Towards a Full Multiple-Inheritance Virtual Machine

Roland Ducournau & Floréal Morandat

LIRMM
the University of Montpellier & CNRS

MASPEGHI/ICOOOLPS’10
Maribor, June 22, 2010
Plan

1. Context and Motivations
2. Virtual Machine Principle
3. Experiments
4. Conclusion and prospects
Plan

1. Context and Motivations
2. Virtual Machine Principle
3. Experiments
4. Conclusion and prospects
Motivations (1/2)

Why be afraid of multiple inheritance?

- from the **semantic** standpoint (so-called conflicts);
  - this is another story...
- from the **efficiency** standpoint.
  - my point today.

Alternatives to multiple inheritance

- multiple subtyping (**Java**);
- mixins/traits (**Scala**).

They are supposed to improve both aspects.
Motivations (1/2)

Why be afraid of multiple inheritance?

- from the semantic standpoint (so-called conflicts);
  - this is another story...
- from the efficiency standpoint.
  - my point today.

Alternatives to multiple inheritance

- multiple subtyping (Java);
- mixins/traits (Scala).

They are supposed to improve both aspects.
Our position

- this improvement is illusory,
- interfaces are as inefficient as full multiple inheritance (the only possible improvement involves attribute access)

Our project

- designing a virtual machine for full multiple inheritance,
- as efficient as JAVA or .NET systems.

Possible target language:
- think of SCALA without class/trait distinction.
**Context**

**Implementation of OO mechanisms**
- method invocation,
- attribute access,
- subtype testing.

**Modern runtime systems**
- static typing,
- dynamic class loading,
- adaptive compilers.

**Consequences**
- code optimization based on temporary CWA,
- possible recompilations when assumptions are questioned.
Context

Implementation of OO mechanisms
- method invocation,
- attribute access,
- subtype testing.

Modern runtime systems
- static typing,
- dynamic class loading,
- adaptive compilers.

Consequences
- code optimization based on temporary CWA,
- possible recompilations when assumptions are questioned.
Objectives

Design and simulation

- before hard coding
- design of
  - an object-oriented implementation,
  - a collection of optimizations,
  - a (re)compilation protocol,
- simulation and assessment
  - of the recompilation cost
  - of various optimizations,
  - on real class hierarchies.
- focus on multiple-inheritance effects
  - we do not consider (yet) other optimizations
Plan

1. Context and Motivations
2. Virtual Machine Principle
3. Experiments
4. Conclusion and prospects
## Principle

### Object representation
- Incremental, definitive object representation
- Based on perfect class hashing.

### Code sequences for OO mechanisms
- Single-subtyping code for constant-position sites, equals class-typed invocations in Java.
- Perfect-hashing code for variable-position sites, approximates interface-typed invocations in Java.

### Compilation and recompilation protocol
- Computes the object representation,
- Selects the appropriate code generation,
- Decides method recompilations.
Object representation (1/3)

Single-subtyping implementation

- methods/attributes introduced by a given class are grouped,
- superclasses are ordered,
- the superclass order is a prefix of the subclass order,
- invariable positions.
Object representation (1/3)

Single-subtyping implementation

- methods/attributes introduced by a given class are grouped,
- superclasses are ordered,
- the superclass order is a prefix of the subclass order,
- invariable positions.
Virtual Machine Principle

Object representation (1/3)

Single-subtyping implementation

- methods/attributes introduced by a given class are grouped,
- superclasses are ordered,
- the superclass order is a prefix of the subclass order,
- invariable positions.
Object representation (2/3)

Multiple subtyping and dynamic loading

- methods/attributes introduced by a given class are grouped,
- superclasses are ordered,
- the prefix condition does not hold for all direct superclasses but it can hold for at least one superclass,
- invariable positions when the prefix condition holds,
- the superclass order depends on the class loading order.

On a single-inheritance hierarchy

- the same layout as the single-subtyping implementation.
  \( \Rightarrow \) the multiple-inheritance overhead might concern only the multiple-inheritance cases
Object representation (3/3)

- $D$ satisfies the prefix condition wrt $B$ not $C$,
- $C$ has variable positions,
- $C$-typed invocations must be recompiled.
Object representation (3/3)

- $D$ satisfies the prefix condition wrt $B$ not $C$,
- $C$ has variable positions,
- $C$-typed invocations must be recompiled.
Object representation (3/3)

- $D$ satisfies the prefix condition wrt $B$ not $C$,
- $C$ has variable positions,
- $C$-typed invocations must be recompiled.
Object representation (3/3)

- $D$ satisfies the **prefix condition** wrt $B$ not $C$,
- $C$ has variable positions,
- $C$-typed invocations must be recompiled.
Compilation Protocol (1/2)

Code generation

- single-subtyping code for
  - *root*-invocations, ie methods introduced by the root class,
  - *self*-invocations, ie receiver typed by the current class, when this class has an invariable position (so far),

- perfect-hashing code for
  - *self*-invocations with variable-position classes,
  - non-*self*-invocations.

Possible recompilations

- for *self*-invocations of invariable-position attributes/methods,
- each class maintains the list of methods to recompile when its attribute/method group is moved.
Compilation Protocol (2/2)

At class loading

- greedy optimization for
  - selecting a preferred direct superclass (for prefix condition) (possibly different for attributes and methods)
  - computing the superclass order(s)
- update of the group positions of superclasses (not in the prefix)
  ⇒ newly moved groups
- (lazy) method (re)compilation
  - newly loaded
  - already compiled but invoking newly moved attributes/methods
Plan

1. Context and Motivations
2. Virtual Machine Principle
3. Experiments
4. Conclusion and prospects
Random class loading simulation

**Statistics on class hierarchy**

Number of
- classes, methods and attributes,
- methods that contain *self*-invocations,
- method/attribute invocation sites,
- root/self/other invocation sites.

**Statistics on random class loading**

Worst-case (ie maximum or minimum) number of
- classes, methods and attributes with variable/invariable positions,
- optimized/unoptimized invocation sites,
- method compilations and recompilations.
## Benchmarks and results

A single benchmark, the PRM compiler

### Recompilation numbers

<table>
<thead>
<tr>
<th>number of</th>
<th>recompilations</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>550</td>
</tr>
<tr>
<td>method definitions</td>
<td>4407</td>
</tr>
<tr>
<td></td>
<td>11%</td>
</tr>
<tr>
<td></td>
<td>6%</td>
</tr>
</tbody>
</table>

### Optimization numbers

<table>
<thead>
<tr>
<th>number of</th>
<th>optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>method <em>root</em>-invocations</td>
<td>100%</td>
</tr>
<tr>
<td>method <em>self</em>-invocation</td>
<td>91%</td>
</tr>
<tr>
<td>other method invocations</td>
<td>0%</td>
</tr>
<tr>
<td>attribute invocations</td>
<td>98%</td>
</tr>
</tbody>
</table>
Plan

1. Context and Motivations
2. Virtual Machine Principle
3. Experiments
4. Conclusion and prospects
Related Works

A. Myers [OOPSLA 1995]

An extension of his double compilation scheme to

- multiple compilations,
- dynamic loading and adaptive compilers,
- method invocation and subtype testing.
Conclusion and prospects

Conclusion

A promising simulation

- worst-case assessment (for the considered hierarchy...)
- very few recompilations (6%)
- attribute invocation as efficient as with multiple subtyping (97%)
- method invocation as efficient as with an intensive use of interfaces (25%)

⇒ as efficient as perfect hashing and attribute coloring

[OOPSLA’09]

Simulation is a promising approach

- compilers are atomic bombs, simulate them!
- random simulations are required to vary class loading orders,
- the key to empirical assessment of worst cases and scalability.
Prospects

Extending the random simulation

- finer-grain recompilation protocol
  - eg distinguishing between introduction class and static type
- optimization of non-self-invocations
  - \( \approx 75\% \) of all method invocations
  - static analysis (eg CHA, type analysis),
  - monomorphic calls as static calls.

More benchmarks

- you are welcome!

Simulation is a promising approach but

- eventually a real virtual machine will be needed...
That’s all folks!

drawings by F’murr.