Externalizing Business-Object Behavior: More on a Point-and Click Rule Editor

Paul Davidowitz

In the previous article (*Smalltalk Report*, September, p. 4), we introduced a point-and-click rule editor that manipulates the ProgramNode tree. We continue to investigate how it works.

TYPING

The tool's type system is based on the *types as sets of classes* approach used by Graver and Johnson.[1,2] A ProgramNodeType is defined as a set of zero or more ProgramNodeTypeOptions. A ProgramNodeTypeOption has a class name and a *taxonomy*. *Taxonomy* indicates whether an exact match with the class is required (isMember) or whether a subclass can also be used (isKind). For example, a type option for Boolean or Number has the isKind taxonomy because these are abstract classes. The ProgramNodeType is depicted with angle brackets, with a vertical bar between options, as in <ByteSymbol | (kindOf: Number)>; the vertical bar is read as or. If kindOf> is notated, the taxonomy is isKind: otherwise, it is isMember.

A type can be said to satisfy another type. For example, <kindOf: Number> is satisfied by <Integer>; and <ByteSymbol | ByteString> is satisfied by <ByteSymbol>. Two types can be merged by adding their unique type options.

There are two special types, <'Anything'> and <'Nothing'>. <'Anything'> is shorthand for <kindOf: Object>. <'Nothing'> has zero options and is used exclusively for either an out-of-scope condition, or for an unknown type inside a nonevaluated block. To illustrate:

```
[: aBoolean |   | t1 |
    t1 := 'abc'.
    t1 "<'Nothing'> out-of-scope" := aBoolean
        ifTrue: [^123]
        ifFalse: [^#xyz]].
[  | t1 |
    t1 := 'abc'.
    [t1 "<'Nothing'> non-evaluated-block"]]
```

Selector Information. One of the MorphConstructs a MessageNodeWrapper knows how to perform is *Change message selector* and *arguments only* (123 + 456 → 123 * nil). The user is usually presented with a set of this construct, with each element of the set having selector information that includes the selector, argument types (if any), and the return type. Where does this set of selector information come from?

For a given type option of the receiver, if the type-option's class is not a business object, the information comes from hardcoded information on the base classes; otherwise it is generated from the business-object's logical-schema attribute/type specification. The set of selector information presented to the user is the intersection of sets from all the type options; therefore, for distant options this may mean selector information from Object only.

Let's say the selection is a MessageNodeWrapper of receiver SmallInteger, for example, 123 isNil. The receiver knows its type (<SmallInteger>), so we ask the class, whose name we get from the single type option, for selector information. This information will come from SmallInteger class and its superclasses.

This is an example of selector information from the class side of Number:

```
^TypedSelector
    singleArgumentSelector: #<
    receiverRequiredType: ProgramNodeType number
    argumentType: ProgramNodeType number
    returnType: ProgramNodeType boolean
```

The selector, required type for receiver, required type for argument, and return type are all specified (via an instance of class TypedSelector). Selector information is inherited and can be overridden. It is specified for operators of those base classes which are used as business-object attribute types (e.g. Boolean, ByteString, Number).

OTHER TYPE OPTIONS

ProgramNodeTypeOption lives in a hierarchy:

```
AbstractProgramNodeTypeOption
    ReifiedBlockValueTypeOption (argumentIndex)
    ProgramNodeTypeOption (className, taxonomy)
```

```
        BlockClosureTypeOption (valueType,
    argumentTypes)
        HomogeneousCollectionTypeOption
    (elementType)
```

HomogeneousCollectionTypeOption describes a homogeneous collection, meaning all elements have the same elementType. *Homogeneous* can be very flexible, as the type can be <'Anything'>, for example. This type option takes selector information from the instance side, as well as the class side. An instance of the collection is created and given one element: a copy of elementType.

Here's an example of selector information from the *instance* side of Collection:

```
^TypedSelector
    singleArgumentSelector: #select:
    receiverRequiredType: (ProgramNodeType
        homogeneousCollectionOfName: self class name
        elementType: self first copy)
    argumentType: (ProgramNodeType
        blockClosureSingleArgument: self first copy
        valueType: ProgramNodeType boolean)
    isArgumentPrototyping: true
    blockArgumentEvaluator:
MessageWrapperBlockArgumentEvaluator nothingOrLoop
    returnType: (ProgramNodeType
        homogeneousCollectionOfName: self class name
        elementType: self first copy)
```

The argument of the #select: message is specified as a block whose single argument is a copy of elementType. The return type is specified as a collection of this type. Note that this selector information specifies using a prototype argument; that is, the argument instead of being nil will be a block with an argument of the correct type.

ReifiedBlockValueTypeOption is used to determine the return type of a message. Here's the selector information for #ifTrue: from the class side of Boolean:

```
^TypedSelector
    singleArgumentSelector: #ifTrue:
    receiverRequiredType: ProgramNodeType boolean
    argumentType: (ProgramNodeType
        blockClosureNoArgumentsAndValue:
            ProgramNodeType anything)
    isArgumentPrototyping: true
    blockArgumentEvaluator:
        MessageWrapperBlockArgumentEvaluator onOrOff
    returnType: (ProgramNodeType options:
        (OrderedCollection
        with: (ReifiedBlockValueTypeOption onArgumentIndex: 1)
        with: ProgramNodeTypeOption nil))
```

This states that the return type for #ifTrue: consists of type options for nil, and for the value of the block expected as the message's first argument. For example, the type of message aBoolean ifTrue: [123] is <UndefinedObject | SmallInteger>; for aBoolean ifTrue: ['abc'], it is <UndefinedObject | ByteString>.

TRAVERSAL OF THE WRAPPER TREE

Traversal of the wrapper tree yields valuable information such as node-wrapper type. Traversal is done in postorder fashion.

A wrapper knows the order in which to traverse its children. For example, A MessageNodeWithArgumentsWrapper has the child traversal order: {receiver, argument-collection}.

The Block Evaluator. If at least one of the arguments of a MessageNodeWithArgumentsWrapper is a block defined (via selector information) as potentially evaluating, we append pseudo-child MessageWrapperBlockArgument Evaluator to the child traversal order (*pseudo* in the sense that the MessageNode itself has no such child). The evaluator serves to simulate evaluation of a block by traversing it. Without the evaluator, the BlockNodeWrapper is treated as a leaf wrapper and is not traversed. The evaluator poses as having the block-argument grandchildren of its parent, as its own children.

The evaluator is defined with a collection of EvaluationMetaSpecs. An EvaluationMetaSpec is a description for one step through of the method specified in the message. This spec is used to produce a collection of child traversal orders. An EvaluationMetaSpec, in turn, is defined with a collection of ArgumentMetaSpecs. The ArgumentMetaSpec states whether the block argument is optional, and whether it is possibly a looping block; this spec is identified by the message-argument index.

For example, here is Boolean>>ifTrue: ifFalse: with its two EvaluationMetaSpecs:

- {required, noLoop, index 1}
- {required, noLoop, index 2}

The ifTrue: argument is the first argument of the message, and thus is designated by *index 1*; the ifFalse: being the second is designated by *index 2*. These EvaluationMetaSpecs produce the two traversal orders: {{1},{2}}; in other words, we must either evaluate the first message argument, the ifTrue: block; or else we must evaluate the second message-argument, the ifFalse> block.

Here is Collection>>detect:ifNone: with its two EvaluationMetaSpecs:

- {required, loop, index 1}
- {optional, loop, index 1}, {required, noLoop, index 2}

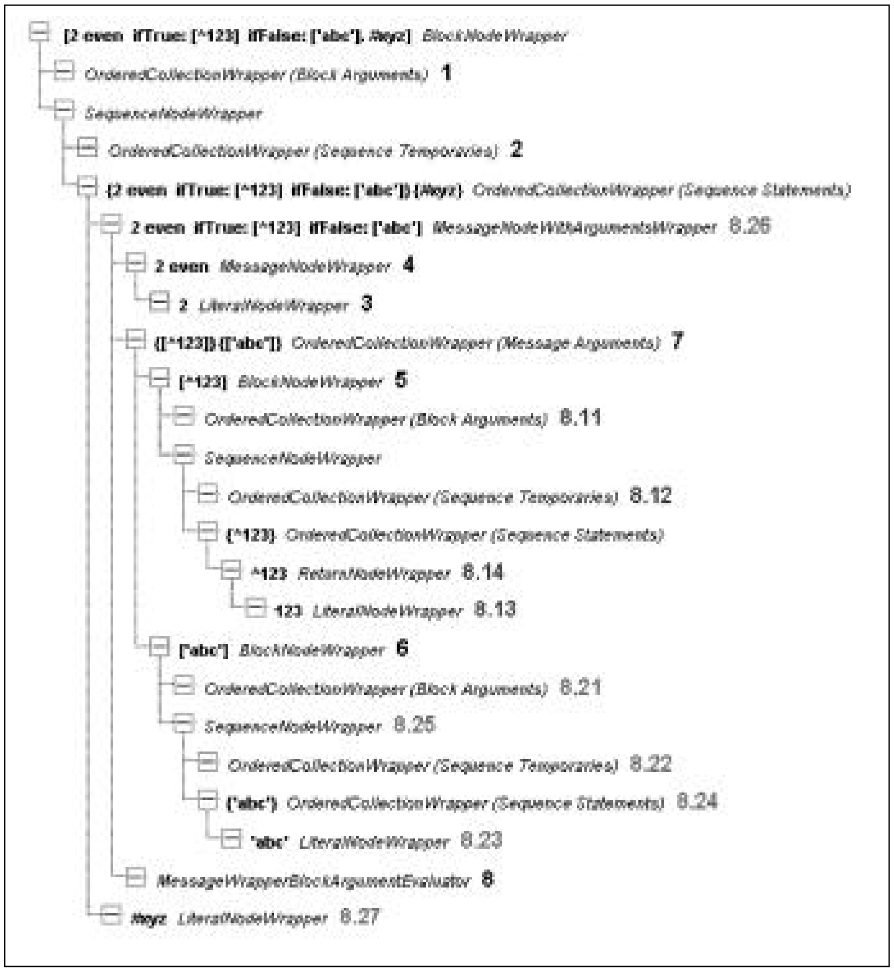
We either loop one or more times evaluating the detect: block or we possibly loop one or more times evaluating the detect: block, followed by definitely evaluating the ifNone: block once.

BRANCHING AT THE EVALUATOR

The evaluator always terminates the current traversal. The result for the evaluator is obtained by branching new traversals and combining the results; each child traversal order of the evaluator produces another branch.

Let's find a block's value type.

```
[2 even
```



```
[2 even ifTrue: [^123] ifFalse: ['abc']. #xyz] BlockNodeWrapper
  OrderedCollectionWrapper (Block Arguments) 1
  SequenceNodeWrapper
    OrderedCollectionWrapper (Sequence Temporaries) 2
    (2 even ifTrue: [^123] ifFalse: ['abc']) (#xyz) OrderedCollectionWrapper (Sequence Statements)
      2 even ifTrue: [^123] ifFalse: ['abc'] MessageNodeWithArgumentsWrapper 8.26
        2 even MessageNodeWrapper 4
          2 LiteralNodeWrapper 3
        ([^123]) (['abc']) OrderedCollectionWrapper (Message Arguments) 7
          [^123] BlockNodeWrapper 5
            OrderedCollectionWrapper (Block Arguments) 8.11
            SequenceNodeWrapper
              OrderedCollectionWrapper (Sequence Temporaries) 8.12
              (^123) OrderedCollectionWrapper (Sequence Statements)
                ^123 ReturnNodeWrapper 8.14
                  123 LiteralNodeWrapper 8.13
          ['abc'] BlockNodeWrapper 6
            OrderedCollectionWrapper (Block Arguments) 8.21
            SequenceNodeWrapper 8.25
              OrderedCollectionWrapper (Sequence Temporaries) 8.22
              ('abc') OrderedCollectionWrapper (Sequence Statements) 8.24
                'abc' LiteralNodeWrapper 8.23
        MessageWrapperBlockArgumentEvaluator 8
      #xyz LiteralNodeWrapper 8.27
```

Figure 9. Branching.

```
        ifTrue: [^123]
        ifFalse: ['abc'].
    #xyz] "what is value-type for the block?"
```

We start at the first terminal node wrapper of the block and perform a forward traversal on the lookout for either a ReturnNodeWrapper or the very last statement of the block. We eventually arrive at the evaluator and proceed to find its result, which in this case will be our answer. Because the evaluator has two child traversal orders, it branches two traversals. We take the first traversal and find a ReturnNodeWrapper of type <SmallInteger>. The second takes us past the confines of the ifFalse: block. We visit a MessageNodeWithArgumentsWrapper, the final outer block statement and stop with a type of <ByteSymbol>. We combine the results to get our answer of <SmallInteger | ByteSymbol>. This traversal is depicted in Figure 9. The first branch occurs in visitations 8.1x shown in dark grey; the second, in 8.2x shown in light grey. Note that although 2 even is always true, the tool is unaware of this.

Looping. Let's look at type inferencing for a temporary variable. Let's find the type of t2 in the final statement of:

```
  [: a1 |   | t1 t2 t3 |
  t1 := 123.
  t2 := 'abc'.

  a1 myCollection
      detect: [: a2 |
          t3 := t1.
          t1 := $a.
          t2 := t3]
      ifNone: [nil].
  t2 "what is my type?" ]
```

If we don't loop at all, t2 type = <ByteString>; if we make one pass, t2 type = <SmallInteger>; if we make two passes, t2 type = <Character>. Thus, results differ depending on the number of times the detect: block is traversed.

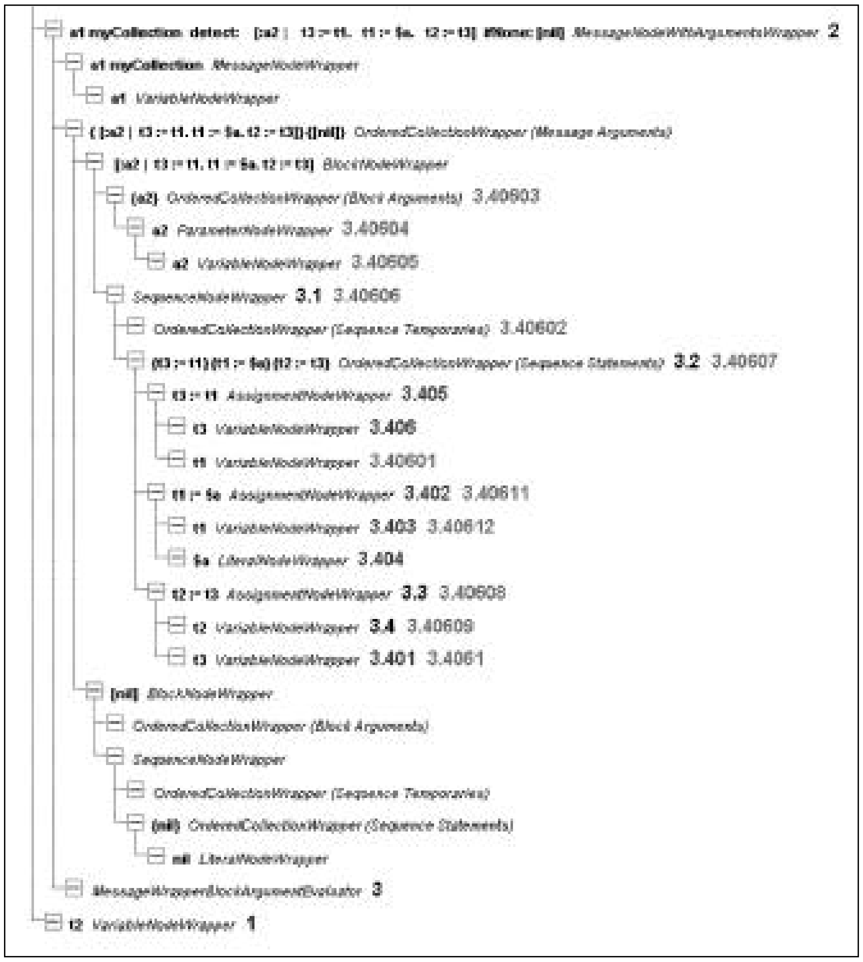
The EvaluationMetaSpecs generate the appropriate child traversal orders, based on the number of descendent assignment statements (excluding those found in child block descendents). Each assignment generates another child traversal order. In the example, we find three assignments in the detect: block. The first EvaluationMetaSpec for #detect:ifNone: therefore generates {{1}, {1,1}, {1,1,1}}, and the second generates {{2}, {1,2}, {1,1,2}, {1,1,1,2}}. (We play it safe to loop the maximum amount, even though in this case only two passes are needed.) If we had no assignments, the resulting child traversal orders would simply be {{1}} from the first spec, and {{2}, {1,2}} from the second.

This brute-force technique is of exponential order, but this is not a concern for minimal branching. The standard technique for type inferencing uses polynomial-order symbolic-execute via solution of equations.[3]

Type inferencing for a temporary variable involves traversing the wrapper tree backwards from the VariableNodeWrapper. Let's find the result for a double pass by walking the child traversal order {1,1} branch of the #detect:ifNone: evaluator. As shown in Figure 10, we proceed backwards on the lookout for AssignmentNodeWrappers with VariableNodeWrapper t2, and eventually reach the detect: block's third statement and stop at the AssignmentNodeWrapper. The result of the AssignmentNodeWrapper is the type of its righthand side, VariableNodeWrapper on t3. So we trigger a new traversal for inferencing the type of t3 (shown in dark grey), but the technique is to continue using the current child traversal order of the evaluator. This new traversal in turn triggers another for inferencing the type of t1 (shown in light grey), again keeping the child traversal order. We know to traverse the detect: block once more, and find type t1 to be <Character>.

TREATMENT OF BLOCKS

The contents of a block are treated as live, whether the block is evaluated or not. This is not an oversight, but a



```
a1 myCollection detect: [:a2 | t3 := t1. t1 := $a. t2 := t3] ifNone: [nil] MessageNodeWithArgumentsWrapper 2
  a1 myCollection MessageNodeWrapper
    a1 VariableNodeWrapper
  ( [:a2 | t3 := t1. t1 := $a. t2 := t3]) ([nil]) OrderedCollectionWrapper (Message Arguments)
    [:a2 | t3 := t1. t1 := $a. t2 := t3] BlockNodeWrapper
      (a2) OrderedCollectionWrapper (Block Arguments) 3.40603
        a2 ParameterNodeWrapper 3.40604
          a2 VariableNodeWrapper 3.40605
      SequenceNodeWrapper 3.1 3.40606
        OrderedCollectionWrapper (Sequence Temporaries) 3.40602
        (t3 := t1) (t1 := $a) (t2 := t3) OrderedCollectionWrapper (Sequence Statements) 3.2 3.40607
          t3 := t1 AssignmentNodeWrapper 3.405
            t3 VariableNodeWrapper 3.406
            t1 VariableNodeWrapper 3.40601
          t1 := $a AssignmentNodeWrapper 3.402 3.40611
            t1 VariableNodeWrapper 3.403 3.40612
            $a LiteralNodeWrapper 3.404
          t2 := t3 AssignmentNodeWrapper 3.3 3.40608
            t2 VariableNodeWrapper 3.4 3.40609
            t3 VariableNodeWrapper 3.401 3.4061
    [nil] BlockNodeWrapper
      OrderedCollectionWrapper (Block Arguments)
      SequenceNodeWrapper
        OrderedCollectionWrapper (Sequence Temporaries)
        (nil) OrderedCollectionWrapper (Sequence Statements)
          nil LiteralNodeWrapper
  MessageWrapperBlockArgumentEvaluator 3
t2 VariableNodeWrapper 1
```

Figure 10. Looping.

necessity. Remember that the rule block itself is dead if it is not being evaluated; surely we want this type checked, as this is the whole point. Likewise for any descendent block.

Also, remember that due to the morphing process, what is currently not a block can perhaps be morphed to a block's first statement, and vice versa (construct *Enclose statement in block* (123→[123]) and construct *Return block's first and only statement* ([123]→123)).

Traversal beyond the confines of a block depends on the situation. If the block is a statement, then there is no traversal beyond it. If the block is a message argument, then it depends on the block evaluator EvaluationMetaSpec acknowledging the block. Even if the block evaluator knows that a given block is never evaluated, the contents of the block are treated as fully viable (e.g., if the user selects inside the block); if it is not evaluated we simply don't traverse past the confines of the block. (An unknown type in this situation is defined as <'Nothing'> as discussed). Currently, all block-evaluator EvaluationMetaSpecs have live blocks.

CREATION OF THE WRAPPER TREE

Creation of the wrapper tree occurs on string input to the tool, as well as acceptance of text from the *FreeStyle* text view. The creation is achieved in iterative fashion, using forward traversal, during which we force ourselves into blocks in order to traverse them.

The MessageNodeWrapper does the main job. It finds the appropriate selector information based on the type of its receiver. From this information, the required types of its arguments are instated, as well as the type of the message itself.

Validation is performed, ensuring that a wrapper conforms to syntax and the limitations of the tool. For example, a ReturnNodeWrapper ensures that it is last in a sequence of statements, and an AssignmentNodeWrapper ensures that its value child is not a BlockNodeWrapper. A faulty wrapper is highlighted in the *FreeStyle* text view along with the appropriate error.

CONCLUSION

Eagle needed a rule language, and Smalltalk itself was chosen. A point-and-click rule editor was developed that constrains the user to produce valid syntax with valid message selectors, via selection of valid ProgramNode tree manipulations. The user is responsible for valid message arguments by selecting from manipulations that satisfy type requirements. State and behavior wrapped around the ProgramNode tree make it type aware and traversable.

Can such a tool be generalized for editing Smalltalk in general? Probably not. Several issues would quickly get out of hand, such as specifying and maintaining selector information for all selectors. But for a rule language subset, the tool has a niche. S

References

1. Graver, J. O. and Johnson, R. E., "A Type System for Smalltalk," In Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 136-150, Jan. 1990.
2. Graver, J. O. Type-Checking and Type-Inference for Object-Oriented Programming Languages, PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Aug. 1989. UIUCD-R-89-1539.
3. Palsberg, J. and Schwartzbach, M. I., Object-Oriented Type Systems, John Wiley and Sons, New York, 1994.

Figures 9 and 10 inadvertently ran as Figures 6 and 7 in part 1 of this article in September. We apologize for any confusion.

Paul Davidowitz is a senior developer at Andersen Consulting. He can be reached at paul.davidowitz@ac.com.