Abstract Interpretation for Dummies

Program Analysis without Tears

Andrew P. Black
OGI School of Science & Engineering at OHSU
Portland, Oregon, USA
What is Abstract Interpretation?

• An interpretation:
  ◦ a way of understanding something

• Abstract:
  ◦ omitting some of the details
An Example: Portland
An Example: Portland
An Example: Portland
An Example: Portland
An Example: Portland

- Each interpretations makes available different information
  - all have less information than the real city!
  - in general, information content is disjoint
  - some interpretations dominate others, but

- Different interpretations have different uses
  - the map is more useful for driving than the photograph
  - it would be more useful still if it showed one-way streets!
Examples from Computing

- “Concrete” artifact is a program with the “standard” semantics
  - e.g., + is an operation that performs arithmetic
- “Abstract” artifact is a program with a non-standard semantics
  - e.g., + is an operation that builds an expression tree, or computes the type of the answer
Fraction >> reduced

reduced

<table>
<thead>
<tr>
<th>gcd</th>
<th>numer</th>
<th>denom</th>
</tr>
</thead>
</table>

numerator = 0 ifTrue: [↑0].
gcd ← numerator

gcd: denominator.

numer ← numerator // gcd.
denom ← denominator // gcd.
denom = 1 ifTrue: [↑numer].

↑Fraction numerator: numer denominator: denom
Fraction $\rightarrow$ reduced

1. Interpret inst vars as their types

\[\text{reduced}\]

\[| \text{gcd numer denom } | \]
\[\text{numerator} = 0 \text{ ifTrue: } [\uparrow0]. \]
\[\text{gcd } \leftarrow \text{numerator} \quad \text{gcd: denominator.} \]
\[\text{numer } \leftarrow \text{numerator} \div \text{gcd.} \]
\[\text{denom } \leftarrow \text{denominator} \div \text{gcd.} \]
\[\text{denom} = 1 \text{ ifTrue: } [\uparrow\text{numer}]. \]
\[\uparrow\text{Fraction numerator: numer denominator: denom} \]

1. Interpret inst vars as their types
Fraction >> reduced

1. Interpret inst vars as their types

reduced

| gcd numer denom |

<integer> = 0 ifTrue: [↑0].

gcd ← <integer> gcd: <integer> .

numer ← <integer> // gcd.

denom ← <integer> // gcd.

denom = 1 ifTrue: [↑numer].

↑Fraction numerator: numer denominator: denom
Fraction >> reduced

reduced
| gcd numer denom |
<integer> = 0 ifTrue: [↑0].
gcd ← <integer> gcd: <integer> .
numer ← <integer> // gcd.
denom ← <integer> // gcd.
denom = 1 ifTrue: [↑numer].
↑Fraction numerator: numer denominator: denom
Fraction >> reduced

2. Interpret constants as their types

```plaintext
reduced
| gcd numer denom |
<integer> = 0 ifTrue: [↑0].
gcd ← <integer> gcd: <integer> .
numer ← <integer> // gcd.
denom ← <integer> // gcd.
denom = 1 ifTrue: [↑numer].
↑Fraction numerator: numer   denominator: denom
```
Fraction >> reduced

2. Interpret constants as their types

reduced
| gcd numer denom |
\<integer\> = \<integer\> ifTrue: [\↑\<integer\>]
gcd ← \<integer\> gcd: \<integer\> .
umer ← \<integer\> // gcd.
denom ← \<integer\> // gcd.
denom = \<integer\> ifTrue: [\↑\<integer\>]
\↑Fraction numerator: numer   denominator: denom
Fraction >> reduced

reduced
| gcd numer denom |

<integer> = <integer> ifTrue: [↑<integer>]
gcd ← <integer> gcd: <integer> .
numer ← <integer> // gcd.
denom ← <integer> // gcd.
denom = <integer> ifTrue: [↑numer]
↑Fraction numerator: numer   denominator: denom
Fraction >> reduced

3. Interpret operations ...

reduced
| gcd numer denom |

\(<\text{integer}> = <\text{integer}> \text{ ifTrue: } [↑<\text{integer}>]\)

gcd \leftarrow <\text{integer}> \text{ gcd: } <\text{integer}> .

numer \leftarrow <\text{integer}> // gcd.

denom \leftarrow <\text{integer}> // gcd.

denom = <\text{integer}> \text{ ifTrue: } [↑\text{numer}]

↑Fraction numerator: numer    denominator: denom
Fraction >> reduced

3. Interpret operations ...

reduced

| gcd numer denom |

<integer> = <integer> ifTrue: [↑<integer>]
<integer> ← <integer> gcd: <integer> .
numer ← <integer> // <integer>
denom ← <integer> // <integer>
denom = <integer> ifTrue: [↑numer]
↑Fraction numerator: numer denominator: denom
Fraction $\gg$ reduced

3. Interpret operations ...

reduced

| gcd numer denom |
<integer> = <integer> ifTrue: [↑<integer>]
<integer> ← <integer> gcd: <integer>.
<integer> ← <integer> // <integer>

denom ← <integer> // <integer>
denom = <integer> ifTrue: [↑<integer>]
↑Fraction numerator: <integer> denominator: denom
Fraction >> reduced

3. Interpret operations ...

reduced

| gcd | numer | denom |

<integer> = <integer> ifTrue: [↑<integer>]
<integer> ← <integer> gcd: <integer> .
<integer> ← <integer> // <integer>
<integer> ← <integer> // <integer>
<integer> = <integer> ifTrue: [↑<integer>]

↑Fraction numerator: <integer> denominator: <integer>
Fraction >> reduced

3. Interpret operations ...

reduced
| gcd numer denom |
\[
\begin{align*}
\text{gcd: } & <\text{integer}> \\
\text{numerator: } & <\text{integer}> \\
\text{denominator: } & <\text{integer}> \\
\text{ifTrue: } & [↑<\text{integer}>] \\
\text{ifTrue: } & [↑<\text{integer}>] \\
\end{align*}
\]
Fraction >> reduced

reduced

| gcd numer denom |

<integer> = <integer> ifTrue: [↑<integer>]
<integer> ← <integer> gcd: <integer>
<integer> ← <integer> // <integer>
<integer> ← <integer> // <integer>
<integer> = <integer> ifTrue: [↑<integer>]
↑<fraction>
Fraction >> reduced

4. Collect answers

\[
\begin{array}{c|ccc}
\text{gcd} & \text{numer} & \text{denom} \\
\hline
<\text{integer}> & = & <\text{integer}> & \text{ifTrue: } [↑<\text{integer}>] \\
<\text{integer}> & ← & <\text{integer}> & \text{gcd: } <\text{integer}> \\
<\text{integer}> & ← & <\text{integer}> & // <\text{integer}> \\
<\text{integer}> & ← & <\text{integer}> & // <\text{integer}> \\
<\text{integer}> & = & <\text{integer}> & \text{ifTrue: } [↑<\text{integer}>] \\
↑<\text{fraction}> & & & \\
\end{array}
\]
Fraction >> reduced

reduced
| gcd numer denom |

↑<integer>

↑<integer>

↑<fraction>

4. Collect answers
My application: Inferring method requirements

- When a method sends a message $m$ to `self`, infer that its class should have a method on $m$
- When a method sends a message $n$ to `super`, infer that its superclass should have a method on $n$
- When a method sends a message $c$ to `self class`, infer that its metaclass should have a method on $c$
Back to ESUG 2003 ...
Back to ESUG 2003 ...

- Virtual Categories
Back to ESUG 2003 ...

- Virtual Categories

```
ifTrue: alternativeBlock

"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error notification if the receiver is non-Boolean. Execution does not actually reach here because the expression is compiled in-line."

self subclassResponsibility
```
Back to ESUG 2003...

- Virtual Categories

- categorization of methods by the browser, based on their characteristics; always up-to-date

```plaintext
ifTrue: alternativeBlock

"If the receiver is false (i.e., the condition is false), then the value is the false alternative, which is nil. Otherwise answer the result of evaluating the argument, alternativeBlock. Create an error notification if the receiver is nonBoolean. Execution does not actually reach here because the expression is compiled in-line."

self subclassResponsibility
```
Collecting information about self-sends requires that we parse the source code of all the methods ...
Collecting information about self-sends requires that we parse the source code of all of the methods ...
John Brant

- I think you can do this by looking at the byte code
- We did something similar once in the refactoring browser
Squeak Byte Code

• instructions for a simple stack-based VM

17 <00> pushRcvr: 0
18 <75> pushConstant: 0
19 <B6> send: =
20 <99> jumpFalse: 23
21 <75> pushConstant: 0
22 <7C> returnTop
23 <00> pushRcvr: 0
24 <01> pushRcvr: 1
25 <E0> send: gcd:
26 <68> popIntoTemp: 0

reduced

| gcd numer denom |
numerator = 0 ifTrue: [↑0].
gcd ← numerator gcd:
denominator.

...
CompiledMethod

• In Squeak, the class CompiledMethod is a subclass of ByteArray.
  ◦ It contains both the instructions (bytes) and the literal table (objects)
  ◦ While the on-disk format is standardized, the in-memory format of the literals depends on the byte order of the processor

• Use CompiledMethod >> literalAt: and InstructionStream to look at CompiledMethods
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  <Your interpreter>
InstructionStream Hierarchy

InstructionStream
  ContextPart
  BlockContext
  MethodContext
  Decompiler
  InstructionPrinter
  <Your interpreter>

interpret the byte-encoded Smalltalk instruction set. maintain a program counter (pc) for streaming through CompiledMethods.
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  <Your interpreter>
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  <Your interpreter>

adds the semantics for execution, and keeps track of sending context
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  <Your interpreter>
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
Decompiler
InstructionPrinter
<Your interpreter>

decompile a method into Smalltalk text by a three-phase process: postfix byte code → prefix symbolic code → node tree → text
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  <Your interpreter>
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
Decompiler
InstructionPrinter
  <Your interpreter>

Print-out a symbolic representation of the byte-codes
InstructionStream Hierarchy

InstructionStream
    ContextPart
        BlockContext
        MethodContext
    Decompiler
    InstructionPrinter
    <Your interpreter>
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  SendsInfo
InstructionStream Hierarchy

InstructionStream
  ContextPart
    BlockContext
    MethodContext
  Decompiler
  InstructionPrinter
  SendsInfo

My concrete subclass of InstructionStream which collects information about self-, super- and class-sends
InstructionStream subclass: #SendsInfo

• The core class of my abstract interpreter
  ◦ or for any similar single method interpreter

• Example usage:

  si ← SendsInfo on: ArrayCollection>>#writeOn: .
  si collectSends
InstructionStream subclass: #SendsInfo

• The core class of my abstract interpreter
  
  ○ or for any similar single method interpreter

• Example usage:

  si ← SendsInfo on: ArrayedCollection>>#writeOn: .
  si collectSends
      answers a SendsInfo that prints as
      [#[#basicSize]
        #( #=>writeOn:
        #( #=>isWords #isPointers)]
SendsInfo >> #collectSends

• “main loop” of the interpreter

    collectSends
        | end |
        end ← self method endPC.
        [pc <= end]
        whileTrue:
            [self interpretNextInstructionFor: self]
SendsInfo >> #collectSends

• “main loop”

collectSends
    | end |
    end ← self method endPC.
    [pc <= end]
    whileTrue:
        [self interpretNextInstructionFor: self]
SendsInfo >> #collectSends

- "main loop" of the interpreter

```plaintext
collectSends
    | end |
end ← self method endPC.
[pc <= end]
   whileTrue:
       [self interpretNextInstructionFor: self]
```
SendsInfo >> #collectSends

- “main loop” of the interpreter

```ruby
collectSends
  | end |
end ← self method endPC.
[pc <= end]
  whileTrue:
    [self interpretNextInstructionFor: self]
```

this SendsInfo is the argument as well as the target
SendsInfo >> #collectSends

- “main loop” of the interpreter

```plaintext
collectSends
    | end |
    end ← self method endPC.
    [pc <= end]
    whileTrue:
        [self interpretNextInstructionFor: self]
```
InstructionStream >> interpretNextInstructionFor:

interpretNextInstructionFor: client
"Send to client, a message that specifies the type of the next instruction."

| byte type offset method |
method ← self method.
byte ← method at: pc.
type ← byte // 16.
offset ← byte \ 16.
pc ← pc + 1.
type=0 ifTrue: [↑client pushReceiverVariable: offset].
type=1 ifTrue: [↑client pushTemporaryVariable: offset].
type=2 ifTrue: [↑client pushConstant: (method literalAt: offset+1)].
type=3 ifTrue: [↑client pushConstant: (method literalAt: offset+17)].
type=4 ifTrue: [↑client pushLiteralVariable: (method literalAt: offset+1)].
type=5 ifTrue: [↑client pushLiteralVariable: (method literalAt: offset+17)].
...

Instruction set Methods

- To detect self- and class-sends, we have to simulate the stack
  - But we can *abstract* over the set of values pushed on stack
  - We need distinguish only between `self`, `self class`, and other stuff
    - actually, there are also `blocks` and `small integers` representing the number of arguments to the block
Sample Instruction Set Methods

`pushTemporaryVariable`: offset
  `self push: #stuff`

`pushConstant`: value
  `self push: value`

`pushReceiver`
  `self push: #self`
The critical instruction set method

data: selector super: superFlag numArgs: numArgs

"Simulate the action of bytecodes that send a message with selector...
The arguments of the message are found in the top numArgs locations on the stack and the receiver just below them."

| stackTop |

...

self pop: numArgs.
stackTop ← self pop.
superFlag

ifTrue: [superSentSelectors add: selector]
ifFalse: [stackTop == #self
  ifTrue: [self tallySelfSendsFor: selector].

stackTop == #class
  ifTrue: [classSentSelectors add: selector]].
The critical instruction set method

send: selector super: superFlag numArgs: numArgs
"Simulate the action of bytecodes that send a message with selector...
The arguments of the message are found in the top numArgs locations on the stack and the receiver just below them."

| stackTop |

...

self pop: numArgs.
stackTop ← self pop.
superFlag

ifTrue: [superSentSelectors add: selector]
ifFalse: [stackTop == #self
  ifTrue: [self tallySelfSendsFor: selector].
  stackTop == #class
  ifTrue: [classSentSelectors add: selector]].
The critical instruction set method

**send**: selector **super**: superFlag **numArgs**: numArgs

"Simulate the action of bytecodes that send a message with selector...
The arguments of the message are found in the top numArgs locations on the stack and the receiver just below them."

```
| stackTop |
```

```
... 
self pop: numArgs.
stackTop ← self pop.
superFlag

  ifTrue: [superSentSelectors add: selector]
  ifFalse: [stackTop == #self
      ifTrue: [self tallySelfSendsFor: selector].
    stackTop == #class
      ifTrue: [classSentSelectors add: selector]].
```
The critical instruction set method

**send**: selector **super**: superFlag **numArgs**: numArgs

"Simulate the action of bytecodes that send a message with selector...
The arguments of the message are found in the top numArgs locations on the stack and the receiver just below them."

| stackTop |

... self **pop**: numArgs.
stackTop ← self pop.
superFlag

**ifTrue**: [superSentSelectors **add**: selector]
**ifFalse**: [stackTop == #self
  **ifTrue**: [self tallySelfSendsFor: selector]
  stackTop == #class
  **ifTrue**: [classSentSelectors **add**: selector]].
What about loops?

• Simplifying assumption:
  
  ◦ any send in a loop does not change from a self-send to a object-send
    
      • this is true because we consider \texttt{temp message} to always be an object-send, even if \texttt{temp == self}.

  ◦ so we never need to go around a loop more than once
    
      • we can just ignore backward jumps
SendsInfo >>jump:

jump: distance
"Simulate the action of a 'unconditional jump' bytecode whose offset is distance."
distance < 0
  ifTrue: [↑ self].
distance = 0
  ifTrue: [self error: 'bad compiler!'].
...

What about forward jumps?

```ruby
classPool
  "Answer the dictionary of class variables."
classPool == nil
  ifTrue: [↑Dictionary new]
  ifFalse: [↑classPool]
```

- instructions like 16 are “merge points” in the instruction sequence:
  - execution can reach a merge point in two ways.
  - which stack should we use?
What about forward jumps?

classPool

"Answer the dictionary of class variables."
classPool == nil
    ifTrue: [^Dictionary new]
    ifFalse: [^classPool]

• instructions like 16 are “merge points” in the instruction sequence:
  - execution can reach a merge point in two ways.
  - which stack should we use?

9 <0D> pushRcvr: 13
10 <73> pushConstant: nil
11 <C6> send: ==
12 <9A> jumpFalse: 16
13 <40> pushLit: Dictionary
14 <CC> send: new
15 <7C> returnTop
16 <0D> pushRcvr: 13
17 <7C> returnTop
What about forward jumps?

```smalltalk
classPool
    "Answer the dictionary of class variables."
classPool == nil
    ifTrue: [Dictionary new]
    ifFalse: [classPool]
```

- instructions like 16 are “merge points” in the instruction sequence:
  - execution can reach a merge point in two ways.
  - which stack should we use?

9 <0D> pushRcvr: 13
10 <73> pushConstant: nil
11 <C6> send: ==
12 <9A> jumpFalse: 16
13 <40> pushLit: Dictionary
14 <CC> send: new
15 <7C> returnTop
16 <0D> pushRcvr: 13
17 <7C> returnTop
What about forward jumps?

classPool
    "Answer the dictionary of class variables."
classPool == nil
    ifTrue: [↑Dictionary new]
    ifFalse: [↑classPool]

- instructions like 16 are “merge points” in the instruction sequence:
  - execution can reach a merge point in two ways.
  - which stack should we use?

9 <0D> pushRcvr: 13
10 <73> pushConstant: nil
11 <C6> send: ==
12 <9A> jumpFalse: 16
13 <40> pushLit: Dictionary
14 <CC> send: new
15 <7C> returnTop
16 <0D> pushRcvr: 13
17 <7C> returnTop
Merging stacks

- Every conditional forward jump marks its destination address as a merge point
  - we save the state of the stack at the jump point in a dictionary keyed by the merge point

```plaintext
jump: distance if: aBooleanConstant
   "Simulate the action of a 'conditional jump' bytecode ..."
   | destination |
   distance < 0 ifTrue:[↑ self].
   distance = 0 ifTrue:[self error: 'bad compiler!'].
   destination ← self pc + distance.
   self pop. "remove the condition from the stack."
   savedStacks at: destination put: stack copy.
```
Merging stack (2)

- We have to check for the possibility of a merge point at every instruction!
  - We do this by overriding InstructionStream >> interpretNextInstructionFor:
Merging stack (2)

- We have to check for the possibility of a merge point at every instruction!
  - We do this by overriding InstructionStream >> interpretNextInstructionFor:

``` Smalltalk
SendsInfo>>interpretNextInstructionFor: client
  self atMergePoint
  ifTrue: [self mergeStacks].
  super interpretNextInstructionFor: client
```
Merging stack (2)

- We have to check for the possibility of a merge point at every instruction!
  - We do this by overriding `InstructionStream >> interpretNextInstructionFor:`
Merging stack (2)

• We have to check for the possibility of a merge point at every instruction!
  ◦ We do this by overriding InstructionStream >> interpretNextInstructionFor:

• Merging stacks is conservative:
  ◦ If element $i$ in either stack is #self, then element $i$ in the merged stack is also #self.
Dealing with Blocks

• The code for blocks is compiled in-line, and copied onto the stack at execution time

```Smalltalk
removeAll: aCollection
daCollection do: [:each | self remove: each].
↑ aCollection
```
Dealing with Blocks

- The code for blocks is compiled in-line, and copied onto the stack at execution time

```smalltalk
9 10 pushTemp: 0
10 89 pushThisContext:
11 76 pushConstant: 1
12 C8 send: blockCopy:
13 A4 05 jumpTo: 20
15 69 popIntoTemp: 1
16 70 self
17 11 pushTemp: 1
18 E0 send: remove:
19 7D blockReturn
20 CB send: do:
21 87 pop
22 10 pushTemp: 0
23 7C returnTop
```

`removeAll: aCollection`

```smalltalk
aCollection do: [:each | self remove: each].
```

Number of arguments of the block
Dealing with Blocks

- The code for blocks is compiled in-line, and copied onto the stack at execution time

```plaintext
removeAll: aCollection

aCollection do: [:each | self remove: each].
↑ aCollection
```
Dealing with Blocks

- The code for blocks is compiled in-line, and copied onto the stack at execution time

9 <10> pushTemp: 0
10 <89> pushThisContext:
11 <76> pushConstant: 1
12 <C8> send: blockCopy:
13 <A4 05> jumpTo: 20
15 <69> popIntoTemp: 1
16 <70> self
17 <11> pushTemp: 1
18 <E0> send: remove:
19 <7D> blockReturn
20 <CB> send: do:
21 <87> pop
22 <10> pushTemp: 0
23 <7C> returnTop
Dealing with Blocks

- The code for blocks is compiled in-line, and copied onto the stack at execution time

1. Sending a block copy remembers the number of arguments
2. When we drop into the block, empty stack and push some dummy arguments
3. No merge of stacks is needed at 20 (or after any unconditional jump
Performance

• Initially: 27 seconds to analyze every method (~45,000) in the image (600 µs per method).

• Where did the time go?
  ◦ Dictionaries: SendsInfo >> #atMergePoint is usually false
  ◦ OrderedCollection: stack manipulations are slow
  ◦ InstructionStream>>interpretNextInstructionFor: uses linear dispatch

• Fix these things ... 260 µs per method
Conclusions

• AI was easy to implement
  ◦ spare time at last two days of ESUG 2003 and flight home
  ◦ Thanks to John Brant and Vassili Bykov

• AI implementation was more accurate than implementation based on parsing source
  ◦ Morph>>layoutInBounds: contains the statement:
    (owner ifNil:[self]) cellPositioning
    which AI detects as self-send

• AI is fast
  ◦ Bytecodes are designed for fast interpretation