First Class Variables as AST Annotations

Marcus Denker

Inria RMoD
Part I: The AST

- AST = Abstract Syntax Tree
- Tree Representation of the Method
- Based on the RB AST
- Used by all tools (refactoring, syntax-highlighting, ...)

Smalltalk compiler parse: 'test ^(1+2)'
• RBMethodNode                    Root
• RBVariableNode                    Variable (read and write)
• RBAssignmentNode              Assignment
• RBMessageNode                  A Message (most of them)
• RBReturnNode                     Return
Inspect a simple AST

• A very simple Example

Smalltalk compiler parse: 'test ^(1+2)'
User: Tools

- Refactoring
- Breakpoints / Watchers
- Syntax Highlight / Code Completion
- AST based Menu in the Code Browser
User: The Compiler

Source → AST → Annotated AST
- RBParser
- OCSemanticAnalyzer

Annotated AST → IR → Bytecode
- OCASTTranslator/IRBuilder
- IRBytecodeGenerator
Variables in the AST

- Example: (Point>>#x)
Problem: Kind of Variable?

- Example: SHRBTextStyler

- Syntax highlighting needs to know which kind
Variables in the AST

- Every definition, read and write gets one new instance of RBVariableNode (as we have to encode the parent for each differently)
  - We just know the name
  - SYNTAX, but no SEMANTICs
    - Kind? (temp or ivar)
    - Variables with same name can be different variables
To the Rescue: Name Analysis

- We have to annotate the AST with information about Variables
- Block/Method: defined Variables are put in a Scope
  - Scopes know the parent Scope
- When we see a use, we loop up the variable in the Scope
Semantic Variables

• Every RBVariableNode gets a semantic variable annotation
  • Both the definition and all uses

• There is one instance for each variable that models
  • name
  • scope it was defined
Variables in the AST

• Example Again: (Point>>#x)
Variables and Compilation

- Compiler just delegates to the Variable, e.g. for instance Variables:

  ```plaintext
  emitStore: methodBuilder
  "generate store bytecode"
  methodBuilder storeInstVar: index
  ```

- `emitStore/emitValue`: defined for each kind of Variables (global/temp/ivar)
Repeat: The AST

- AST = Abstract Syntax Tree
- Tree Representation of the Method
- Produced by the Parser (part of the Compiler)
- Used by all tools and the Compiler
- We need to model Variables semantically to make it useful
Now Step Back
Forget Part I
(for now)
Look at it from Reflective Point of View
Part II
First Class Variables
First: Variables in ST80
Instance Variables

- Defined by the Class (list of variable names)
- Can be read via the object:
  - `instVarNamed:` (put:), `#instVarAt:` (put:)
- Instance Variables have an offset in the Object
- Defined by the order of the defined vars in the Hierarchy

```haskell
1@2 instVarNamed: 'x'
```
Temporary Variable

- Defined by a method or Block
  - Arguments are temps, too
- Can be read via the context
  - `#tempNamed:, tempNamed:put:`

\[
\text{\texttt{[\{temp \mid temp := 1. thisContext tempNamed: 'temp'} \texttt{\} value}}
\]

- With Closures this is more complex than you ever want to know!
Globals

- Entries in the “Smalltalk globals” Dictionary
- Contain the value
- Can be read via the global Dictionary
- Access via #value / value: on the Association
- Class Vars and Pool Vars are just Associations from other Dictionaries
“Everything is an Object”
For Variables... not really
Globals/Class Vars

- Here we have at least the Association (#binding):
  
  Object binding

- But there is no “GlobalVariable” class
  
  - No API other than #value:/#value
  
  - Classes define just names of variables
Instance Variables

• The class just knows the names

  Point allInstVarNames

• There is no Object representing instance variables

• Classes define just names of variables

• Bytecode accesses by offset
Temporary Variables

- The methods know nothing. Even to know the variable name we need the compiler (and the source)
- There is no object representing temp Variables
- Reflective read and write is *hard* -> compiler needs to create extensive meta-data
Why Not Do Better?

• Every defined Variable is described a meta object

• Class Hierarchy: Variable
The Hierarchy

- Variable
  - LiteralVariable
  - ClassVariable
  - GlobalVariable
  - UndeclaredVariable
  - WorkspaceVariable

- LocalVariable
  - ArgumentVariable
  - TemporaryVariable

- ReservedVariable
  - SelfVariable
  - SuperVariable
  - ThisContextVariable

- Slot
Example: vars of a class

- Get all Variables of a class
- Inspect it
- #usingMethods

Point instanceVariables
Instance Variable

- Read x in a Point
  
  (Point instanceVariables first) read: (5@4)

- Write
  
  point := 5@4.
  (Point instanceVariables first) write: 100 to: point.

- read/write without sending a message to the object!
Globals

- Object binding class
- Object binding read

- We keep the Association API so the Global Variables can play the role of associations in the global dictionary.

Object binding using Methods
Temporary Variables

- There are too many to allocate them all
- They are created on demand (with the AST)

```((LinkedList>>#do:) temporaryVariableNamed: 'aLink')```
#lookupVar:

- Every variable knows the scope it was defined in.
- Every scope knows the outer scope.
  
  (Point slotNamed: #x) scope outerScope
  
- #lookupVar: looks up names along the scope.
  
  [[ | temp | thisContext lookupVar: 'temp' ] value.
  
  [[ | temp | thisContext lookupVar: 'Object' ] value.
Debugger: Read Vars

- In the Debugger we to be able to read Variables from a Dolt.

- lookupVar, then readInContext works for all Variables!

```plaintext
[ | temp | temp :=1 . (thisContext lookupVar: 'temp')
     readInContext: thisContext] value
```

- If you know the context, you can read any variable

- DoItVariable: Nice names in Dolts (→ Show Us)
Part III: Putting it Together

- We have seen how Semantic Variables are needed to make the AST useful
- We have seen First Class Variables as part of the Reflective Model
- Do we really need the two?
Solution: Scope

- What is needed? Add the concept of Scope
  - Scope of a global is Smalltalk globals
  - Scope of an instance variable is the class
  - Scope of temp: method and block scope
Example: Point x

(Point slotNamed: #x) scope == Point

(Point lookupVar: #x) == (Point slotNamed: #x)

(Point>>#x) ast variableNodes first variable == (Point slotNamed: #x)
What do we get?

- Simplified Name Analysis in the Compiler
- Open Compiler: Define your own kinds of Variables
- While fully integrated in the Reflective Model
  - Reflective Reading/Writing
- All tools work for you own kinds of Variables
What we did not see...

- Define your own kinds of Variables (e.g. subclasses of Slot / ClassVariable)
- Fluid Class Definitions: How to create classes that use these variables
- How this enables Dolts with nice variable names
- Reflection: MetaLinks on Variables
Thanks...

- This is the work on *many* contributors from the Pharo Community

- Thanks for lots of interesting discussions, ideas, and code!
Questions?

• We have seen how the AST needs semantic variables to be useful

• We have seen First Class Variables as part of the Reflective model

• First Class Variables, with just adding the concept of a Scope, can serve as semantic annotations on the AST