Working with Bytecodes: IRBuilder and ByteSurgeon

Marcus Denker
Reasons for working with Bytecode

• Generating Bytecode
  - Implementing compilers for other languages
  - Experimentation with new language features

• Bytecode Transformation
  - Adaptation of running Systems
  - Tracing / Debugging
  - New language features
Overview

1. Introduction to Squeak Bytecodes
2. Generating Bytecode with IRBuilder
3. Introduction to ByteSurgeon
The Squeak Virtual Machine

• From last lecture:
  - Virtual machine provides a virtual processor
  - Bytecode: The ‘machine-code’ of the virtual machine
  - Smalltalk (like Java): Stack machine

• Today:
  - Closer look at Squeak bytecode
Bytecode in the CompiledMethod

- CompiledMethods format:

<table>
<thead>
<tr>
<th>Header</th>
<th>Number of temps, literals...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literals</td>
<td>Array of all Literal Objects</td>
</tr>
<tr>
<td>Bytecode</td>
<td></td>
</tr>
<tr>
<td>Trailer</td>
<td>Pointer to Source</td>
</tr>
</tbody>
</table>

(Number>>#asInteger)inspect
Example: Number>>asInteger

• Smalltalk code:

```smalltalk
Number>>asInteger
    "Answer an Integer nearest the receiver toward zero."

    ^self truncated
```

• Symbolic Bytecode

```smalltalk
9 <70> self
10 <D0> send: truncated
11 <7C> returnTop
```
Example: Step by Step

- 9 <70> self
  - The receiver (self) is pushed on the stack
- 10 <D0> send: truncated
  - Bytecode 208: send literal selector 1
  - Get the selector from the first literal
  - start message lookup in the class of the object that is top of the stack
  - result is pushed on the stack
- 11 <7C> returnTop
  - return the object on top of the stack to the calling method
Squeak Bytecodes

- 256 Bytecodes, four groups:
  - Stack Bytecodes
    - Stack manipulation: push / pop / dup
  - Send Bytecodes
    - Invoke Methods
  - Return Bytecodes
    - Return to caller
  - Jump Bytecodes
    - Control flow inside a method
Stack Bytecodes

- Push values on the stack, e.g., temps, instVars, literals
  - e.g. 16 - 31: push instance variable
- Push Constants (False/True/Nil/1/0/2/-1)
- Push self, thisContext
- Duplicate top of stack
- Pop
Sends and Returns

• Sends: receiver is on top of stack
  - Normal send
  - Super Sends
  - Hard-coded sends for efficiency, e.g. +, -

• Returns
  - Return top of stack to the sender
  - Return from a block
  - Special bytecodes for return self, nil, true, false (for efficiency)
Jump Bytecodes

- Control Flow inside one method
- Used to implement control-flow efficiently
- Example:

```plaintext
^ 1<2 ifTrue: ['true']
```

9 <76> pushConstant: 1
10 <77> pushConstant: 2
11 <B2> send: <
12 <99> jumpFalse: 15
13 <20> pushConstant: 'true'
14 <90> jumpTo: 16
15 <73> pushConstant: nil
16 <7C> returnTop
What you should have learned...

• ... dealing with bytecodes directly is possible, but very boring.
• We want reusable abstractions that hide the details (e.g. the different send bytecodes)

• We would like to have frameworks for
  – Generating bytecode easily
  – Transforming bytecode
Generating Bytecodes

- IRBuilder: A tool for generating bytecode
- Part of the new compiler for Squeak 3.9
- Idea: a symbolic Assembler for Squeak
IRBuilder: Simple Example

• Number>>asInteger

iRMethod := IRBuilder new
    numRargs: 1;
    addTemps: #(self); "receiver"
    pushTemp: #self;
    send: #truncated;
    returnTop;
    ir.

aCompiledMethod := iRMethod compiledMethod.

aCompiledMethod valueWithReceiver: 3.5
    arguments: #()
IRBuilder: Step by Step

- Number>>asInteger

  iRMethod := IRBuilder new

  - Make a instance of IRBuilder
IRBuilder: Step by Step

- **Number>>asInteger**

  iRMethod := IRBuilder new
  numRargs: 1;

- Define arguments. Note: “self” is default argument
IRBuilder: Step by Step

- Number>>asInteger

    iRMethod := IRBuilder new
    numRargs: 1;
    addTemps: #(self); "receiver"

    - define temporary variables. Note: arguments are temps
IRBuilder: Step by Step

- Number>>asInteger

  iRMethod := IRBuilder new
  numRargs: 1;
  addTemps: #(self); "receiver"
  pushTemp: #self

- push "self" on the stack
IRBuilder: Step by Step

• Number>>asInteger

iRMethod := IRBuilder new
    numRargs: 1;
    addTemps: #(self); "receiver"
    pushTemp: #self
    send: #truncated;
- call method truncated on “self”
**IRBuilder: Step by Step**

- **Number>>asInteger**

  ```smalltalk
  iRMethod := IRBuilder new
  numRargs: 1;
  addTemps: #(self); "receiver"
  pushTemp: #self
  send: #truncated;
  returnTop;
  ```

  - return Top of Stack
IRBuilder: Step by Step

• Number>>asInteger

iRMethod := IRBuilder new
numRargs: 1;
addTemps: #(self); "receiver"
pushTemp: #self
send: #truncated;
returnTop;
ir.

- tell IRBuilder to generate Intermediate Representation (IR)
IRBuilder: Step by Step

- Number>>asInteger

```scheme
iRMethod := IRBuilder new
    numRargs: 1;
    addTemps: #(self); "receiver"
    pushTemp: #self
    send: #truncated;
    returnTop;
    ir.

    aCompiledMethod := iRMethod compiledMethod.
```

- Generate method from IR
IRBuilder: Step by Step

- Number>>asInteger

iRMethod := IRBuilder new
  numRargs: 1;
  addTemps: #(self); "receiver"
  pushTemp: #self
  send: #truncated;
  returnTop;
  ir.

aCompiledMethod := iRMethod compiledMethod.

aCompiledMethod valueWithReceiver:3.5
  arguments: #()
- Execute the method with receiver 3.5 and no arguments.
- “3.5 truncated”

Marcus Denker
IRBuilder: Stack Manipulation

- popTop - remove the top of stack
- pushDup - push top of stack on the stack
- pushLiteral:
- pushReceiver - push self
- pushThisContext
IRBuilder: Symbolic Jumps

- Jump targets are resolved:
- Example: false ifTrue: ['true'] ifFalse: ['false']

iRMethod := IRBuilder new
  numRargs: 1;
  addTemps: #(self); "receiver"
  pushLiteral: false;
  jumpAheadTo: #false if: false;
  pushLiteral: 'true'; "ifTrue: ['true']"
  jumpAheadTo: #end;
  jumpAheadTarget: #false;
  pushLiteral: 'false'; "ifFalse: ['false']"
  jumpAheadTarget: #end;
  returnTop;
  ir.
IRBuilder: Instance Variables

- Access by offset
- Read: getField:
  - receiver on top of stack
- Write: setField:
  - receiver and value on stack
- Example: set the first instance variable to 2

```
iRMethod := IRBuilder new
    numRargs: 1;
    addTemps: #(self); "receiver"
    pushLiteral: 2;
    pushTemp: #self;
    setField: 1;
    pushTemp: #self;
    returnTop;
    ir.
```

```
aCompiledMethod := iRMethod compiledMethod.
aCompiledMethod valueWithReceiver: 1@2 arguments: #()
```
IRBuilder: Temporary Variables

- Accessed by name
- Define with addTemp: / addTemps:
- Read with pushTemp:
- Write with storeTemp:
- Example: set variables a and b, return value of a

irMethod := IRBuilder new
numRargs: 1;
addTemps: #(self); "receiver"
addTemps: #(a b);
pushLiteral: 1;
storeTemp: #a;
pushLiteral: 2;
storeTemp: #b;
pushTemp: #a;
returnTop;
ir.
IRBuilder: Sends

• normal send

```ruby
builder pushLiteral: 'hello'
builder send: #size;
```

• super send

```ruby
....
builder send: #selector toSuperOf: aClass;
```

- The second parameter specifies the class were the lookup starts.
IRBuilder: Lessons learned

• IRBuilder: Easy bytecode generation
  – Jumps
  – Instance variable
  – Temporary variables
  – Sends

• Next: Manipulating bytecode
ByteSurgeon

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

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

- Goal: logging message send.
- First way: Just edit the text:

```ruby
example
  self test.
```

Transcript show: ‘sending #test’.
```ruby
example
  self test.
```
Logging with Bytesurgeon

- Goal: Change the method without changing program text
- Example:

```smalltalk
(Example>>#example)instrumentSend: [:send |
  send insertBefore:
    'Transcript show: ''sending #test'' '
]
```
Logging: Step by Step

```
(Example>>#example)instrumentSend: [:send |
  send insertBefore:
    'Transcript show: ''sending #test'' '.
]
```

Example >> #example

- takes a name of a method
- returns the CompiledMethod object

Class

Name of Method

>>: - takes a name of a method

Marcus Denker
Logging: Step by Step

(Example>>#example)instrumentSend: [:send |
  send insertBefore:
    Transcript show: ‘’sending #test’’’.
]

- instrumentSend:
  - takes a block as an argument
  - evaluates it for all send bytecodes
Logging: Step by Step

(Example>>#example)instrumentSend: [:send |
  send insertBefore:
    ‘Transcript show: ‘’sending #test’’ ‘.
]

- The block has one parameter: send
- It is executed for each send bytecode in the method
Loggingle: Step by Step

- Objects describing bytecode understand how to insert code
  - insertBefore
  - insertAfter
  - replace

```
(Example>>#example)instrumentSend: [:send |
  send insertBefore: 
  'Transcript show: ''sending #test'' '.
]
```
Logging: Step by Step

- The code to be inserted.
- Double quoting for string inside string
  - Transcript show: ‘sending #test’
Inside ByteSurgeon

- Uses IRBuilder internally

- Transformation (Code inlining) done on IR
ByteSurgeon Usage

• On Methods or Classes:

   MyClass instrument: [.... ].
   (MyClass>>#myMethod) instrument: [.... ].

• Different instrument methods:
  - instrument:
  - instrumentSend:
  - instrumentTempVarRead:
  - instrumentTempVarStore:
  - instrumentTempVarAccess:
  - same for InstVar
ByteSurgeon: Lessons learned

• ByteSurgeon: Tool for editing bytecode
  – Simple example
  – Based on IRBuilder

• Next: Advanced ByteSurgeon
**Advanced ByteSurgeon:**

- Goal: extend a send with after logging

```example
    self test.
```

```example
    self test.
    Logger logSendTo: self.
```
Advanced ByteSurgeon

- With ByteSurgeon, something like:

(Example>>#example)instrumentSend: [:send |
  send insertAfter:
    ‘Logger logSendTo: ?’.
]

- How can we access the receiver of the send?
- Solution: Metavariable
Advanced ByteSurgeon

- With ByteSurgeon, something like:

```smalltalk
(Example>>#example)instrumentSend: [:send | 
    send insertAfter:
    'Logger logSendTo: <meta: #receiver>' .
]
```

- How can we access the receiver of the send?
- Solution: Metavariable
Implementation Metavariabes

- Stack during send:

  - Problem I: After send, receiver is not available
  - Problem II: Before send, receiver is deep in the stack
Metavariables: Implementation

- Solution: ByteSurgeon generates preamble
  - Pop the arguments into temps
  - Pop the receiver into temps
  - Rebuild the stack
  - Do the send
  - Now we can access the receiver even after the send
Metavariables: Implementation

25 <70> self
26 <81 40> storeIntoTemp: 0
28 <D0> send: test
29 <41> pushLit: Transcript
30 <10> pushTemp: 0
31 <E2> send: show:
32 <87> pop
33 <87> pop
34 <78> returnSelf
End

- Short overview of Squeak bytecode
- Introduction to bytecode generation with IRBuilder
- Manipulating bytecode with ByteSurgeon

- Questions?