approach however does not enforce the inter-language reflection to be safe. Any of the languages might expose its own unsafe reflection to all languages, making all languages involved unsafe.

Schärli *et al.* propose object-oriented encapsulation policies for dynamically typed languages [10]. The encapsulation deficiency of a highly reflective system is fixed by assigning access policies to objects, thus objects can be protected from each other depending on the context they are used in. Contrary to their approach, our system is built from the ground up to support safe reflection, and not patched post-mortem to turn an inherently insecure system safe.

Piumarta *et al.* propose a minimal message based object model [8]. The only primitive operation is the bind operation, that searches in a cache for executable code. In case of a cache miss, a method lookup at the application level is performed and expected to return a pointer to executable code. The system is bootstrapped by filling the caches with a minimal set of methods. This gives a mutable object model that permits the implementation of various lookup strategies, however it forces the application layer to deal with low level concepts such as executable code pointers and thus breaks encapsulation completely.

## 5. CONCLUDING REMARKS

In this paper we have identified an encapsulation problem between code running in application and interpreter level. This limits the possible reuse of interpreter code. In our presented solution we ensured encapsulation by unifying the interface between objects from the interpreter and the application context. We first built the system fully in terms of itself to then break circular dependencies by introducing encapsulation-preserving objects in the interpreter context that are polymorph to the application's objects.

While the implementation of SCHEMETALK only demonstrated the integration of methods into a language, the technique described in this paper should be applied on all levels of any *context-aware language*. The decision on how to respond to a changing context should be handled by the object in question, and not by hardcoded choices made by the interpreter.

Even though the current implementation is safe, we run on top of a mostly non-reflective system making our performance suffer. To gain performance we would have to bring our system to the level of the host language. This can only be done from within a language if it is reflective. To bootstrap such an environment, the first incarnation has to be geared towards the lowest system available, namely the actual hardware.

## 6. ACKNOWLEDGMENTS

We thank Camillo Bruni, Tudor Gîrba, Adrian Kuhn, Oscar Nierstrasz and John Plaice for their feedback on this paper. We gratefully acknowledge the financial support of the

Swiss National Science Foundation for the project “Bringing Models Closer to Code” (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010).

## 7. REFERENCES

- [1] N. Adams and J. Rees. Object-oriented programming in scheme. In *Conference Record of the 1988 ACM Conference on Lisp and Functional Programming*, pages 277–288, Aug. 1988.
- [2] K. Auer. Reusability through self-encapsulation. In *Pattern languages of program design*, pages 505–516. ACM Press/Addison-Wesley Publishing Co., 1995.
- [3] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, *ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.
- [4] K. Gybels, R. Wuyts, S. Ducasse, and M. D'Hondt. Inter-language reflection — a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, July 2006.
- [5] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.
- [6] D. H. Lorenz and J. Vlissides. Pluggable reflection: decoupling meta-interface and implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Washington, DC, USA, 2003. IEEE Computer Society.
- [7] P. Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, Dec. 1987.
- [8] I. Piumarta and A. Warth. Open reusable object models. Technical report, Viewpoints Research Institute, 2006. VPRI Research Note RN-2006-003-a.
- [9] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium, OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, New York, NY, USA, 2006. ACM.
- [10] N. Schärli, A. P. Black, and S. Ducasse. Object-oriented encapsulation for dynamically typed languages. In *Proceedings of 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'04)*, pages 130–149, Oct. 2004.