Runtime Bytecode Transformation for Smalltalk

Marcus Denker (University of Berne, Switzerland)
Stéphane Ducasse (University of Berne, Switzerland)
Éric Tanter (University of Chile)

Motivation

- Runtime transformation needed for
  - Adaptation of running systems
  - Tracing / debugging
  - New language features (MOP, AOP)

ByteSurgeon

- Library for bytecode transformation in Smalltalk
- Full flexibility of Smalltalk: Runtime
- Provides high-level API
- For Squeak, but portable

Why Bytecode?

- No need to change the VM
- No source needed
- Other languages possible
- Performance

Examples

Counts the number of Bytecodes:

```
InstrCounter reset.
Example instrument: [ :instr  | InstrCounter increase ]
```

Counts the number of Sends:

```
InstrCounter reset.
Example instrumentSend: [ :instr  | InstrCounter increase ]
```

Introspection:

```
(Example>>#aMethod) instrumentSend: [ :send |
                        Transcript show: send selector printString]
```

Transformations

Modification: inlining of code

insertBefore:, insertAfter:, replace:

```
(Example>>#aMethod) instrumentSend: [ :send |
        send insertAfter: 'InstrCounter increase']
```

```
(Example>>#aMethod) instrumentSend: [ :send |
        send insertAfter: 'Transcript show:', send selector printString].
```

User-defined Variables

Concatenate strings:

```
(Example>>#aMethod) instrumentSend: [ :send |
    send insertAfter: 'Logger logSend:' , send selector printString]
```

Poor man's quasi-quoting:

```
(Example>>#aMethod) instrumentSend: [ :send |
    send insertAfter: 'Logger logSend: <: #sel> ']
        using: { #sel -> send selector }
```

Metavariables

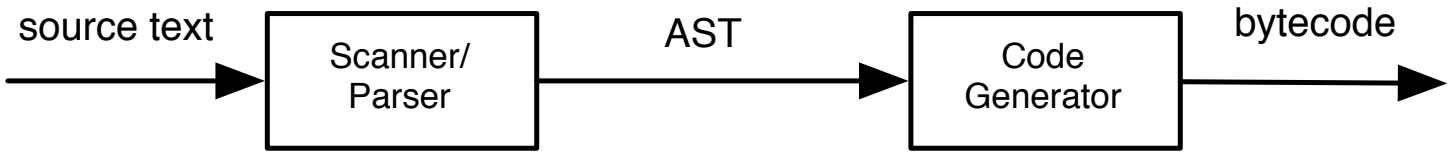
- Goal: extend a send with after logging
- Problem: How to access receiver and args?
- Solution: metavariables

```
Example instrumentSend: [ :s |
    m insertAfter: 'Logger logSendTo: <meta: #receiver> '
]
```

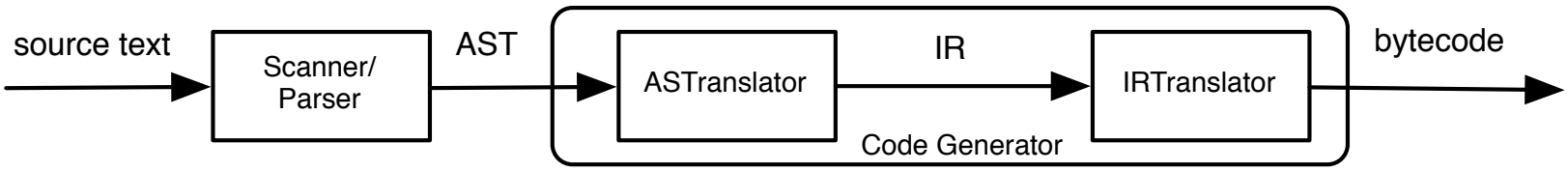
#receiver, #arguments, #argn, #result....
#value, #newvalue

Implementation

The Smalltalk Compiler:



Backend of NewCompiler:



Examples

- MethodWrapper with ByteSurgeon
  - 40 lines of code
  - Slower on install (factor 6)
  - Faster on execution (factor 5.3)

- A very simple MOP (MetaObject Protocol)

Simple MOP

- Goal: Control instance variable access
- Code for a trace metaobject:

```
TraceMO>>instVarRead: name in: object
    | val |
    val := object instVarNamed: name.
    Transcript show: 'var read: ', val printString; cr.
    ^val.

TraceMO>>instVarStore: name in: object value: newVal
    Transcript show: 'var store: ', newVal printString; cr.
    ^object instVarNamed: name put: newVal.
```

MOP

```
MOP class >>install: mop on: aClass
    | dict |
    dict := Dictionary newFrom: #mo -> mop.
    aClass instrumentInstVarAccess: [:instr |
        dict at: #name put: instr varname.
        instr isRead
            ifTrue: [instr replace: '<: #mo> instVarRead: <: #name> in: self'
                                using: dict ]
            ifFalse: [instr replace: '<: #mo> instVarStore: <: #name>
                                        in: self  value: <meta: #newvalue> '  using: dict]
    ]
```

A simple MOP in <10 lines

Benchmark

Recompilation Vs. ByteSurgeon:

	time	factor
Bytesurgeon	4817	1
standard compiler	9760	2.03
new compiler	33611	6.98

Future Work

- Improvements
  - AST vs. Bytecode
- Applications of ByteSurgeon
  - Geppetto MOP
  - Omniscient Debugger