Building run-time analysis tools by means of pluggable interpreters

Michel Tilman

System Architect, Unisys Belgium

mtilman@acm.org

http://users.pandora.be/michel.tilman

ESUG'2000 Summer School

Southampton

August 28, 2000

Contents

- Introduction
- Method wrappers
- Interpreter
- Compatiblity issues
- Demo
- References

Introduction

- Need for tools
  - Coverage
    - Methods, blocks, arguments, variables
  - Run-time typing information
  - …
- Current tools
  - Smalltalk reflection
  - Low-level, hard to understand, operational
    - Bytecodes, processes, ….

Introduction

- Smalltalk language
  - Simple semantics
- Interpreter
  - Declarative feel
  - Easier to understand
  - Selective
    - Per method
    - Embedded interpreter

Method wrappers

- Intercept method activations
  - Smalltalk reflection
    - Localized ‘tricks’
  - Easy to extend
- Usage
  - Before-after methods
  - Build interaction diagrams
  - ...
  - Activate interpreter

Interpreter

- Recursive descent
  - Abstract grammar
    - Parse tree
  - Eval method
- Smalltalk = run-time engine
  - Garbage collection
  - Primitives
    - Include methods supporting interpreter implementation
  - Unwinding stack
  - ….

Abstract grammar

- **Expression tree**

```
Expression ()
        Assignment ('variable' 'value')
        Code ('arguments' 'temporaries')
                Block ('level')
                Method ('mclass' 'selector')
        Literal ('value')
        MessageExpression ('receiver' 'selector' 'arguments' 'methodCache')
        PseudoVariable ('isSuper')
        Return ('value')
        Variable ()
                InstanceVariable ('index' 'class')
                LocalVariable ('name' 'level' 'index')
                SharedVariable ('binding')
```

Contexts

Contexts

```
CodeContext ()
        BlockContext ('outerContext' 'level')
        MethodContext ('receiver' 'method' 'returnHandler')
```

Block closures

```
BlockClosure ('block' 'outerContext')
```

Evaluation

Method wrappers

```
valueWithReceiver: anObject arguments: arrayOfObjects
        "Evaluate the method by invoking the interpreter."

        ^ self methodExpression valueWithReceiver: anObject arguments: arrayOfObjects
```

Method expressions

```
valueWithReceiver: anObject arguments: arrayOfObjects
        ”Create a suitable context. Evaluate the method with given receiver and arguments
in this context. Answer the result.”

        | context |

        context := self contextWithReceiver: anObject arguments: arrayOfObjects.
        ^ self valueInContext: context
```

Evaluation

Method expressions

```
contextWithReceiver: anObject arguments: arrayOfObjects
        ”Answer a context for evaluating the receiver. Install a return handler block.”

        ^ MethodContext on: self receiver: anObject arguments: arrayOfObjects
returnHandler: [ :value | ^ value ]
```

Method (Code) expressions

```
valueInContext: aMethodContext
        ”Evaluate the statements for side effects. Answer the result of the last evaluation.”

        | answer |

        self statements do: [ :stat | answer := stat eval: aMethodContext ].
        ^ answer
```

Evaluation

Literal expressions

```
eval: aContext
        ”Answer the value.”

        ^ value
```

Pseudo-variables (“self”, “super”)

```
eval: aContext
        ”Answer the receiver.”

        ^ aContext receiver
```

Evaluation

Shared variables

```
eval: aContext
        ”Answer the value bound to the variable.”

        ^ binding value
```

Shared variables

```
bind: anObject context: aContext
        ”Bind the object to the variable.”

        binding value: anObject.
        ^ anObject
```

Evaluation

Instance variables

```
eval: aContext
        ”Answer the value bound to the variable.”

        ^ aContext receiver instVarAt: index
```

Instance variables

```
bind: anObject context: aContext
        ”Bind the object to the variable.”

        ^ aContext receiver instVarAt: index put: anObject
```

Evaluation

Local variables

```
eval: aContext
        ”Answer the value bound to the variable.”

        ^ (aContext contextAt: level) at: index
```

Local variables

```
bind: anObject context: aContext
        ”Bind the object to the variable.”

        ^ (aContext contextAt: level) at: index put: anObject
```

Evaluation

Assignment expressions

eval: aContext

”Bind the value to the variable. Answer the value.”

^ variable bind: (value eval: aContext) context: aContext

Return expressions

eval: aContext

”Evaluate the value expression. Return the answer as the return value of the method that ‘defines’ this return expression.”

aContext returnHandler value: (value eval: aContext)

Evaluation

Block expressions

```
eval: aContext
        ”Build a block closure for this context. Answer the closure.”

        ^ BlockClosure block: self outerContext: aContext
```

Executing block closures

```
value
        ”Execute the block (closure) without arguments. Answer the result.”

        ^ block numTemps = 0
                ifTrue: [ block valueInContext: outerContext ]
                ifFalse: [ block valueInContext: (BlockContext outerContext:
outerContext level: block level + 1 size: numTemps)]
```

Evaluation

- **Wrapping block closures in ‘native’ closures**

```
asNativeBlockClosure
        ”Wrap the receiver in a native block closure. Note: this is self-modifying code.”

        | numArgs |

        (numArgs := block numArgs) = 0 ifTrue: [ ^ [ self value ]].
        numArgs = 1 ifTrue: [ ^ [ :arg1 | self value: arg1 ]].
        numArgs = 2 ifTrue: [ ^ [ :arg1 :arg2 | self value: arg1 value: arg2 ]].
        numArgs = 3 ifTrue: [ ^ [ :arg1 :arg2 :arg3 | self value: arg1 value: arg2 value: arg3 ]].
        numArgs = 4 ifTrue: [ ^ [ :arg1 :arg2 :arg3 :arg4 | | args |
                                                        args := Array new: 4.
                                                        args at: 1 put: arg1.
                                                        args at: 2 put: arg2.
                                                        args at: 3 put: arg3.
                                                        args at: 4 put: arg4.
                                                        self valueWithArguments: args ]].

        self generateUpTo: numArgs.
        ^ self asNativeBlockClosure
```

Evaluation

Message expressions

```
eval: aContext
        ”Evaluate receiver and arguments. Do the actual lookup and execute the method found.”
        | rcvr args behavior |

        rcvr := receiver eval: aContext .
        args := arguments collect: [ :arg | arg eval: aContext ].
        behavior := receiver isSuper
                                ifTrue: [ aContext methodClass superclass ]
                                ifFalse: [ rcvr class ].
        ^ self perform: selector receiver: rcvr arguments: args class: behavior
```

Evaluation

Message expressions

perform: sel receiver: rcvr arguments: args class: aBehavior

”Retrieve the method for given selector, starting the lookup in the class. Execute the method and return the result if successful. Otherwise fail with a #doesNotUnderstand: message. Note that executing a #perform: in the false branche instead of executing the method will acutally be faster. The lookup is only relevant in so far we are interested in keeping track of the different method invocations at the call site.”

```
| method |

^ (method := self lookupMethod: sel in: aBehavior) isNil
        ifTrue: [ rcvr doesNotUnderstand: (Message selector: sel arguments: args)]
        ifFalse: [ method valueWithReceiver: rcvr arguments: args ]
```

Evaluation

Message expressions

```
lookupMethod: sel in: aBehavior
        ” Search for an appropriate method. First perform a lookup in the local cache.
Upon failure do a regular lookup and cache the result.”

        ^ methodCache at: aBehavior ifAbsentPut: [ self basicLookupMethod: sel in:
aBehavior]
```

Message expressions

```
basicLookupMethod: sel in: aBehavior
        "Search for a method in the class hierarchy. Answer nil upon failure."

        ^ (array := aBehavior findSelector: sel) isNil
                                ifTrue: [  nil ]
                                ifFalse: [ array at: 2 ]
```

Compatibility issues

- BlockClosure
  - copied values, ...
- Pseudo-variables
  - thisContext
- No saveguards
  - Primitives
  - ‘Kernel’ methods

Demo

- Tools
  - Coverage tool
    - Updating number of activations in #eval: methods
  - Gathering run-time typing
    - Remembering types in #eval: methods
      - Variables
      - Method invocations at call site
- Fine-grained
  - Disjoint lexical occurrences of same variable

References

- **Method wrappers**
  - http://st-www.cs.uiuc.edu/~brant