Discussion Topics

- Single Dispatching
- Multiple Dispatching
- Predicate Dispatching
- Open Classes
- Generic Functions
- Code examples
- Predicate dispatching as multiple dispatching
- Efficient dispatching using DAGs

Reference Publications

"Efficient Multiple and Predicate Dispatching"

Craig Chambers and Weimin Chen, University of Washington

OOPSLA 1999

"Predicate Classes"

Craig Chambers, University of Washington

ECOOP 1993

"MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java"

Curtis Clifton, Gary T. Leavens, Craig Chambers, Todd Millstein

July 2000

Method Dispatching

- Methods as messages in OOP
- Method dispatching
  - Static
  - Dynamic
- Dispatching styles
  - based on receiver
  - based on arguments
  - based on receiver and arguments

Single Dispatching

- Dispatching based on
  - Run-time type of receiver object
  - Not based on run-time type of arguments
- Sometimes inconvenient to program
- Examples
  - Java, C++

Motivating Example

```
public class Shape {
   /****/
    public boolean intersect(Shape s) {
    /*…...*/
   }
}
public class Rectangle {
   /****/
    public boolean intersect(Rectangle s) {
    /*efficient code for two rectangles*/
   }
}
```

Example (contd)

```
Shape s1, s2;
Rectangle r1,r2;
r1 =  new Rectangle(…);
r2 = new Rectangle(…);
s1 = r1; s2 = r2;
r1.intersect(r2);
r1.intersect(s2);
s1.intersect(r2);
s1.intersect(s2);
```

One possible solution

```
public class Rectangle {
   /****/
   public boolean intersect(Shape s) {
     if (s instanceof Rectangle )
          this.intersect((Rectangle)s);
     else super.intersect(s);
   }
   public boolean intersect(Rectangle s) {
    /*efficient code for two rectangles*/
   }
}
```

Single Dispatching - drawbacks

- “instanceof” tedious and error-prone
- not easily extensible
  - what if a new shape is introduced in the example?

Double dispatching

- Double dispatching a solution
  - Visitor pattern
  - Still tedious

```
class Shape {
 bool intersect(Shape s) {
       s.beingIntersected(this);
 }
 bool intersectRect(Rectangle r) {
       return false;
 }
class Rectangle extends Shape {
  bool beingIntersected(Shape s) {
      s.intersectRect(this);
  }
  bool intersectRect(Rectangle r) {
     /******/
   }
}
```

Dispatching internals

- *Static overloading* of methods
  - Java, C++
  - intersect method in example
- Name, number and static argument types
- Statically overloaded methods belong to distinct *generic functions*

Generic functions

- A set of methods with the same signature
- Formal parameters all renamed to be the same
- Dispatching involves selecting the best matching method
- Member methods could be physically defined separately

Canonical Syntax of Generic Functions

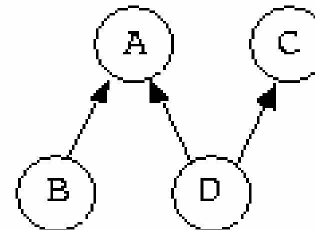
```
DF          ::= df Name(Name_1,...,Name_k) Case_1 or ... or Case_p
Case        ::= Conjunction => Method_1 ... Method_m
Conjunction ::= Atom_1 and ... and Atom_q
Atom        ::= Expr@Class | Expr@!Class
```

"Efficient Multiple and Predicate Dispatching"
Craig Chambers and Weimin Chen

Generic Function Example

Example class hierarchy:

```
object A;
object B isa A;
object C;
object D isa A, C;
```



Canonicalized dispatch function:

```
df Fun(f1, f2)
     (f1@A and f1.x@A and f1.x@!B and (f1.y=f2.y)@true) => m1   {c1}
  or (f1.x@B and f1@B) => m2                                     {c2}
  or (f1.x@B and f1@C and f2@A) => m2                            {c3}
  or (f1@C and f2@C) => m3                                       {c4}
  or (f1@C) => m4                                                {c5}
```

"Efficient Multiple and Predicate Dispatching"
Craig Chambers and Weimin Chen

GF in Shapes Example

```
public class Shape {
   /****/
    public boolean intersect(Shape s)
    {
    /*.....*/
   }
}
public class Rectangle {
   /****/
 public boolean intersect(Shape s)
 {/***/}
 public boolean intersect(Rectangle s)
 {
   /*efficient code for two rectangles*/
 }
}
```

m1

m2

m3

```
df intersect1(f1,f2)
    f1@Rectangle and f2@Shape => m2
or f1@Shape and f2@Shape => m1

df intersect2(f1,f2)
   f1@Rectangle and f2@Rectangle => m3
```

Multiple Dispatching

- Multimethods
  - *statictype@specializer*
- Method dispatch based on run-time type of arguments also
- “intersect(Shape@Rectangle *r*)”
  - Dynamically dispatch on formal parameter *r*.

Multiple dispatching example

```
public class Shape {
   /****/
   public boolean intersect(Shape s) {
    /*.....*/
   }
}
public class Rectangle {
   /****/
   public boolean intersect(Shape@Rectangle s) {
    /*efficient code for two rectangles*/
   }
}
```

GF in Shapes Example with multimethods

```
public class Shape {
   /****/
    public boolean intersect(Shape s)
    {
    /*…..*/
   }
}
public class Rectangle {
   /****/
public boolean
    intersect(Shape@Rectangle s)
 {
  /*efficient code for two rectangles*/
 }
}
```

m1

m2

```
df intersect1(f1,f2)
   f1@Shape and f2@Shape => m1
or f1@Rectangle and f2@Rectangle => m2
```

Dispatch Strategy

- Identify applicable methods based on argument type
  - Subset of methods in the GF
- Choose the most specific applicable method
  - Pointwise subtype of all applicable methods
- No applicable method
  - message-not-understood error
- Multiple most-specific methods
  - Message-ambiguous error

Dispatch strategy applied

```
Shape s1, s2;
Rectangle r1,r2;
r1 =  new Rectangle(…);
r2 = new Rectangle(…);
s1 = r1; s2 = r2;
r1.intersect(r2);
r1.intersect(s2);
s1.intersect(r2);
s1.intersect(s2);
```

```
df intersect1(f1,f2)
   f1@Shape and f2@Shape => m1
or f1@Rectangle and f2@Rectangle => m2
```

m1 – Shape::intersect(Shape)

m2 – Rectangle(Shape@Rectangle)

(Rectangle, Rectangle) a pointwise sub-type of (Shape,Shape)

Predicate classes

- Normal classes with predicate expressions
- Dynamic class hierarchy
- Implicit property-based classification
  - Based on run-time value,state,etc
- Classes follow normal inheritance rules

Predicate objects - Example

```
object buffer isa collection;
field elements(b@buffer); -- a queue of elements
field max_size(b@buffer); -- an integer
method length(b@buffer) { b.elements.length }
method is_empty(b@buffer) { b.length = 0 }
method is_full(b@buffer)  { b.length = b.max_size }

pred empty_buffer isa buffer when buffer.is_empty;
method get(b@empty_buffer) { ... } -- raise error or block caller

pred non_empty_buffer isa buffer when not(buffer.is_empty);

method get(b@non_empty_buffer) {
   remove_from_front(b.elements) }

pred full_buffer isa buffer when buffer.is_full;
method put(b@full_buffer, x) { ... } -- raise error or block caller

pred non_full_buffer isa buffer when not(buffer.is_full);
method put(b@non_full_buffer, x) {
   add_to_back(b.elements, x); }

pred partially_full_buffer isa
   non_empty_buffer, non_full_buffer;
```

GF for predicate dispatching

GF	::=	gf Name($Name_1$,...,$Name_k$) $Method_1$ ... $Method_n$	
Method	::=	**when** Pred { Body }	
Pred	::=	Expr@Class	*test whether Expr is an instance of Class or a subclass*
	|	test Expr	*test whether Expr (a boolean-valued expression) is true*
	|	Name := Expr	*bind Name to Expr, for use in later conjuncts and the method body*
	|	not Pred	*the negation of Pred*
	|	$Pred_1$ and $Pred_2$	*the conjunction of $Pred_1$ and $Pred_2$*
	|	$Pred_1$ or $Pred_2$	*the disjunction of $Pred_1$ and $Pred_2$*
	|	true	*the always-true default predicate*
Expr	::=	*host language expression: must have no externally visible side-effects*	
Class	::=	*host language class name*	
Name	::=	*host language identifier*	

Dispatch Strategy for predicate methods

- Generic function applied to argument tuple
- Set of applicable methods found by
  - binding arguments to formals
  - evaluating predicate for each method
- Most specific method selected
  - $m_1 \leq_{Method} m_2$ when m_1's predicate implies m_2's
  - m_1 is atleast as specific as m_2
- Errors
  - message-ambiguous and message-not-understood errors

Predicate dispatching as a superset

- All other models a subset of the predicate dispatching model
- Single dispatching
  - formal1@C_m
- Multiple dispatching
  - $formal_1$@$Class_{m1}$ and $formal_2$@$Class_{m2}$ ….
- Predicate classes
  - $formal_i$@*PredClass*, *PredClass* is a subclass of *Class* becomes

  $formal_i$@*Class* **and test** *Test, Test* is the predicate

Algorithm

- Time and space efficient dispatch functions for predicate dispatching model
- Three stages
  - Reduce general predicate dispatching model to simpler multiple dispatching model
  - Strategy for performing multiple-dispatching using a series of single dispatches (lookup DAG)
  - Implementation strategies for each of the single dispatches

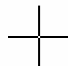
First stage

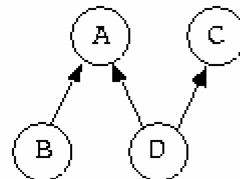
- Replace all **test** expressions with `Expr@True`
- Remove all `Name := Expr` clauses with `Expr`
- Convert predicates into disjunctive normal form
- Replace `not(Expr@Class)` with `Expr@!Class`
- Place each method `m`'s predicate into canonical form

  `Conjunction => m`
- Remove all atomic tests that are guaranteed to be true by static analysis
- Remove all conjunctions that are guaranteed to be false by static analysis
- Merge any duplicate conjunctions

Example class hierarchy:

```
object A;
object B isa A;
object C;
object D isa A, C;
```





Example source generic function:

gf Fun(f1, f2)
  when f1@A and t := f1.x and t@A and (not t@B) and f2.x@C and test(f1.y = f2.y) { ...m_1... }
  when f1.x@B and ((f1@B and f2.x@C) or (f1@C and f2@A)) { ...m_2... }
  when f1@C and f2@C { ...m_3... }
  when f1@C { ...m_4... }

Assumed static class information for expressions (*StaticClasses*):

f1: *AllClasses* – {D} = {A,B,C}
f2: *AllClasses* = {A,B,C,D}
f1.x: *AllClasses* = {A,B,C,D}
f2.x: *Subclasses*(C) = {C,D}
f1.y=f2.y: bool = {true,false}

Canonicalized dispatch function:

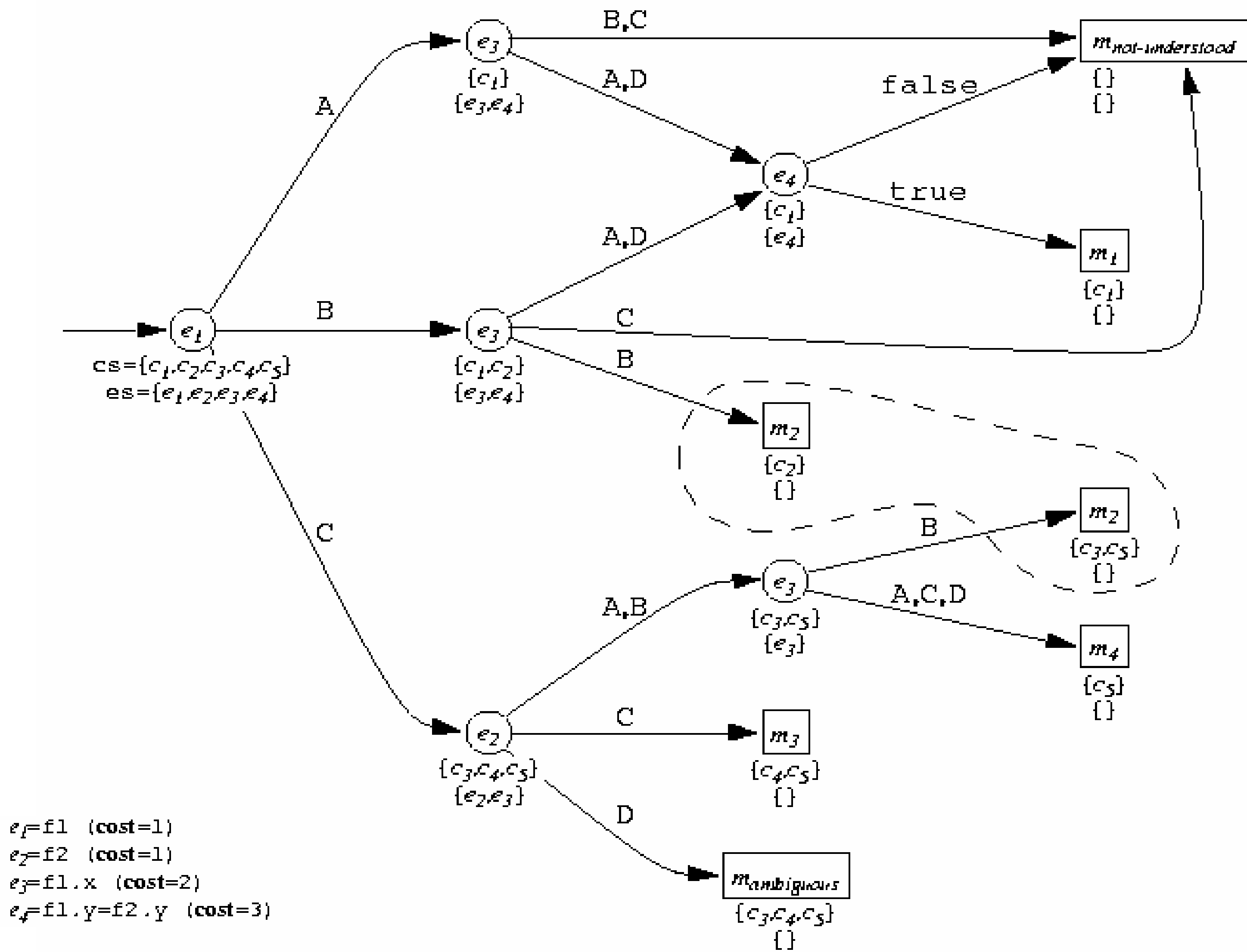
df Fun(f1, f2)
   (f1@A and f1.x@A and f1.x@!B and (f1.y=f2.y)@true) => m_1 {c_1}
 or (f1.x@B and f1@B) => m_2 {c_2}
 or (f1.x@B and f1@C and f2@A) => m_2 {c_3}
 or (f1@C and f2@C) => m_3 {c_4}
 or (f1@C) => m_4 {c_5}

Canonicalized expressions and assumed evaluation costs:

e_1=f1 (**cost**=1)
e_2=f2 (**cost**=1)
e_3=f1.x (**cost**=2)
e_4=f1.y=f2.y (**cost**=3)

Constraints on expression evaluation order (ignoring transitively implied constraints):

$e_1 \rightarrow_{\text{Expr}} e_3$; $e_3 \rightarrow_{\text{Expr}} e_1$; $\{e_1, e_3\} \rightarrow_{\text{Expr}} e_4$
$\{e_1, e_3\} \leq_{\text{Expr}} e_4$



B,C
A
A,D
false
true
A,D
B
C
B
C
A,B
B
A,C,D
C
D
$m_{not-understood}$
m_1
m_2
m_2
m_3
m_4
$m_{ambiguous}$
cs={c_1,c_2,c_3,c_4,c_5}
es={e_1,e_2,e_3,e_4}
e_1=f1 (cost=1)
e_2=f2 (cost=1)
e_3=f1.x (cost=2)
e_4=f1.y=f2.y (cost=3)