AOStA

and some ideas

Marcus Denker
Software Composition Group
University of Berne
Overview

Part I:

AOSTA Revisited: A Short Overview

Part II:

A Code Generator for AOSTA

Part III:

Beyond Performance
Part I: AOStA

- AOStA: Adaptive Optimizations for Smalltalk

- Profiled execution to identify hot spots

- Compiles to optimized bytecode

- Dynamic deoptimization (debugging)

- Written in Smalltalk
**Profiled execution**: two areas for JIT-compiled methods

- The optimized area works as usual
- In the unoptimized area methods have a counter for each send and backward branch

**Collecting type information**
- via Polymorphic Inline Caches
- need to be readable from Smalltalk

**Optimizations:**
- e.g., Inlining
  Specializations for known types
Status

2003:

- Design
- Frontend: Bytecode transformed to SSA
- Middle: SSA Framework, sample optimizers
- For VisualWorks

2004:

- Backend: transformation out of SSA
- Simple Code Generator
- Done with Squeak
Part II: More about the backend

- Short introduction to SSA (Static Single Assignment)

- Two steps:
  1) Deconstruction of SSA
  2) Code generation

- Some examples
SSA - Static Single Assignment Form

- SSA: Each Variable has **one** assignment

- If control flow merges, we need to select the variable from the path we came from

```
examplePhiSimple
| a b |
a := 3.
a = 3
    ifTrue: [ b := 1]
    ifFalse: [ b := 2 ].
^b
```

```
1
  t1a := 3
  if not (t1a = 3) goto BB2

2
  t2c := 2

3
  t2a := 1
  goto BB4

4
  t2b := PHI(t2a, t2c)
  s3a := t2b
  ^s3a
```
SSA

- Very nice for many optimizations
- but: Code generation not possible directly
  --> Need to remove virtual selector functions (PHI-functions)

Two step code generation

1) Deconstruction of SSA
2) Code generation
SSA Deconstruction

Canonical method:

- $a_1 = 1$
- $a_2 = 2$
- $a_3 = \text{PHI}(a_1, a_2)$

Problems:
- Wrong after some optimizations
- Copies need to be removed
Phi-Congruency Method


**Idea:** Transform program that all variables are the same in PHI:

\[ a_1 = \text{PHI}(a_1, a_1) \quad \rightarrow \quad a_1 = a_1 \]

- Insert copies
- Renaming
Phi-Congruency: Overview

- Two step process:

  insert copies if needed
  
  TSSA → CSSA

  rename variables, delete PHI.
  
  CSSA → non-SSA

- Nice properties:

  - Without any optimizations, no copies are needed
  - Simple heuristics for copy placement

Number of copies

  canonical: ~16000
  
  new: ~1000  (without Opt: 0)
IRBuilder

Symbolic Assembler

Example:

```plaintext
| ir aCompiledMethod |
ir := IRBuilder new
  rargs: #(self);
  pushLiteral: 1;
  returnTop;
  ir.
aCompiledMethod := ir compiledMethod.
```

Execute:

```plaintext
aCompiledMethod valueWithReceiver: nil arguments: #()
```

Install in the system:

```plaintext
Float addSelector: #test withMethod: aCompiledMethod.
```
```
examplePhiSimple
 | a b |
 a := 3.
 (a = 3)ifTrue: [b := 1]
   ifFalse: [b := 2].
 ^b
```

```
BB9: [  
t1a := 3.
   if not (t1a = 3) goto BB18]
BB15: [  
t2a := 1.
   goto BB20]
BB18: [  
t2c := 2:]  
BB20: [  
t2b := PHI(t2a, t2c).
   s3a := t2b.
   ^s3a]
```
Part III: TODO/Ideas

TODO... lots. e.g. dynamic deoptimization

Possible experiments:

-> AOSTA on Squeak with Jitter
-> Does it make sense with just an interpreter?
-> Exupery as a backend

All these are related to performance.

Question: What else could be possible?
Runtime Translation as a System Service

- Enables more late binding

Example:

- iVars are accessed via offsets
- offsets are calculated at compile time
- makes changes and experiments harder

- Make a MOP practical

- Allows a much simpler System
MOPs and other strange stuff

MOP: Meta Object Protocol.

Idea: Provide an API for changing the language semantics and implementation at runtime.

(e.g., meaning of inheritance)

For Squeak: MetaClassTalk

- Nice, but slow
- A runtime translator could regain performance

Example: ClassBoxes
Two Kinds of Bytecode

"Image" level Vs. Interpreter Level

- Image level bytecode can be simple:
  => No optimizations at all

- Image level bytecode and interpreter bytecode could even be different:
  => Latebinding of the execution format

- Why not just use the AST?
"2 Worlds"

**Software-Engineering**
- AST instead of Bytecode
- late bound
- no optimizations

**Execution**
- bytecode or binary
- optimized
- late binding resolved

Translator