Safe Reflection Through Polymorphism

Toon Verwaest and Lukas Renggli

Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch/

Presented by Javier Cubo (University of Málaga, Spain)

CASTA 2009
August 24, 2009
Amsterdam

Agenda

- Introduction
- The Encapsulation Problem
- Enforcing Encapsulation
- Concluding Remarks

Introduction

- **Programming languages** define high-level views over the execution semantics of a host system
  - these abstractions layers hide the internal semantics

- **Crossing this barrier** is 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
    - backtracking support to Smalltalk → realign Smalltalk's stack frames
  - imposing an overhead on the performance of the new language
    - functional lang implemented on JVM top → JVM assumes stack frames needed for each call, while functional langs rely on recursion (tail-call optimization)

Introduction

PROBLEM

It is hard for application code to cross the barrier between the high-level model and the low-level execution engine

- **Crossing this barrier** is 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
    - backtracking support to Smalltalk → realign Smalltalk's stack frames
  - imposing an overhead on the performance of the new language
    - functional lang implemented on JVM top → JVM assumes stack frames needed for each call, while functional langs rely on recursion (tail-call optimization)

Introduction

- **Current mainstream interpreters** internally consider the application code as data
  - by directly accessing this data to decide on how to proceed with the interpretation → the encapsulation of the application is broken
  - interpreter more reflective → appl breaks the interpreter assumptions

- **Homogeneous** system
  - lang's execution semantics in terms of itself→**encapsulation not broken**
    - by **unifying the interface** between objects from **the interpreter and the application context**

- **Characteristics**
  - encapsulation enables **reusability** → same interpreter used for diff langs
  - to bootstrap the system → **circular dependencies are broken**
    - by introducing objects that know how to perform required low-level evaluation
  - imposing the same **strong encapsulation** upon all objects of the system
  - **interpretation and application contexts** communicate with each other
    - by using the same mechanisms

Introduction

- **Curr** ... applic ...
  - ... with the ... n
  - ... umptions

PROBLEM
The encapsulation of the application and the assumptions of the interpreter are broken

PROPOSAL
Bottom-up approach to reflection

- **Homogene** ...
  - lang's e ... **sulation not broken**
    - by ... **interpreter and the ap** ...

PROPOSAL
Bottom-up approach to reflection

- **Characteristics**
  - encapsulation enables **reusability** → same interpreter used for diff langs
  - to bootstrap the system → **circular dependencies are broken**
    - by introducing objects that know how to perform required low-level evaluation
  - imposing the same **strong encapsulation** upon all objects of the system
  - **interpretation and application contexts** communicate with each other
    - by using the same mechanisms

The Encapsulation Problem

- **Current mainstream languages** take a **top-down** approach to add **reflection**
  - **adding** application-level objects to the interpreter-level objects
- **Two representations** of running interpreter and their objects
  - **application level** and **interpreter level**
    - to ensure **causal connection** → a system synchronizing the two levels must be put in place
- Reflective languages allow applications to communicate with the interpreter through two main mechanisms
  - **meta-object protocol**
  - **predefined memory layout**

The Encapsulation Problem

Meta-object Protocol

- **PyPy**: object-oriented Python interpreter written in itself

```
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)
```

- **Two types** of functions
  - *native functions* evaluated at interpreter-level → `call_obj_args`
  - user *function objects* evaluated at application-level → `call_args`

- **Breaks the encapsulation** of both interpreter and application level function objects

The Encapsulation Problem

Predefined Memory Layout

- **Squeak**: an open-source Smalltalk implementation
  - highly **reflective** system allowing developers to use any object as a class if the object follows a certain memory layout
    - first slot → reference to the superclass
    - second slot → reference to a dictionary of methods
    - third slot → contain an integer encoding various properties of the class (size of instances)

The Encapsulation Problem

- In both previous cases → **violation of the encapsulation** of the objects
  - the duality in representation causes problems
    - by not forcing conformity with both representations
  - the interpreter-level API of application-level objects abused
    - even from the application-level to go around encapsulation designed to protect objects from the outside world

Enforcing Encapsulation

- **Unifying interface** between code of the interpreter and application contexts
  - **preserving encapsulation** across the meta-barrier

- Code from both contexts **communicates through this unified interface**

- By providing a **common reflective interface** → encapsulation ensured at a single place
  - language becomes reflective through the meta-object protocol of the interpreter

Enforcing Encapsulation

- **SchemeTalk**: object-oriented language built on top of Scheme
  - combines syntax of Scheme with message passing semantics of Smalltalk
  - prototype implementation uses closures to capture the state of objects

- **Class**

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

- **Sending a message**

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

- **Scheme code in the interpreter context**

```
(+ 39 21)
```

Enforcing Encapsulation

- **Interfaces** provided by SchemeTalk objects are the same as those provided by Scheme closures
  - **non-reflective** → encapsulation of objects guaranteed
- **Sending a message** to an object in SchemeTalk → a lookup in the class hierarchy
  - once a method object is found → system sends the message `'execute` to the method object with the args
- The **class of a method** is implemented using the same infrastructure as the previous model class

Enforcing Encapsulation

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

Enforcing Encapsulation

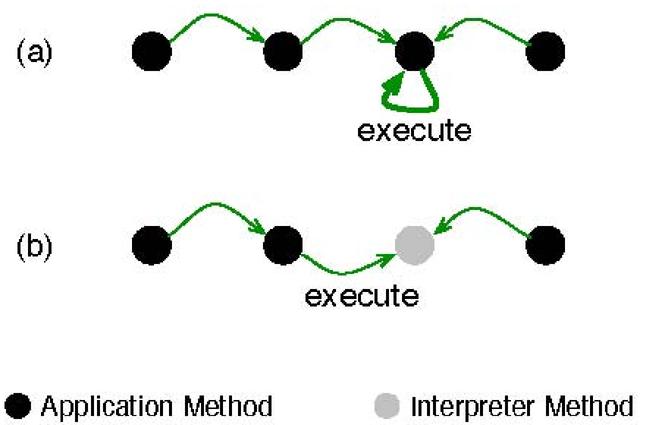
- ***self*** object of the execution engine → it is defined using concepts of the message send of the application context
  - code defining the semantics for method execution itself depends on the semantics of the method execution

- In **traditional systems** this circular dependency is broken by not directly relying on objects in the application context
  - methods would be tagged interpreter objects
  - interpreter checks if the looked up method is an object internal to the interpreter → it natively executes its code
  - reflective interp would allow appls to insert custom methods
    - by falling back to normal message sends in case the retrieved object was not an interpreter-level object

Enforcing Encapsulation

- This way of building a system is **not object-oriented**
  - in OO system the behaviour types would be decided based on the polymorphic behaviour of the retrieved object
  - instead this way breaks the encapsulation of the object by directly checking its runtime type

- To **break the circular dependency** in an OO fashion
  - VM must ensure that objects from application context support the same interface as objects from interpretation context (**polymorphism**)



Enforcing Encapsulation

- Scheme easily builds code in interpreter context using the same interface as SchemeTalk objects
  - **dispatch-objects** introduce OO to Scheme
    - by adding objects which directly understand a set of messages

```
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))
```

Enforcing Encapsulation

- In contrast to traditional reflective systems this implementation is **safe by design**
  - unified interface of interpreter and application level objects
    - applications directly communicate with interpreter's objects through the same interface as other objects
    - by avoiding duality and related synchronization problems
  - objects never break encapsulation of other objects → the interpreter-level objects **cannot read raw memory**
    - by making wrong assumptions about the handled objects
  - properly implemented encapsulation enforces the interpreter to handle all objects safely

Concluding Remarks

- An **encapsulation problem** between code running in application and interpreter level has been identified
  - that limits the reuse of interpreter code
- The presented approach ensures the encapsulation by **unifying the interface** between objects from interpreter and application contexts
  - system built in terms of itself breaking the circular dependencies
    - by preserving encapsulation of interp context objects polymorph to appl context ones
- ShemeTalk implementation only demonstrated the integration of methods into a language
  - the proposed technique **should be applied on levels** of any context-aware lang
- Current implementation of this approach is run on top of a mostly non-reflective system making the performance suffer
  - to gain performance → bring the system to the level of the host language
    - which can only be done from within a language if it is reflective
  - to bootstrap such an environment → work with the lowest system available (HW)

Thanks a lot for your attention!

- Congratulations to the authors for this work!