

Partial Evaluation for Class-Based Object-Oriented Languages

Ulrik P. Schultz
Center for Pervasive Computing
University of Aarhus

22 May 2001

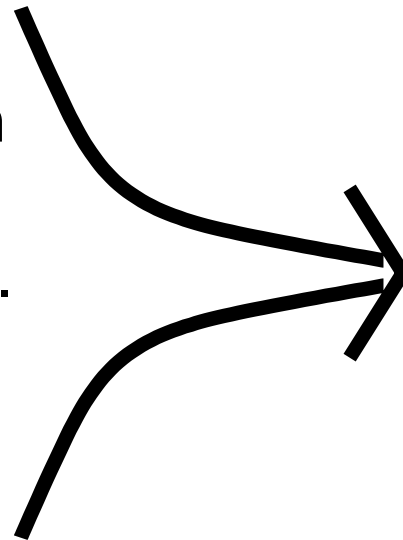
PE & OO

Partial Evaluation

inheritance vs. specialization

objects, classes, methods, ...

analysis & design



OOPE

Object-Oriented Paradigm

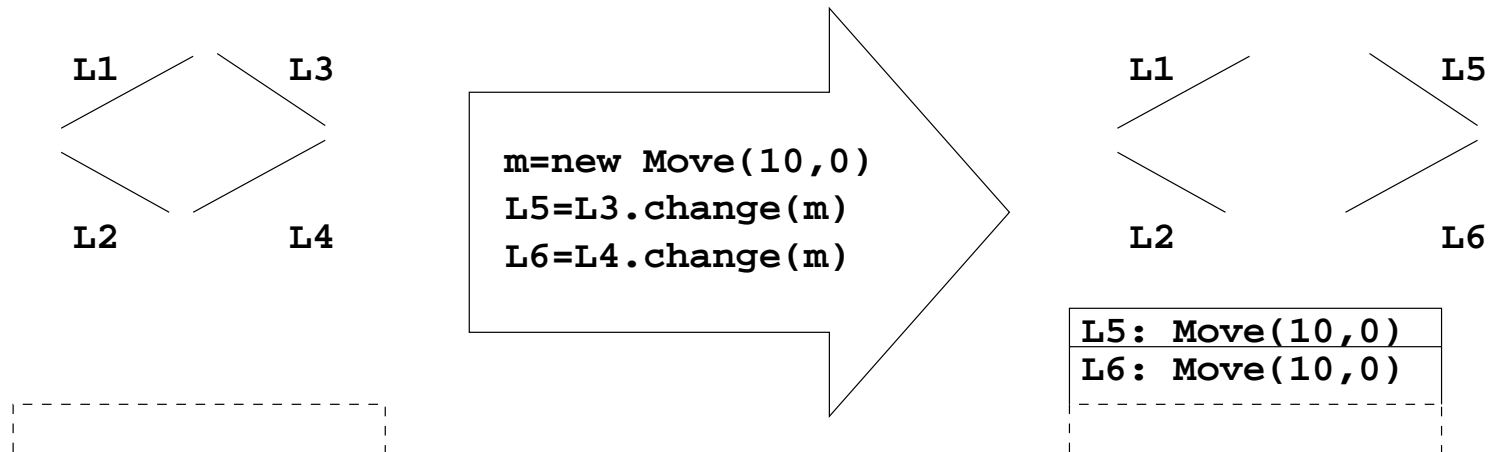
Overview

- Goals for OOPE
 - specialize programs for known input
 - specialize a program slice
- Earlier work
 - prototype PE for Java
 - specialization of typical object-oriented designs
- This work: minimal, formal, and scalable approach
- This presentation: explanation and motivation

Plan

1. An example: modifiable shapes
2. OOPE specialization approach
3. Formalization of OOPE
4. Scaling up OOPE
5. Discussion of speedups
6. Conclusion and perspectives

Example: modifiable shapes (command pattern)



Modifiable shapes, implementation

```
class Shape { Shape change(Mod m) { return ⊥; } }
```

```
class Point extends Shape {  
  int x, y;  
  Point(int x,int y) { this.x=x; this.y=y; }  
  Shape change(Mod m) { return m.mod(this); }  
}
```

```
class Mod { Point mod(Point x) { return ⊥; } }
```

```
class Swap extends Mod {  
  Point mod(Point p) { return new Point(p.y,p.x); }  
}
```

Example: `new Point(4,5).change(new Swap())`

Example continued

- A new kind of shape:

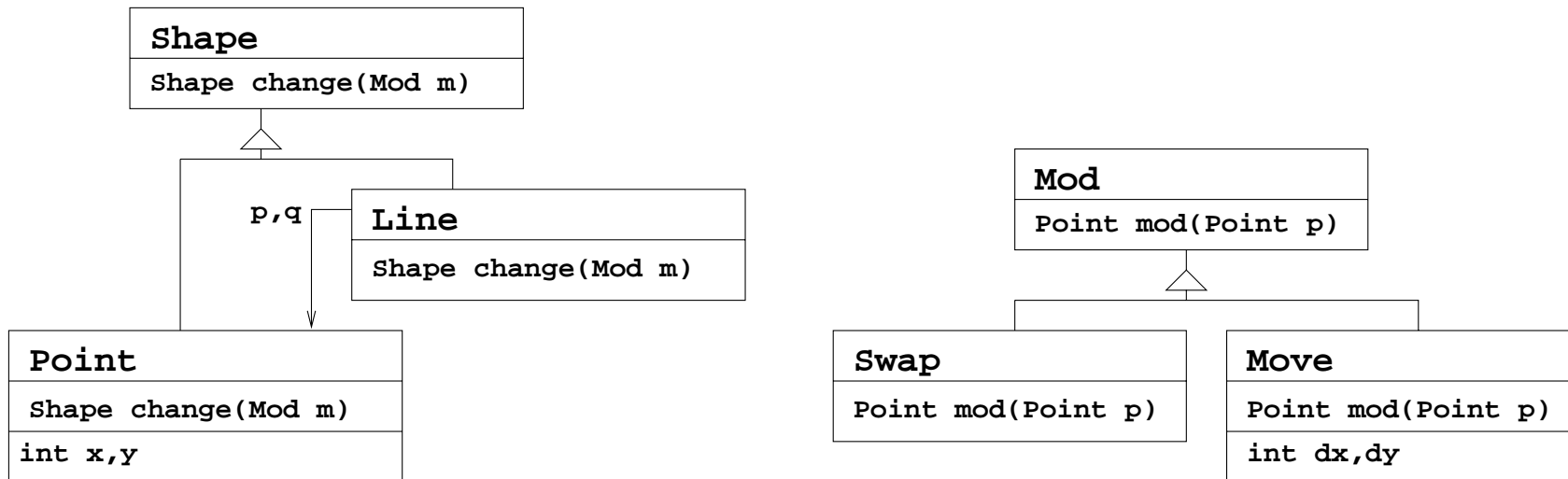
```
class Line extends Shape {
    Point p, q;
    Line(Point p,Point q) { this.p=p; this.q=q; }
    Shape change(Mod m) { return new Line( m.mod(this.p), m.mod(this.q) ); }
}
```

- A new kind of modification command:

```
class Move extends Mod {
    int dx, dy;
    Move(int x,int y) { this.dx=x; this.dy=y; }
    Point mod(Point p) { return new Point(p.x+this.dx,p.y+this.dy ); }
}
```

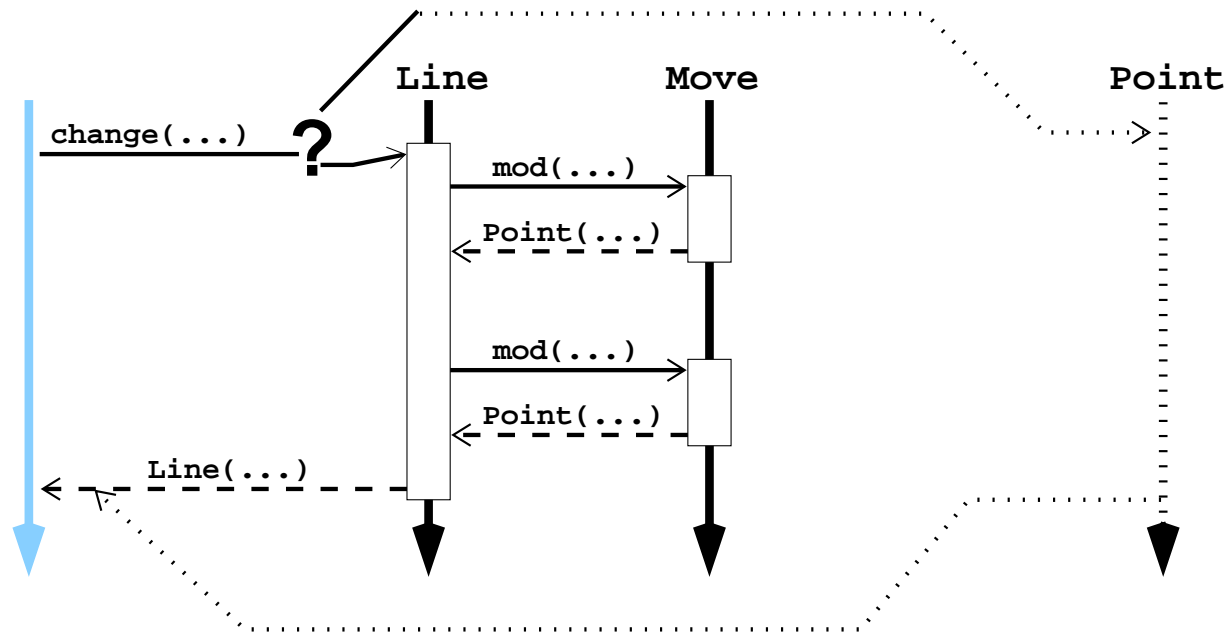
- Example: `new Line(p1,p2).change(new Move(1,1))`

Example overview



Sample use: Moving a shape to the right

```
Shape dyn_shape = ...; dyn_shape.change( new Move(1,0) );
```



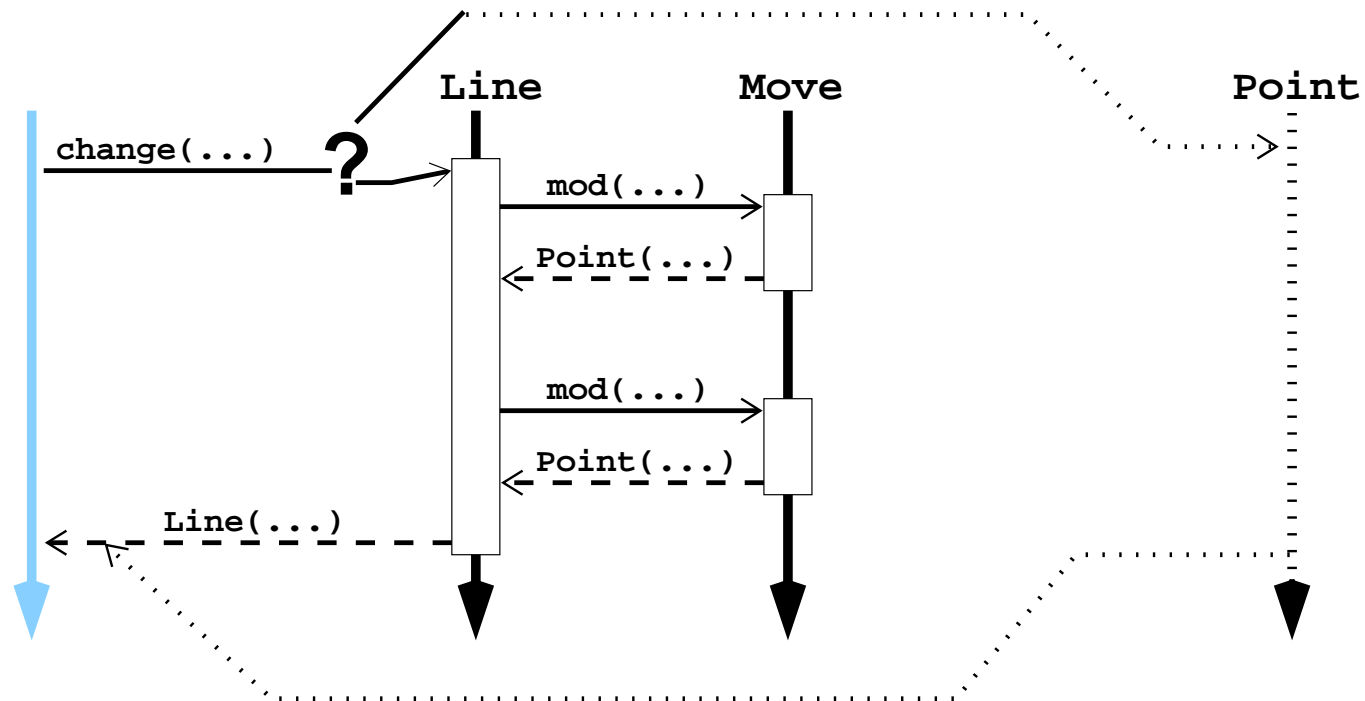
How to specialize this program?

Specializing an object-oriented program

- Execution of an OO program \approx sequence interactions between objects
- Static input data makes some of these interactions static
- OOPE: specialize away static interactions, residualize dynamic interactions

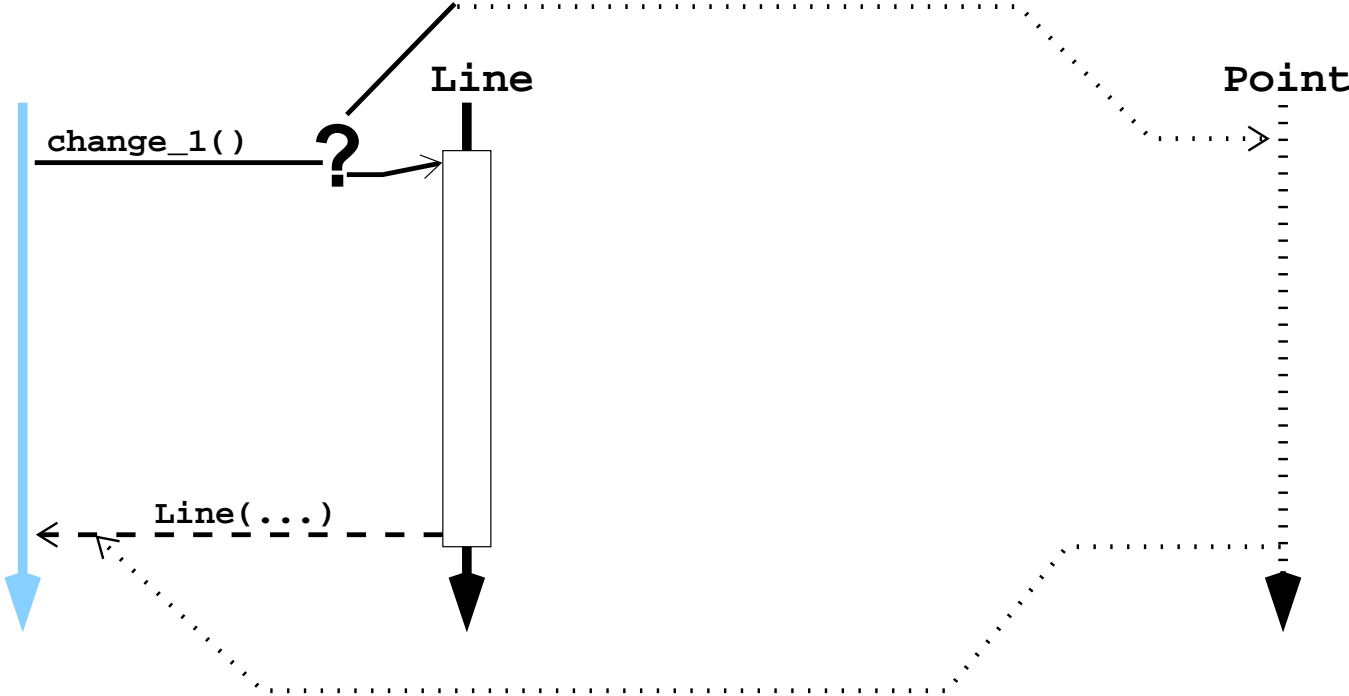
Concretely: specializing the shape example

```
Shape dyn_shape = ...; dyn_shape.change( new Move(1,0) );
```



Specialized version of the modifiable shape example

```
Shape dyn_shape = ...; dyn_shape.change_1();
```



OOPE

1: Specialize object interactions by generating specialized methods

?: How to generate specialized methods?

- specialize for static “this” and static formal parameters
- specialize field access, virtual dispatch, functional or imperative computations

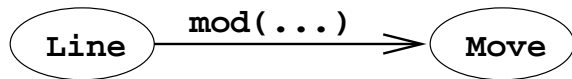
Specializing a method

- Static field accesses
 - propagate values, eliminate indirect references
- Static functional or imperative computations
 - reduce as usual
- Virtual dispatches
 - virtual dispatch \approx conditional over the receiver type

Specializing a virtual dispatch

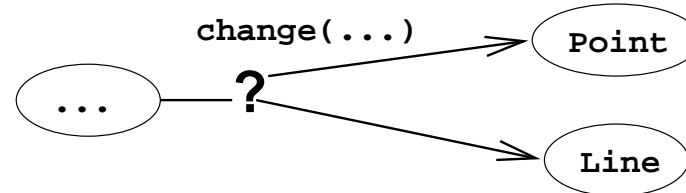
receiver has static "this"

`m.mod(this.p)`



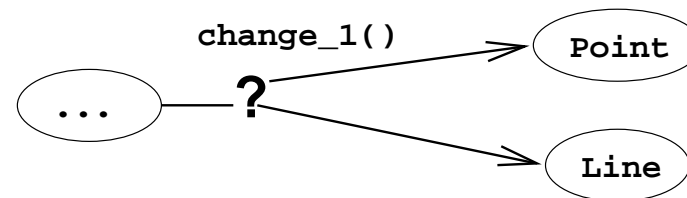
receiver has dynamic "this"

`dyn_shape.change(...)`



reduce virtual dispatch,
unfold `Move.mod` into caller

speculatively specialize each
`change` into `change_1`,
residualize call to `change_1`



OOPE

- 1:** Specialize object interactions by generating specialized methods
- 2:** Specialize methods for their arguments, by specializing field access, virtual dispatching, and other computations
- ?:** What to do with the specialized methods?

Putting the specialized methods back

- Modular specialization \Rightarrow must retain generic code
- Put methods directly into existing classes?
 - obscures program
 - complicates maintenance
 - may break encapsulation invariants

Idea: use aspect-oriented programming

- Observation: specialization *cross-cuts* the program class structure
- *Aspect-oriented programming*: modularly express program functionality that cross-cuts the program structure [Kiczales et al, ECOOP'97]
- General approach: produce specialized program in aspect language, use weaver to compile
- Concrete approach: use AspectJ language [Kiczales et al, ECOOP'01]

Result in AspectJ syntax

```
aspect Move_right {  
  
    introduction Shape {  
        Shape change_1() { return ⊥; }  
    }  
  
    introduction Point {  
        Shape change_1() {  
            return new Point(this.x+1,this.y+0);  
        }  
    }  
  
    introduction Line {  
        Shape change_1() {  
            return new Line(new Point(this.p.x+1,this.p.y+0),  
                            new Point(this.q.x+1,this.q.y+0));  
        }  
    }  
}
```

OOPE

- 1:** Specialize object interactions by generating specialized methods
- 2:** Specialize methods for their arguments, by specializing field access, virtual dispatching, and other computations
- 3:** Encapsulate specialized methods in an aspect

In the paper

- Formalization of OOPE for a small Java-like language
- Discussion of scaling up the formalized OOPE to realistic scenarios
- More specialization examples (strategy and visitor)

Formalization

- Minimal Java-like language without side-effects, off-line monovariant PE
- Difficult to concisely express specialization of virtual dispatch
- Well-annotatedness and BTA straightforward
- Proof of correctness is future work

Scaling up to realistic scenarios

- Formalization does not include side-effects
- Side-effects and higher-order values together are difficult
 - Use an alias analysis to track all objects [C-Mix]
 - Use alias and type information to convert virtual dispatches into conditionals (“defunctionalize”)
 - Works for Java-like language (statically typed, class based, simple type system)
- Precision of BTA: class-polyvariance, method-polyvariance, matching alias analysis, and partially static data [Tempo]

Experimental verification: JSpec

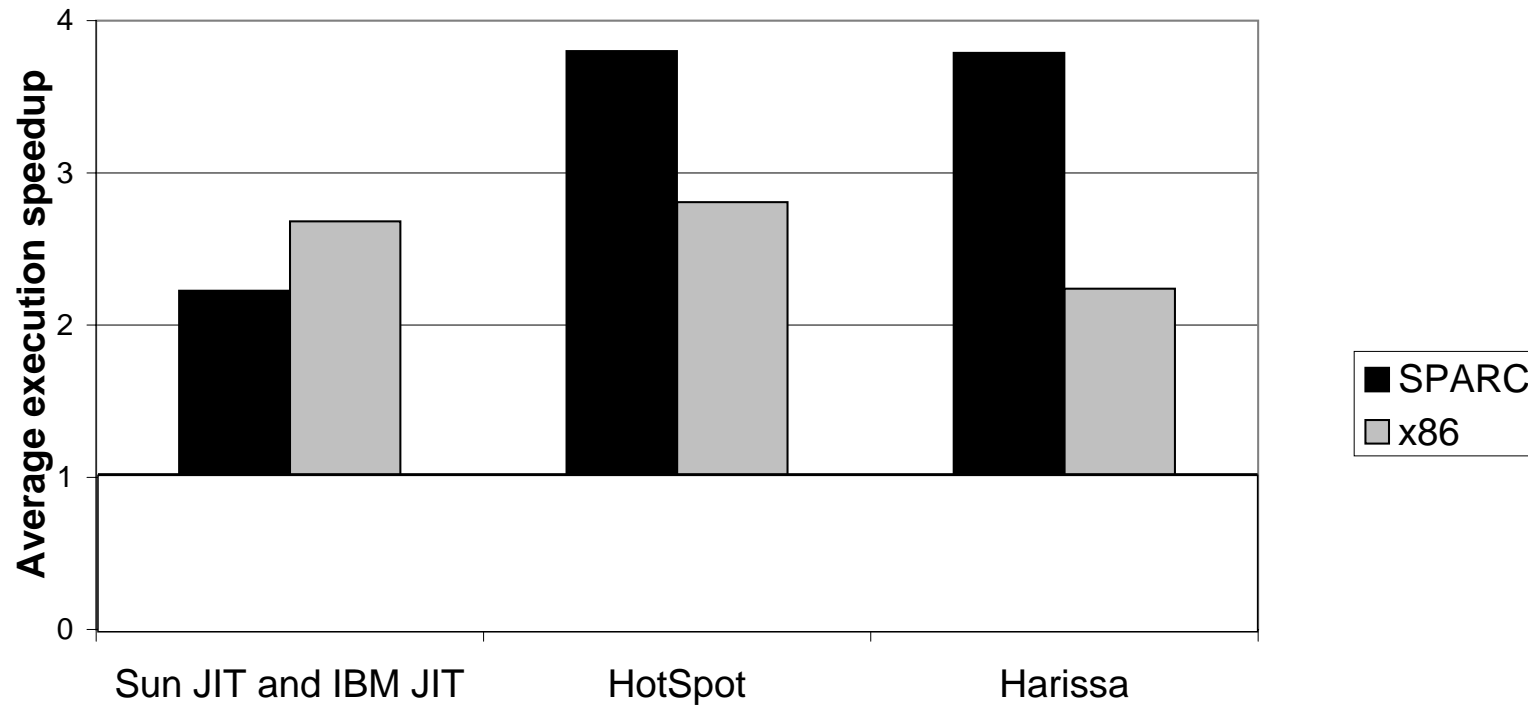
JSpec:

- PE for Java (excluding exception handlers and `finally`)
- Based on OOPE principles presented here; reuses Tempo through translation-based approach
- Has been successfully used to specialize “large” Java programs

Speedups?

- Object-oriented compilers such as HotSpot perform (limited) program specialization, guided by profiling
- OOPE (as embodied by JSpec):
 - more thorough and more aggressive
 - less general

Experiments with JSpec



Conclusion

- OOPE: specialize program by specializing object interactions, specialize interactions by specializing methods, and specialize a method by specializing field access, virtual dispatches, and other computations
- Aspect-oriented programming gives a convenient representation of the residual program
- OOPE has been formalized though not proven (this work), and scales up to realistic scenarios (JSpec)

Perspectives

- Improvements to OOPE as presented here. . .
 - arity raising (structure splitting)
 - modifying class definitions
 - . . .
- Specialization for other object-oriented languages
- Run-time specialization for Java

Resources

Publications on OOPE:

- [ECOOP99] Schultz, Lawall, Consel, Muller: Towards Automatic Specialization of Java Programs
- [ASE00] Schultz, Lawall, Consel: Specialization Patterns
- [PhD Dissertation] Schultz: Object-Oriented Software Engineering using Partial Evaluation

JSpec is available (version 0.2) from the JSpec home page:

<http://www.irisa.fr/compose/jspec>

All are accessible from <http://www.daimi.au.dk/~ups>

Inheritance vs. program specialization

- Differences
 - inheritance adds state and behavior
 - partial evaluation fixes state and removes generic behavior
- *Instantiation-time specialization* generates subclasses
- Appropriate form of inheritance could perhaps express method specialization

Instantiation-time specialization

Generic program

```
class Power {
  int e;
  Power(int i) { this.e=i; }
  int raise(int b) {
    int res=1, i=this.e;
    while(i-->0) res*=b;
    return res;
  }
}
...
dynamic_context(new Power(3))
```

Specialized program

```
class Power_3 {
  Power_3() { super(3); }
  int raise(int b) { super.raise(b); }
}
aspect Power_of_three {
  introduction Power {
    int raise_3(int b) {
      return b*b*b;
    }
  }
}
...
dynamic_context(new Power_3())
```

What about block structure?

Generic program	Incorrect specialization
<pre>class A { ... } class B { class C extends A { int f(...) { ... } } int f(...) { ... } A m() { return new C(); } } ... someA.m().f(...) ...</pre>	<pre>class A { ... } class B { class C extends A { int f(...) { ... } int f_1(...) { ... } } int f(...) { ... } A m() { return new C(); } } ... ((C)(someA.m())).f_1(...) ...</pre>

Solution: interface lifting

Correctly specialized program

```
class A { ... }
abstract class C extends A {
  abstract int f_1(...);
}
class B {
  class X extends C {
    int f(...) { ... }
    int f_1(...) { ... }
  }
  int f(...) { ... }
  A m() { return new X(); }
}
... ((C)(someA.m())).f_1(...) ...
```

JSpec

- $\text{JSpec} = \text{AspectJ} \circ \text{Assirah} \circ \text{Tempo}' \circ \text{Harissa} \circ \text{javac}$
- $\text{Tempo}' = \text{Tempo}$ with structure polyvariance and special inlining (but still monovariant alias analysis)

- | Java | C |
|----------------------------|--|
| <code>object.field</code> | <code>object->field_id</code> |
| <code>object.m(x,y)</code> | <code>object->_svtable->m_id(object,x,y)</code>
(transformed into conditional by Tempo) |

Generic shape example in JSpec

```
class Point extends Shape {  
  int x,y;  
  Point(int x,int y) { this.x=x; this.y=y; }  
  void change(Mod m) { m.mod(this); }  
}
```

```
class Move extends Mod {  
  int dx,dy;  
  Move(int x,int y) { p.x+=this.dx; p.y+=this.dy; }  
}
```

Specialized shape example in JSpec

```
aspect Move_right {  
  ...  
  introduction Point {  
    void change_1() { this.x+=1; this.y+=0; }  
  }  
  ...  
}
```

Extending the formalization

- Side-effects with monovariant classes:
slightly more complex, perhaps more convincing, not really useful in practice
- Class and method polyvariance plus partially static objects:
definitely more complex, but doable; convincing?
- All of the above (including polyvariant alias analysis):
really hard, convincing, but useful as a formalization?

Encapsulation invariants

Generic program

```
class Client {
  ...
  void access(Safe s) {
    ... s.read(key) ...
  }
}

class Safe {
  private Object content;
  private int key;
  ...
  Object read(int k) {
    return k==this.key?this.content:null;
  }
}
```

Specialized program

```
class Client {
  ...
  void access_1(Safe s) {
    ... s.read_1() ...
  }
}

class Safe {
  ...
  Object read_1() {
    return this.content;
  }
}
```