Status Talk

Interpretation and Compilation Techniques for Role-based Programming Languages

By Lars Schütze

Advisor: Prof. Jerónimo Castrillón Mazo
Co-advisor: Prof. Uwe Aßmann
Motivation

Published in [Schütze2017]
Variation Points for Language Implementation

- **Application level**
  - Generally executable
  - Logic executed by VM
  - Performance impact

- **Virtual Machine level**
  - Tied to specific VM
  - Only necessary control flow
  - Language agnostic optimizations
    - Compiler intrinsic
    - Internal data structures
Roles, Layers and Aspects

Role-Orientation

Context

- Role1
  - m1()
- Role2
  - m1()

Context

- Class
  - m1()

Aspect-Orientation

- Class
  - m1()

Advice

target(Class) && call(m1)
Summary of Optimizations of Related Approaches

- **Context-Oriented Programming**
  - JIT compiler intrinsic
  - Composition and inlining
  - Polymorphic and adaptive call sites

- **Aspect-Oriented Programming**
  - Reduce weave time
  - Lazy activation and optimization
  - Virtual machine adaptations

![Diagram showing the layers of a software system with focus on Semantic Gap]
Context-Oriented Programming

- Layers and partial methods
  - Dynamic and global scopes
  - Before / after / around advices

- Layer composition
  - Lookup defined on current active layers
  - Control-flow dependent
  - Sideway composition incur overhead

- Traditional optimizations
  - Cache and invalidate lookup information

[Lincke2011]
Optimizations in Context-Oriented Programming

- Meta-Tracing JIT compilation in ContextPyPy [Pape2016]
  - Guards protect specialized code
  - Promote layer composition (lookup)
    - JIT compiler intrinsic
    - Enforced specialization of trace

- Efficient layer activation in ContextJS [Krahm2012]
  - Composition of partial methods
  - Remove wrapper and inline composition

- Composition-aware call sites in JCop [Appeltauer2010]
  - Composition cached at call site
  - Re-composition updates call site targets
Aspect-Oriented Programming

- Different specifications
  - Pointcut
  - Advice
  - Join point (shadow)

- Aspect weaving and deployment
  - Not fully computable at weave-time
  - Runtime checks incur overhead
  - Control-flow and instance-local aspects (roles) most challenging

https://www.slideshare.net/koneru9999/aspect-oriented-programming-introduction, slide 10
Join Point Optimizations in AOP

- Compiler-compiler [Masuhara2003]
  - Reduce runtime checks with partial statically woven program
  - Expression interpreter and partial evaluation
  - Test-and-execute as conditional branch
    - Conditional either static or dynamic value

- Envelope-based weaving [Bockisch2005]
  - Envelope replaces original method
    - Same signature and name
    - Original method is encapsulated
  - Weave envelope method instead of call sites
  - Faster aspect compiler
Aspect Deployment Optimizations in AOP

- Minimal hooks [Popovici2003]
  - Dynamic weaving using JIT compiler
    - Non-optimizing baseline compiler
    - Compiles always on execution
  - Separate base and advice code
  - Weave hooks at native code

- Lazy envelopes [Bockisch2006]
  - Envelopes implemented in high level IR (HIR)
  - Different method compilation strategies
    - Method invalidation upon advice activation
    - Lazy recompilation

```
1. // run-time checks
2. CMP JTOC[weaver],0
3. BEQ noCallback
4. MOV S0, PR[activeThread]
5. CMP S0 [aopLock],0
6. BNE noCallback
7. MOV TO,JTOC[fieldModTags]
8. CMP TO [ARRAY'LENGTH],fieldId
9. BLE noCallback
10. ...// weaver callback
11. ...
12. noCallback:
13. ...// method body
```

```
m() # m()
  normal method envelope
```
Virtual Machine with Aspect Support

- Compiler adaptations in RVM
  - Optimizing compiler
  - New bytecodes in HIR
    - aaitpush
    - beginadvice / endadvice
    - Invokevirtualadvice

- Instance-local aspects (roles)
  - Clone TIB for instance
  - m() cannot be inlined anymore

[11]

Dealing with Roles at Runtime

- Map roles to types and patterns
  - Conceptual Design [Fowler1997]
  - Modeling Language [Steimann2000]
  - Framework Design [Riehle2000]

- Roles as Language Construct [Graversen2006]
  - Delegation and split objects
  - Configurable dispatch
  - Optimization considerations
Dispatching with Contextual Roles

- Role Relationship Pattern [Fowler1997]

- Who is responsible for dispatch?

![Role Diagram]

- Person
- PersonRole
- Group
- Manager
- Engineer
- Salesman
## Contextual Role-based Programming Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Year</strong></td>
<td><strong>Approach</strong></td>
</tr>
<tr>
<td>powerJava</td>
<td>2003</td>
<td>Translation</td>
</tr>
<tr>
<td>EpsilonJ</td>
<td>2008</td>
<td>Translation</td>
</tr>
<tr>
<td>ScalaRoles</td>
<td>2008</td>
<td>Library</td>
</tr>
<tr>
<td>SCROLL</td>
<td>2014</td>
<td>Library</td>
</tr>
<tr>
<td>JHelena</td>
<td>2014</td>
<td>Translation</td>
</tr>
<tr>
<td>LyRT</td>
<td>2016</td>
<td>Library</td>
</tr>
</tbody>
</table>
Analyzing Role-based Programming Languages

- Different implementation strategies
  - Role model
  - Dispatch

- Benchmark with context-dependent roles

Published in [Schütze2017]
Performance Results

Published in [Schütze2017]
Performance Characteristics of Role Languages

ObjectTeams:
- Data Types: 30%
- Runtime: 22%
- Weaving: 41%
- Benchmark: 7%

SCROLL:
- Data Types: 16%
- Data Structures: 12%
- Role Comparison: 64%
- Benchmark: 8%

LyRT:
- Data Types: 89%
- Weaving: 11%

All approaches miss efficient implementation of dispatch
Contextual Roles with ObjectTeams

- **Role Model**
  - Roles as inner classes
  - Role class associated to base class
  - Small pointcut language

- **Language implementation**
  - Compiler
    - Generate role dispatch code
  - Weaver
    - Load binding meta-data
    - Dispatch base classes
  - Runtime
Dynamic Dispatch in ObjectTeams

- Same code pattern always used

```java
class Role playedBy BaseClass {
    void withFee() {
        base.someMethod();
    }
    void withdraw() {
        replace withFee();
    }
}
```

Indirections everywhere:

```java
switch( id ) {
    case ...
        buildStack1();
        LIFTING();
        ParameterMapping();
        RoleMethod1();
    case ...
        buildStack2();
        RoleMethod2();
}
```
Dynamic Dispatch in ObjectTeams (cont.)

- Problems
  - Unnecessary method calls
  - Code execution depends on runtime arguments
  - Argument collectors and wrapper types
  - Costly context access with every call

- Solution
  - Runtime generated dispatch plan
  - Primitive types
  - Fast and cheap context access
Role Dispatch Plans at Run Time

- Combine meta-data and runtime information
  - Binding information and context
  - Dead code elimination (DCE) by construction
- Select and combine role methods
  - Combinator graph [Rose2009]
- Optimizable by virtual machine
  - Constant-folding, inlining
  - Reuse plans
Optimizing Dispatch in Role-Oriented Programs

Base Method

Arguments

$\alpha^B_{(i)} : i < |A^B_{(*)}|$

$\alpha^B_{(0)} \sim \text{target type}$

Translation
Polymorphism
[Herrmann2004]

Role Method
Optimizing Dispatch in Role-Oriented Programs

- Replace callin with base call
  - Next replace callin or base method
  - Not determinable

- Call graph pruning
  - Absent base call

- Dispatch plan is compile time constant
Results

- Not only applicable to ObjectTeams
- The more reuse the more speedup
- Team fluctuation ~ 3x slowdown
  - Invalidation using switchpoints
  - Translation polymorphism dependent on team instance
Semantic Gap

- **Application level**
  - Generally executable
  - Logic executed by VM
  - Performance impact

- **Virtual Machine level**
  - Tied to specific VM
  - Only necessary control flow
  - Language agnostic optimizations
    - Compiler intrinsic
    - Internal data structures
Modern Virtual Machines

VM with JIT Code Generation

Source Program

Compiler

Runtime Info

Input

Interpreter

Output

Binary

Run Time

Development Time

Meta-Tracing [Bolz2009]

if := cnt + 1

0

1

cnt

Tree Rewriting and Partial Evaluation

[Wuerthinger2012, Wuerthinger2013]

cnt := 0

cnt := cnt + 1

if :=
Tree Rewriting and Partial Evaluation

- Replace AST nodes from profiling feedback
  - Rewriting nodes given a type lattice

- Generate code from interpreter definition
  - Compilation of tree assumes constant nodes
  - Partial evaluation of AST yields high level IR
  - 1st Futamura projection [Futamura1999]

- Partial evaluation [Kotzmann2005]
  - Escape analysis
  - Removes heap allocations
Towards a Minimal Pivot Role Language

- **Language features**
  - Context-dependent roles
  - Static and dynamic roles
  - Unrelated and related player types

- **Object Model and Dispatch**
  - Prototypes in SELF [Chambers1989]
    - Direct tagged pointers vs indirect through object table
  - Prototypes with multiple dispatch [Salzman2005]
    - Multi-methods describe behavior of multiple interacting objects
  - Prototypes describe stateful behavior
  - Acquisition dispatch [Graversen2006]
Optimizing role dispatch with dispatch plans
Published


Submitted

References


References (cont.)


OSM in Truffle

Figure 3: Components of the Truffle OSM

```
var a = {};
// a's shape is {}
a.x = 4;
// a's shape is {x:int}
a.y = 2;
// a's shape is {x:int, y: int}
var b = {x: "one", y: "two"};
// b's shape is {x: String, y: String}
var c = {x: "one", y: 2};
// c's shape is {x: String, y: int}
```

Figure 4: JavaScript example demonstrating the relation between objects and shapes

Figure 5: Shape tree for the JavaScript code in Figure 4