

Visitor-Oriented Programming

Jens Palsberg

UCLA

Joint work with Thomas VanDrunen (Purdue University)

Overview

Idea: Everything is a visitor and everything can be visited.

	function	—	visitor	—	object
easy to write new functions?	Yes	—	Yes	—	No
easy to extend the data types?	No	—	Yes	—	Yes

Contributions:

- a visitor calculus
- Peripaton: a language based on the calculus, with an implementation
- translations: lambda calculus \rightarrow visitor calculus \rightarrow Java
- four versions of the 1st translation, all programmed in Peripaton
 - programming idioms based on visitors
- correctness proof for the 2nd translation:
 - type preservation
 - behavior preservation

Related work

In the visitor calculus, each value is a visitor and every visitor call uses double dispatching.

- Castagna, Ghelli, and Longo, I&C 1995: $\lambda\&$ -calculus = λ -calculus plus overloaded functions with multiple dispatching.
- Leavens and Millstein, OOPSLA 1998: Tuple = multiple dispatch added to a conventional object-oriented language
- Millstein and Chambers, ECOOP 1999: Dubious = an object-oriented core language with symmetric multimethods, a module system, and a type system
- Clifton, Leavens, Chambers, and Millstein, OOPSLA 2000: MultiJava = Java with multimethods

The Visitor Pattern

```
class Expression {
    void accept(Visitor v) { }
}
class Abstraction
extends Expression {
    String id;
    Expression Expr;
    void accept(Visitor v) {
        v.visit(this);
    }
}
class Application
extends Expression {
    Expression expr1;
    Expression expr2;
    void accept(Visitor v) {
        v.visit(this);
    }
}
class LambdaVar
extends Expression {
    String id;
    void accept(Visitor v) {
        v.visit(this);
    }
}
```

```
class Visitor {
    void visit(Abstraction e) { }
    void visit(Application e) { }
    void visit(LambdaVar e) { }
}
class TypeChecker extends Visitor {
    void visit(Abstraction e) { ... }
    void visit(Application e) {
        ...
        e.expr1.accept(this);
        ...
        e.expr1.accept(this);
        ...
    }
    void visit(LambdaVar e) { ... }
}
class Inliner extends Visitor { ... }
class PrettyPrinter extends Visitor { ... }
```

The Visitor Calculus

p	$::=$	(\bar{c}, e)	<i>programs</i>
e	$::=$		<i>expressions</i>
		$(e e)$	<i>invocations</i>
		$\text{new } t[\bar{e}]$	<i>creations</i>
		x	<i>field references</i>
		acceptor	<i>parameter references</i>
c	$::=$	$t : t\{\bar{x}; \bar{m}\}$	<i>classes</i>
m	$::=$	$(t \mapsto e)$	<i>method mappings</i>

A pre-defined class: $\text{visitor } \{ (\text{visitor} \mapsto \text{acceptor}) \}$

Operational semantics:

$$\frac{\text{find-method}(\bar{c}, t_1, t_2) = (t \mapsto e) \quad \text{fields}(\bar{c}, t_1) = \bar{x}}{(\bar{c}, X \langle (\text{new } t_1[\bar{v}_1] \text{ new } t_2[\bar{v}_2]) \rangle) \rightarrow (\bar{c}, X \langle [\text{acceptor}, \bar{x}/\text{new } t_2[\bar{v}_2], \bar{v}_1] e \rangle)}$$

Example: $\lambda x.x$

```
cla0: visitor {  
  (visitor -> acceptor)  
}
```

```
new cla0[]
```

```
class visitor extends Object {  
  visitor() {}  
  visitor accept(visitor x) {  
    return x.visit_visitor(this);  
  }  
  visitor visit_visitor(visitor acceptor) {  
    return acceptor;  
  }  
  visitor visit_cla0(cla0 acceptor) {  
    return this.visit_visitor(acceptor);  
  }  
}  
class cla0 extends visitor {  
  cla0 () {  
    super();  
  }  
  visitor accept(visitor x) {  
    return x.visit_cla0(this);  
  }  
  visitor visit_visitor(visitor acceptor) {  
    return acceptor;  
  }  
}
```

```
new cla0()
```

Example: $\lambda f.\lambda x.fx$ in the visitor calculus

```
cla0: visitor {  
  f;  
  (visitor -> (f acceptor))  
}  
cla1: visitor {  
  (visitor -> new cla0[acceptor])  
}  
  
new cla1[]
```

Example: $\lambda f.\lambda x.fx$ in Java

```
class visitor extends Object {
    visitor() {}
    visitor accept(visitor x) {
        return x.visit_visitor(this);
    }
    visitor visit_visitor(visitor acceptor) {
        return acceptor;
    }
    visitor visit_cla1(cla1 acceptor) {
        return this.visit_visitor(acceptor);
    }
    visitor visit_cla0(cla0 acceptor) {
        return this.visit_visitor(acceptor);
    }
}
```

Example: $\lambda f.\lambda x.fx$ in Java

```
class cla0 extends visitor {
    visitor f;
    cla0(visitor f) { super(); this.f = f; }
    visitor accept(visitor x) {
        return x.visit_cla0(this);
    }
    visitor visit_visitor(visitor acceptor) {
        return acceptor.accept(f);
    }
}

class cla1 extends visitor {
    cla1() { super(); }
    visitor accept(visitor x) {
        return x.visit_cla1(this);
    }
    visitor visit_visitor(visitor acceptor) {
        return new cla0(acceptor);
    }
}
```

Peripaton

p	$::=$	$\bar{c} e$	<i>programs</i>
e	$::=$	$(e e)$	<i>expressions</i>
		$(e e < \bar{e} >)$	<i>invocations</i>
		$\text{new } t[\bar{e}]$	<i>invocations with extra, non-dispatching parameters</i>
		x	<i>creations</i>
		<i>acceptor</i>	<i>variable references</i>
		<i>this</i>	<i>dispatching parameter</i>
		$\text{let } (\bar{x} = \bar{e}) \text{ in } e \text{ end}$	<i>self references</i>
		$\{e\}.x$	<i>let expressions</i>
		<i>gensym</i>	<i>non-local field references</i>
		$\text{if } e = e \text{ then } e \text{ else } e$	<i>unique identifier creations</i>
c	$::=$	$t : t\{\bar{x}; \bar{m}\}$	<i>branch expressions</i>
m	$::=$	$(t \mapsto e)$	<i>classes</i>
		$(t < \bar{x} > \mapsto e)$	<i>method mappings</i>
			<i>ordinary mappings</i>
			<i>mappings with extra, non-dispatching parameters</i>

Example

```
Goal : visitor {  
  expr;  
}
```

```
Abstraction : visitor {  
  id;  
  expr;  
}
```

```
Application : visitor {  
  expr1;  
  expr2;  
}
```

```
LambdaIdentifier : visitor {  
  uid;  
}
```

From Functions to Visitors in Peripaton

```
EncodingVisitor : visitor { ...
  (Application <v, cl> ->
    let t1 = (this {acceptor}.expr1 <v, cl>),
        t2 = (this {acceptor}.expr2
              <v, {t1}.classList>)
    in
      new TargetGoal[{t2}.classList,
                    new Invocation[{t1}.expr,
                                    {t2}.expr]]
    end)
}
```

Removing non-local field references

```
Goal : visitor {  
  expr;  
  (GetExpr -> expr)  
}
```

```
GetExpr : visitor {}
```

```
EncodingVisitor : visitor { ...  
  (Application <v, cl> ->  
    let t1 = (this (acceptor new GetExpr1[])  
              <v, cl>),  
        t2 = (this (acceptor new GetExpr2[])  
              <v, (t1 new GetClassList[])>)  
    in  
      new TargetGoal[(t2 new GetClassList[]),  
                    new Invocation[  
                      (t1 new GetExpr[]),  
                      (t2 new GetExpr[])]]  
    end)  
}
```

Removing non-dispatching arguments

```
EncodingVisitor : visitor { ...
  (Application ->
    let t1 = (new EncodingVisitor[v, cl]
              (acceptor new GetExpr1[])),
        t2 = (new EncodingVisitor[
              v, (t1 new GetClassList[])]
              (acceptor new GetExpr2[]))
    in
      new TargetGoal[(t2 new GetClassList[]),
                    new Invocation[
                      (t1 new GetExpr[]),
                      (t2 new GetExpr[])]]
    end)
}
```

Removing let expressions

```
EncodingVisitor : visitor { ...
  (Application ->
    new TargetGoal[
      ((new EncodingVisitor[
        v,
        ((new EncodingVisitor[v, cl]
          (acceptor new GetExpr1[]))
          new GetClassList[]))
        (acceptor new GetExpr2[]))
      new GetClassList[]),
    new Invocation[
      ((new EncodingVisitor[v, cl]
        (acceptor new GetExpr1[]))
        new GetExpr[]),
      ((new EncodingVisitor[
        v,
        ((new EncodingVisitor[v, cl]
          (acceptor new GetExpr1[]))
          new GetClassList[]))
        (acceptor new GetExpr2[]))
        new GetExpr[])]])
  }
```

Conclusion

A foundation for visitor-oriented programming.

Contributions: a calculus, a language, idioms, translations, proofs.