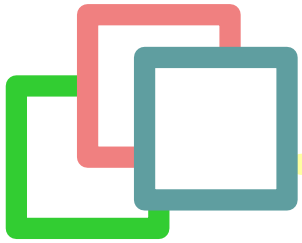

Inside AOStA

an adaptively optimizing Smalltalk architecture

Paolo Bonzini—ESUG 2003, Bled

AOStA is an experimental project hoping to implement Self's optimization techniques in a Smalltalk virtual machine.

There are some similarities and some differences:

- √ AOStA does use profiled execution to identify optimization techniques in a Smalltalk virtual machine.
- √ AOStA will use dynamic deoptimization to allow source-level debugging of optimized code.
- x AOStA's optimizer is written in Smalltalk.
- x AOStA compiles to optimized bytecode instead of native code.

An AOStA enabled VM has two areas for JIT-compiled methods.

- √ The optimized area works as usual
- √ In the unoptimized area methods have a countdown for each send and backward branch
  - ▷ When the counter trips, the optimizer is called back by the VM!

An AOStA enabled VM has Polymorphic Inline Caches and allows Smalltalk code to read them.

- √ PICs are a natural way to find information about the types of the receivers.
- √ Many virtual machines already implement them!

The optimizer can:

- √ Will do some type inferencing based on the known types of the primitives' answers
- √ Inline methods whose receiver class is constant, or replace them with constant-class sends.
- √ Duplicate different parts of the methods to specialize for the receiver classes contained in the PIC.
- √ Use additional bytecodes to perform common primitive operations directly from the translated methods
  - ▷ SmallInteger operations
  - ▷ instantiation
  - ▷ accessing indexed instance variables, either with or without bounds-checking

The optimizer reads and produces bytecode. As with other compiler projects, it is split into:

- √ a front-end (done). Converts bytecode to SSA form, stored as instances of `SSANode`.
- √ a middle-end (still very limited). Does the optimization tasks. Currently it can perform inlining, code duplication is the biggest thing that is lacking)
- √ a back-end (to be done) recreates bytecodes from SSA form. Responsible for converting some special selectors to the extended bytecodes.

Similar to the Smalltalk compiler's node hierarchy but more low-lovel (gotos instead of structured control flow).

First hierarchy level

SSANode	
StatementNode	assignments, sends, branches
ValueNode	intermediate values in a computation (variables, message sends)

A ValueNode can be converted to a StatementNode with a decorator, the EffectNode.

StatementNode

StatementNode	
BranchNode	branch unconditionally to another basic block
ConditionalNode	branch conditionally to another basic block
EffectNode	decorator for ValueNode
AssignmentNode	also holds def-use chains
ReturnNode	method returns
BlockReturnNode	block returns
SequenceNode	groups instances of StatementNode
BasicBlock	fundamental unit for data-flow analyses
SSASuperblockNode	used by inlining and code duplication
SSAMethodNode	
VoidNode	for dead-code elimination

A ConditionalNode also holds the profiling information from the virtual machine.

ValueNode

ValueNode	
ConstantNode	
KnownTypeNode	decorator for the type inferencer
MessageSendNode	message sends not considered by the inliner
ReceiverNode	self
RestrictNode	assigns type information to other nodes
ThisContextNode	thisContext
PhiFunctionNode	for SSA form
PrimitiveNode	holds the primitive's error code
VariableNode	another big class hierarchy, quite intuitive

Instance of MessageSendNode also hold profiling information from the virtual machine.

Using the `SSANode` hierarchy is similar to using the Refactoring Browser's parse nodes. For example:

- √ both have methods to replace a node with another
- √ both heavily use visitors. In AOStA, an optimization pass is implemented with a subclass of `SSANodeVisitor`: the `Optimizer` class invokes many visitors in succession.

The next slides will present a simple constant-propagation pass.

There are a few predefined visitors which order the basic blocks in different orders. Picking the correct one can speed up the optimization very much.

The system computes dominance information (which basic blocks must execute before which) as part of converting into SSA, and this information is available to the visitors.

- √ visiting the dominator tree in breadth-first order
- √ visiting the dominator tree in depth-first order
- √ visiting basic blocks in no particular order (fast, useful for local analyses such as inlining)

This is one of the first passes that are run. In particular, it runs before dead-code elimination because it exposes dead assignments.

```
Kernel.AOStA defineClass: #ConstantPropagation
    superclass: #{Kernel.AOStA.SSABreadthFirstDominatorTreeVisitor}
    indexedType: #none
    private: false
    instanceVariableNames: 'knownConstants '
    classInstanceVariableNames: ''
    imports: ''
    category: 'AOStA'
```

Let's start by initializing the `knownConstants` instance variable. It will map a constant temporary variable's `TempNode` to the corresponding `ConstantNode`.

```
initialize
    super initialize.
    knownConstants := Dictionary new
```

Then, whenever we find a `TempNode` with an entry in `knownConstants`, we can replace it with the `ConstantNode`.

```
visitTempNode: aNode
    knownConstants at: aNode ifPresent: [ :constant |
        aNode replaceWith: constant ].
```

The visitor method for `AssignmentNode` can optimize if the right-hand of the assignment is a `ConstantNode`; if it is a ϕ-function, we can check if the operands of the ϕ-function are all the same.

If the optimization succeeds, we propagate the constant into all the uses. Even if it does not, we recurse by calling the superclass implementation, in order to replace the temporaries in the right-hand.

```
visitAssignmentNode: aNode
    aNode isPhiFunctionAssignmentNode
        ifTrue: [
            self
                visitPhiFunctionNode: aNode value
                for: aNode variableNode].
    aNode value isConstantNode
         ifTrue: [self propagateConstantAssignmentNode: aNode]
         ifFalse: [super visitAssignmentNode: aNode]
```

This method propagates the constant into all its uses.

```
propagateConstantAssignmentNode: aNode
    knownConstants at: aNode variableNode put: aNode value.
    aNode uses copy do:
        [:each |
        | statement |
        statement := each statement.
        statement isNil ifFalse: [self visitNode: statement]]
```

If the right-hand of the assignment is a ϕ-function, this method is invoked, which checks if the operands of the ϕ-function are all the same.

```
visitPhiFunctionNode: aNode for: aVariableNode
    | value seed constant |
    value := seed := Object new.
    constant := aNode inputs allSatisfy: [ :each || thisValue |
        thisValue := knownConstants at: each ifAbsent: [ ^self ].
        value == seed
            ifTrue: [ value := thisValue. true ]
            ifFalse: [ value = thisValue ]].

    constant ifTrue: [
        knownConstants at: aVariableNode put: value.
        aNode replaceWith: value ]
```

The last method in the constant propagation pass prevents recursion into ϕ-function nodes, because we do not want `TempNodes` to be replaced there too.

```
visitPhiFunctionNode: aNode
    "Do nothing, we process phi functions from visitAssignmentNode:"
```

The pass is made of only four methods!

AOStA is:

- √ fun: writing compiler code in Smalltalk is so much simpler!
- √ relatively easy to support in a virtual machine, especially if it already supports Polymorphic Inline Caching.
- √ relatively portable: the interface between AOStA-enabled virtual machines and the optimizer is well defined.

The architecture's potential is very high—of course, as with most experimental projects, the main problem is the implementors' lack of free time. You're welcome!