Superinstructions for the SOMns Interpreter

Friedrich Weber
fweber@posteo.de

October 12, 2017

My project consists of two parts:

- A dynamic analysis for automatically identifying promising candidates for superinstructions. The underlying idea of the candidate detection heuristic is explained in Section 1.
- Integration of several candidates for superinstructions into the SOMns interpreter. A short description of the implemented candidates can be found in Section 2.

1 Heuristic

I integrated the superinstruction detection into the dynamic metrics tool coming with SOMns. Thus, it can be invoked by passing the `-dm` option to the SOMns interpreter. For example, the invocation

```
./som -G -dm ./core-lib/Hello.ns
```

prints a sorted list of superinstruction candidates to the file metrics/superinstruction-candidates.txt. The option `-G` is necessary to disable JIT compilation using Graal.

The heuristic consists of two stages, namely the *context collection* stage and the *candidate detection* stage, which are explained separately in the following.

1.1 Context Collection

During the execution of the program in question, the modified SOMns interpreter counts the number of activations for each AST node. In this, it also takes the Java type of the activation result into account. Consider, for example, the following code snippet:

```
value:: value + 1
```

which will be parsed into an AST similar to

```
LocalVariableWriteNode
  EagerBinaryPrimitiveNode
    LocalVariableReadNode
    IntegerLiteralNode
    AdditionPrim
```

In the following, we assume that the value slot stores a numeric value represented by a Java Long and consider a hypothetical program run in which the LocalVariableReadNode is activated 100 times. During execution, the dynamic analysis records the fact that the LocalVariableReadNode has been activated 100 times, each time producing a Long value. This dynamic analysis is implemented in the tools.dym.nodes.TypeCountingNode and tools.dym.profiles.TypeCounter classes.

After execution, the analysis constructs a set of so-called *activation contexts*. Each activation context consists of two parts, a *trace* and a *type*:

```
ActivationContext(type=..., trace=...)
```

Application contexts are implemented in the Java class tools.dym.superinstructions. ActivationContext.

The *type* denotes the Java type of the node activation result. The *trace* is a alternating sequence of strings and integers and has the general form

$$[C_0, s_0, C_1, ..., s_{n-1}, C_n]$$

in which all C_k are Java node class names (Strings) and all s_k are child slot indices (Integers). We call the number n the *trace length*.

A trace represents the execution environment in which a node was activated. The rightmost node class name (i.e. C_n) is the class name of the activated node. Its preceding node class name C_{n-1} denotes its parent AST node and s_{n-1} denotes the slot in which the activated node can be found in the parent's list of children. Similarly, the class name C_{n-2} denotes its grandparent AST node and s_{n-2} denotes the slot in which the activated node's parent can be found in the grandparent's list of children. This construction is continued up to a predefined length. Thus, an activation trace contains the ancestors of a node up to a predefined depth.

Instances of EagerPrimitive nodes are handled separately: Here, the heuristic constructs an artificial node class which contains the operation name, e.g. an EagerBinaryPrimitiveNode node with an addition primitive is denoted as PrimitiveOperation:+.

In my heuristic, I decided to construct traces up to $n = 2$. Thus, the following activation contexts are constructed for the activation of LocalVariableReadNode in the exemplary AST:

```
ActivationContext(type=Long, trace=[LocalVariableReadNode])
ActivationContext(type=Long, trace=[PrimitiveOperation:+, 0,
                                    LocalVariableReadNode])
ActivationContext(type=Long, trace=[LocalVariableWriteNode, 0,
                                    PrimitiveOperation:+, 0,
                                    LocalVariableReadNode])
```

Each activation context is annotated with its total number of activations. In the example, all three activation contexts are annotated with 100 (because the corresponding LocalVariableReadNode was activated 100 times).

The result of the context collection stage is a map mapping activation contexts to activation counts. The context collector is implemented in the Java class tools.dym.superinstructions.ContextCollector.

1.2 Candidate Detection

Based on this map, the heuristic aims to detect superinstruction candidates. A superinstruction candidate is represented by an AST subtree which could be replaced by just one new node (constituting a superinstruction). The underlying objective is to find candidates for which the resulting superinstruction would be activated a large number of times. In the optimal case, this would result in a considerable speedup as the number of dynamic dispatches at runtime is significantly reduced.

First, the heuristic extracts all activation contexts with traces of length $n = 2$, sorts them by their activation count in descending order and extracts a predefined number of contexts with the highest activation counts (currently 100).

Consequently, these contexts are all of the form

`ActivationContext(type=`$type$`, trace=`$[C_0, s_0, C_1, s_1, C_2]$`)`

and represent the 100 AST subtrees that were activated most frequently during the execution. For each of these activation contexts, the heuristic attempts to construct a superinstruction candidate.

Let us consider an example and choose the context *ctx* which we define as follows:

```
ActivationContext(type=Long, trace=[LocalVariableWriteNode, 0,
                                    PrimitiveOperation:+, 0,
                                    LocalVariableReadNode])
```

Based on the activation context, the heuristic decides that the superinstruction candidate should have the following shape:

```
LocalVariableWriteNode
  PrimitiveOperation:+
    LocalVariableReadNode (of type Long)
    <unknown>
```

As PrimitiveOperation:+ is a binary operation, it requires two node children. However, only the node class located in the first child slot is fixed. The second child class is yet to be determined.

In order to determine the second child node class of the PrimitiveOperation:+ node, the heuristic searches through *all* collected activation contexts to find contexts that match the following shape:

```
ActivationContext(type=?, trace=[LocalVariableWriteNode, 0,
                                 PrimitiveOperation:+, 1,
                                 ?])
```

In other words, the heuristic searches for contexts whose traces have

```
[LocalVariableWriteNode, 0, PrimitiveOperation:+, 1]
```

as a prefix. The matching activation contexts are sorted by their activation count in descending order. Then, the heuristic chooses the activation context with the highest number of activations.

In our example, let us assume that the most frequently activated activation context is the following:

```
ActivationContext(type=Long, trace=[LocalVariableWriteNode, 0
                                    PrimitiveOperation:+, 1,
                                    IntegerLiteralNode])
```

As a consequence, the heuristic decides to use the IntegerLiteralNode class as the second child of the PrimitiveOperation:+ node and the resulting superinstruction candidate looks as follows:

```
LocalVariableWriteNode
  PrimitiveOperation:+
    LocalVariableReadNode (of type Long)
    IntegerLiteralNode (of type Long)
```

Please note that the example shows a simplified scenario, as the LocalVariableWriteNode node class only has one child slot. If C_0 references a node class with more than one child slot, the remaining child slots of the candidate are determined similarly to above.

It might happen that a child slot could not be filled because it has never been activated during the execution (which means that no activation contexts of that slot exist). In that case, the slot is filled with a placeholder value "?".

After having constructed one candidate for each of the top 100 superinstruction contexts, the candidates are written to the file metrics/superinstruction-candidates.txt.

Please refer to the source code of tools.dym.superinstructions.CandidateDetector for more information.

2 Superinstructions

I have implemented five superinstructions in the SOMns interpreter which I chose after running the detection heuristic on core-lib/Benchmarks/AllSmall.ns[1]. The respective implementations can be found in the package som.interpreter.nodes.superinstructions. For each superinstruction, a *replacement routine* takes care of replacing matching subtrees with the respective superinstruction at runtime.

Superinstructions can be globally disabled by setting the som.superinstructions option to false:

```
./som -G -Dsom.superinstructions=false ...
```

2.1 WhileSmallerEqualThanArgumentNode

This superinstruction represents a while loop whose guard checks that a local numeric variable is less than or equal to a local argument. It replaces the subtree

```
WhileInlinedLiteralsNode (with expectedBool == true)
  EagerBinaryPrimitiveNode
    LocalVariableReadNode (of type Long)
    LocalArgumentReadNode (of type Long)
    LessThanOrEqualPrim
  ExpressionNode
```

[1] This file corresponds to All.ns but excludes the expensive PageRank and GraphSearch runs.

with the superinstruction

```
WhileSmallerEqualThanArgumentNode
  ExpressionNode
```

The replacement routine can be found in som.interpreter.nodes.specialized.whileloops .WhileInlinedLiteralsNode.

An exemplary instance of the superinstruction is found in core-lib/Kernel.ns:

```
[ i <= limit ] whileTrue: [ block value: i. i:: i + step ]
```

2.2 IfSumGreaterNode

This superinstruction is somewhat specific: It replaces the subtree

```
IfInlinedLiteralsNode (expectedBool == true)
  EagerBinaryPrimitiveNode
    EagerBinaryPrimitiveNode
      LocalVariableReadNode (of type double)
      LocalVariableReadNode (of type double)
      AdditionPrim
    DoubleLiteralNode
    GreaterThanPrim
  ExpressionNode
```

with the superinstruction

```
IfSumGreaterNode
  ExpressionNode
```

In other words, the superinstruction encapsulates the expression

```
(left + right > than) ifTrue: body
```

whereas left and right are local variables, and than is a Double literal. The replacement routine can be found in som.interpreter.nodes.specialized.IfInlinedLiteralsNode.

An instance (and, as it seems, the only instance) of the superinstruction can be found in core-lib/Benchmarks/Mandelbrot.ns:

```
(zrzr + zizi > 4.0) ifTrue: [
  ...
].
```

2.3 Optimized Variable Writes

The following three superinstructions correspond to optimized versions of variable write operations. Consequently, the respective replacement routines can be found in the som.interpreter.nodes.LocalVariableNode.LocalVariableWriteNode class.

2.3.1 IncrementOperationNode

This superinstruction represents the increment of a local Long variable by a fixed integer. In other words, the following subtree:

```
LocalVariableWriteNode
  EagerBinaryPrimitiveNode
    LocalVariableReadNode (of type Long)
    IntegerLiteralNode (of type Long)
    AdditionPrim
```

is replaced with a superinstruction `IncrementOperationNode`. Note that the frame slot referenced by the LocalVariableWriteNode node needs to coincide with the frame slot referenced by the LocalVariableReadNode node.

This superinstruction comes in handy if the executed program increments a local counter variable, e.g. via the aforementioned statement `value:: value + 1`. The directory *superinstructions/increment* contains two interesting case studies:

- deoptimize-increment.ns illustrates a scenario in which a subtree is optimized to a IncrementOperationNode superinstruction, but has to be deoptimized to the original subtree shortly after. This is because the slot type changes from Long to Object.
- no-optimization.ns illustrates the case in which a possible optimization is not performed because the slot type is changed during the execution of the program.

2.3.2 AssignProductToVariableNode

This superinstruction represents the multiplication of two local variables of type Double whereas the result is stored in another local variable. In other words, the subtree

```
LocalVariableWriteNode
  EagerBinaryPrimitiveNode
    LocalVariableReadNode
    LocalVariableReadNode
    MultiplicationPrim
```

is replaced with just one node:

```
AssignProductToVariableNode
```

An exemplary usage of the superinstruction can be found in the file corelib/Benchmarks/Mandelbrot.ns:

```
zrzr:: zr * zr
```

2.4 AssignSubtractionResultNode

This superinstruction encapsulates the following sequence of actions:

- sending two different messages to two different receivers

- subtracting the two result values
- writing the result (a Double) to a local slot

In other words, the superinstruction replaces the subtree

```
LocalVariableWriteNode
  EagerBinaryPrimitiveNode
    GenericMessageSendNode
    GenericMessageSendNode
    SubtractionPrim
```

with the following superinstruction:

```
AssignSubtractionResultNode
  GenericMessageSendNode
  GenericMessageSendNode
```

An exemplary instance of the superinstruction can be found in core-lib/Benchmarks/NBody.ns:

```
dx:: iBody x - jBody x
```