Multiple dispatching

If we can specialize first argument for particular classes,
why not any subset of arguments?
⇒ **multimethods**

```
method =(p1@Point, p2@Point) {
  p1.x = p2.x & { p1.y = p2.y } }

method =(p1@ColoredPoint, p2@ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Resolve overloading by:

- find all applicable methods
- select & invoke single most-specific application method

```
let p1 = new_point(...);
let p2 = new_point(...);
let cp1 = new_colored_point(...);
let cp2 = new_colored_point(...);

print(p1 = p2);   -- only Point×Point applies
print(p1 = cp2); -- ditto
print(cp1 = p2); -- ditto
print(cp1 = cp2);-- both apply, CP×CP wins
```


Multimethod overriding

One mm overrides another if:

- for all the other's specializers,
  the first method's corresponding specializers are equal to or descend from the other's, and
- and either:
  - at least one of the first's specializers strictly descends from the other's, or
  - one of the first method's formals is specialized while the other's is not

```
method foo(p1@Point, p2@ColoredPoint)
  overrides
method foo(p1@Point, p2@Point)

method foo(p1@ColoredPoint, p2@ColoredPoint)
  overrides
method foo(p1@ColoredPoint, p2)
```


Ambiguous methods

Two methods may be mutually ambiguous:
neither overrides the other

```
method foo(p1@Point, p2)
  ambiguous with
method foo(p1, p2@Point)

method foo(p1@ColoredPoint, p2@Point)
  ambiguous with
method foo(p1@Point, p2@ColoredPoint)
```

It's an error to send a message and find no single most specific application method

- no applicable methods: message not understood
- no single most-specific method: message ambiguous

Resolving ambiguities

Can resolve ambiguities by defining an overriding method

- method can do its own thing, or
  it can use directed resend to invoke one or more of existing methods

```
method foo(p1@ColoredPoint, p2@Point) { ... }
method foo(p1@Point, p2@ColoredPoint) { ... }

method foo(p1@ColoredPoint, p2@ColoredPoint) {
  -- select first foo method above:
  resend(p1, p2@Point) }
```

Examples of multimethods

Multimethods useful for binary operations

- 2+ arguments drawn from some abstract domain with several possible implementations

Examples:

- equality over comparable types
- < etc. comparisons over ordered types
- arithmetic over numbers
- `union`, `intersection`, etc. over set representations

Multimethods useful for cooperative operations even over different types

Examples:

- `display` for different kinds of shapes on different kinds of output devices
  - standard default implementation for each kind of shape
  - overridden with specialized implementations for certain devices
- operations taking flag constant objects, with different algorithms for different flags

Double-dispatching

If don't have multimethods, can simulate with double-dispatching [Ingalls, OOPSLA'86]

```
class Point {
  ...
  method =(p2) {
    p2.equalToPoint(self) }
  method equalToPoint(p1:Point) {
    p1.x = self.x & { p1.y = self.y } }
  method equalToColorPoint(p1:ColorPoint) {
    equalToPoint(p1) }
}

class ColorPoint {
  ...
  method =(p2) {
    p2.equalToColorPoint(self) }
  method equalToColorPoint(p1:ColorPoint) {
    equalToPoint(p1) &
      { p1.color = self.color } }
}
```


Evaluation of multimethods

Advantages:

- unify & generalize:
  - top-level procedures (no specialized arguments)
  - regular singly dispatched methods (specialize first argument)
  - overloaded methods (resolve overloading dynamically, not statically)
- naturally allow existing objects/classes to be extended with new behavior
- mechanize double-dispatching, make extensible w/o modifying existing code

Problems:

- what's the programming model?
- what's the encapsulation model?
- how to typecheck uses & definitions of multimethods?
- how to implement efficiently?

Multiple inheritance

Can inherit from several parent objects:

```
object Shape;
object Rectangle isa Shape;
object Rhombus isa Shape;
object Square isa Rectangle, Rhombus;
```

MI can be natural in application domain

MI can be useful for better factoring & reuse of code

But MI introduces semantic complications....

Ambiguities

Can get ambiguities due to MI, just like with MMs,
e.g. if two superclasses define methods,
neither of which overrides the other

```
object Rectangle isa Shape;
  method area(r@Rectangle) { ... }
object Rhombus isa Shape;
  method area(r@Rhombus) { ... }
object Square isa Rectangle, Rhombus;

let s := new_square(4);
... area(s) ...   → ambiguous!
```

Can resolve ambiguities by adding overriding method,
just like with MMs

```
  method area(s@Square) {
    resend(s@Rectangle) }
```


Diamond-shaped inheritance

How to determine method overriding if parent object is reachable along multiple inheritance paths?

```
object Shape;
  field center(s@Shape);
  method is_shape(s@Shape) { ... }
  method is_rectangular(s@Shape) { ... }
object Rectangle isa Shape;
  method is_rectangular(r@Rectangle) { ... }
  method area(r@Rectangle) { ... }
object Rhombus isa Shape;
  method area(r@Rhombus) { ... }
object Square isa Rectangle, Rhombus;

let s := new_square(4);
... is_shape(s) ...         → ambiguous?
... is_rectangular(s) ...   → ambiguous?
... area(s) ...             → ambiguous?
... center(s) ...           → ambiguous?
```

Different languages resolve these questions differently,
or forgo multiple inheritance entirely

Cecil semantics: inheritance as a partial ordering

In Cecil, inheritance graph defines a partial ordering over objects

- induces a corresponding partial ordering over methods based on the ordering of their specializers (pointwise on tuples of specializers, for multimethods)
- this partial ordering on methods is the overriding relationship

In Cecil, fields are present once,
independent of along how many paths they are inherited

In Cecil:

```
... is_shape(s) ...         → Shape's
... is_rectangular(s) ...   → Rectangle's
... area(s) ...             → ambiguous
... center(s) ...           → Shape's
```


To share or not to share?

What is the semantics of instance variables of superclass being inherited through multiple paths in diamond-shaped inheritance?

Options:

- top of diamond is shared
  - shared parents' fields included only once in subclasses
  - used in Cecil, C++ virtual inheritance
- top of diamond is duplicated
  - shared parents' fields duplicated along each path
  - normal C++ inheritance

Mixins

MI enables new programming idioms, including **mixins**
- highly factored abstract objects

Typically, organize attributes along independent axes
- several possible implementations (mixins) of each axis
- each concrete subclass picks one mixin for each axis

Example axes for shapes in a user interface:
- colored or not, bordered or not, titled or not, ...
- method for handling mouse clicks
- ...

```
object CheckBox
  isa Square,
      ColoredShape,
      BorderedShape,
      ShapeWithIcon,
      ClickableShape,
      ...;
```

Different mixin axes have common parent (e.g. Shape),
leading to diamond-shaped inheritance patterns

Summary, so far

Key OO ideas, across all OO languages
- ADTs: objects/classes, methods, fields, encapsulation
- method overloading, dynamic dispatching
- inheritance, method overriding
- idioms like factoring, abstract objects

Choices in dynamically typed OO languages, seen in Cecil
- class-based vs. classless model
- fields accessed directly or via messages
- single dispatching vs. multiple dispatching
- single inheritance vs. multiple inheritance
- class-based vs. module-based encapsulation
- presence of first-class functions

Typechecking OO Languages

In OO language, want static checking to ensure the absence of:
- message-not-understood errors
- message-ambiguous errors

Want to allow subclasses to be used in place of superclasses,
as long as doesn't create errors like the above

General strategy:
- declare (or infer) "types" of variables, "signatures" of operations
- check that assignments/initializations/returns only store "subtypes" of variable's type
- check that messages with particular actual argument types find at least one matching signature, guaranteeing the msg is understood unambiguously and specifying msg result type
- check that methods & fields completely and unambiguously implement covering signatures

Points of variation: what's a type? subtype? signature?

Simple idea: type = class

In many OO languages, types and classes/objects are the same
- subtyping = inheritance
- signatures derived directly from `method` & `field` decls, and from explicit `signature` decls
  - `signature` decls like abstract method decls
- overriding methods keep same argument & result types, for unspecialized formals

```
abstract object Point; -- Point is a type too
  signature x(p:Point):num;
  signature y(p:Point):num;
  method area(p@:Point):num { p.x * p.y }
  method rotate(p@:Point, amount:num):Point{.}
  method =(p1@:Point, p2@:Point):bool {
    p1.x = p2.x & { p1.y = p2.y } }

template object CartesianPoint isa Point;
  var field x(p@:CartesianPoint):num := 0;
  var field y(p@:CartesianPoint):num := 0;
  method new_cartesian_point(x0:num, y0:num
                            ):CartesianPoint {
    concrete object isa CartesianPoint ... }
```

Interfaces

Previous declarations imply the following interfaces

- can also be declared explicitly if desired

```
type Point;
  signature x(Point):num;
  signature y(Point):num;
  signature area(Point):num;
  signature rotate(Point, amount:num):Point;
  signature =(Point, Point):bool;

type CartesianPoint subtypes Point;
  -- no need to add x & y signatures for CartesianPoint
  signature set_x(CartesianPoint, num):void;
  signature set_y(CartesianPoint, num):void;
  signature new_cartesian_point(num, num
                               ):CartesianPoint;
```

Use these declarations for typechecking
initializations, assignments, and message sends

Examples

Legal:

```
let var p1:CartesianPoint :=
  new_cartesian_point(1,2);
let var p2:Point := new_cartesian_point(3,4);
p2 := p1;
p1.x := 20;
print(p1 = p2);
p2 := p2.rotate(60);
```

Illegal (static type error):

```
p2.x := 20;
p1 := p2;
p1.x := "hi there";
p1 := p1.rotate(60);
```


Method compatibility

Sufficient condition for safety:
overriding method has same argument and result types as overridden method

- ensures that using signature from originating method in checking calls won't be broken if overriding method selected at run-time

Relaxed conditions (**covariance**):

- allow argument & result types to be more *precise* in overriding method

Alternative relaxed conditions (**contravariance**):

- allow result types to be more *precise* in overriding method
- allow argument types to be more *general* in overriding method

Are either or both these relaxations safe?

Example

```
method copy(p@:Point):Point {
  ... }
method copy(p@:ColorPoint):ColorPoint {
  ... }

let p:Point := ...;
let q:Point := p.copy;
... q.x ...

method move(p@:Point, n:Num):void {
  ... }
method move(p@:ColorPoint, i:Int):void {
  ... }
method move(p@:Point3D, j:Any):void {
  ... }

let p:Point := ...;
move(p, 3.4);
let cp:ColorPoint := ...;
move(cp, 5);
let p3d:Point3D := ...;
move(p3d, "hi");
```

Overriding field types

If overriding a field with a method, or vice versa,
what kinds of changes can be made to the field's type?

```
var field foo(p@:Point):Foo;

method foo(p@:ColorPoint):Foo' { ... }
method set_foo(p@:ColorPoint, v:Foo'):void {
  ... }

field bar(p@:Point):Bar;

method bar(p@:ColorPoint):Bar' { ... }
```

What is the weakest safe relationship between `Foo` & `Foo'` and between `Bar` and `Bar'`?

Hint: compute the rules from the rules for methods...

Binary methods and typechecking

Another example (note single dispatching, as in most OOLs); is this OK?

```
abstract object Point;
method =(p1@:Point, p2:Point):bool {
  p1.x = p2.x & { p1.y = p2.y } }

abstract object ColoredPoint isa Point;
method =(p1@:ColoredPoint, p2:ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Example of use:

```
let p1:Point := new_cartesian_point(3,4);
let p2:Point := new_colored_point(3,4,"Blue");
print(p1 = p1);
print(p1 = p2);
print(p2 = p1);
print(p2 = p2);
```


Binary methods with multimethods

Another example (note multiple dispatching); is this OK?

```
abstract object Point;
method =(p1@:Point, p2@:Point):bool {
  p1.x = p2.x & { p1.y = p2.y } }

abstract object ColoredPoint isa Point;
method =(p1@:ColoredPoint, p2@:ColoredPoint) {
  resend & { p1.color = p2.color } }
```

Example of use:

```
let p1:Point := new_cartesian_point(3,4);
let p2:Point := new_colored_point(3,4,"Blue");
print(p1 = p1);
print(p1 = p2);
print(p2 = p1);
print(p2 = p2);
```

What semantics for mixed colored & plain points is desired?

Inheritance vs. subtyping

Classes & types, inheritance & subtyping are not the same:

- a class represents an **implementation**
  (a set of methods and fields),
  and inherits from other implementations to share code
- a type represents an **interface** (a set of signatures),
  and subtypes from other interfaces
- a class may **conform** to a type,
  meaning that the class implements the type's interface

In Cecil, can program these separately:

- `type`, `subtypes`, `signature` declarations for interfaces
- `representation`, `inherits`, `implementation` declarations for implementation
- `subtypes` declarations to connect implementations and interfaces

Common case: inheritance and subtyping is parallel

- `object` defines like-named `representation` & `type`
- `isa` defines parallel `inherits` & `subtypes`
- `method` defines parallel `implementation` & `signature`
- `@:` does parallel `@` and `:`

Example

```
-- types and signatures as before:
type Point; signature area...;
type CartesianPoint subtypes Point;

abstract representation PtRep
     conforms Point;
  implementation area(p@PtRep:Point):num ..
  implementation rotate(p@PtRep:Point,
                        amount:num):Point ...
  implementation =(p1@PtRep:Point,
                   p2@PtRep:Point):bool ...

template representation CartPtRep
     inherits PtRep conforms CartesianPoint;
  field implementation x(
     p@CartPtRep:CartesianPoint):num := 0;
  field implementation y(
     p@CartPtRep:CartesianPoint):num := 0;
  implementation new_cartesian_point(
       x0:num, y0:num):CartesianPoint {
    concrete representation
     inherits CartPtRep
     subtypes CartesianPoint { ... } }
```


Benefits of separating inheritance and subtyping

Clarity of thinking

Sensible to implement interface w/o inheriting code

Sometimes sensible to reuse code w/o being a subtype:

```
object Point;
  -- lots of Point methods here
  method =(p1@:Point, p2:Point):bool { ... }
object ColoredPoint inherits Point;
  -- inherit most Point methods
  method =(p1@:ColoredPoint,
           p2:ColoredPoint):bool { ... }
```

- can't mix Points & ColoredPoints (since ColoredPoints not subtypes of Points), but can still reuse code

Sometimes the two relations are opposite:

```
object deque subtypes stack;
  ... methods & signatures for push/pop/top,
      push_bottom/pop_bottom/bottom ...
object stack inherits deque;
  ... signatures for push/pop/top ...
```


Costs of separating inheritance and subtyping

Verbosity in common case ⇒ need syntactic sugar

Complexity

Difficult to typecheck safety of inheriting w/o subtyping

Checking method implementations

Last part of typechecking:
"check that methods & fields completely and unambiguously implement covering signatures"

Straightforward w/ single dispatching & monolithic classes:

- method's arguments at least as general as signature's, method's result at least as specific (contravariance)
  - **conformance**
- also check at each class that no method ambiguities from multiple inheritance
  - **consistency**
- also check at each concrete class that all signatures overridden with real methods
  - **completeness**

Example

```
abstract class Foo {
  signature bar(Num):Num;
  method foo(Int):Int { ... }
}--consistency of foo,bar OK

concrete class Bar {
  method baz(String):Void { ... }
}--completeness & consistency of baz OK

concrete class Baz subclasses Foo, Bar {
  method bar(Any):Int {...}; -- conformance OK
}--completeness & consistency of foo,bar,baz OK
```


Checking multimethods

With multimethods and/or ability to add methods to existing classes from the outside, need a more global check

Basic, brute-force strategy,
given whole program
(all `signature`, `method`, `class`, `inherits`, `type`, `subtypes` decls):

- foreach signature:
- foreach combination of concrete classes that pointwise conform to the signature's argument types:
- verify that there is a unique most-specific target method for this message on these argument classes (is **complete** & **consistent**) and the arguments **conform** to the method's formal types

Research questions:

- how to make the check efficient?
- how to make the check modular?

Example

Signature & method declarations:

```
signature =(Point,Point):bool;
method =(p1@:Point, p2@:Point):bool {...}
method =(p1@:ColoredPoint,
         p2@:ColoredPoint):bool {...}
```

Representations conforming to `Point`:
`Point`, `ColorPoint`, `Point3D`

Enumerating all combinations of these 3 representations shows = is completely & consistently & conformingly implemented