Cecil

• Inspired by Self:
  A classless object model
  Uniform use of messages for everything
• Inspired by CLOS:
  Multiple dispatching
  Extends both OO and functional programming styles
• Inspired by Trellis:
  Static typechecking
  Optional
  Support mixing dynamically and statically typed code

Bindings

• Use let to define (local and global) variables
  • add var keyword to allow assignment,
    otherwise immutable
  • must initialize at declaration
    let int := 1;
    let var count := 0;
    count := count + int;

Functions

• Use method to define functions
  • last expression evaluated is returned
  • can overload name for different numbers of
    arguments
    let var count := 0;
    method foo(a, b, c) {
      count := count + 1;
      let var d := a + b;
      let e := frob(d, c);
      d := d + e;
      d + 5
    }
    method frob(x, y) { x - frob(y) + 1 }
    method frob(x) { - x / 5 }

Closures: first-class functions

• Code in braces is a 0-argument function value
  let closure := { factorial(10) + 5 };
• Evaluation of closure delayed until eval is sent:
  eval(closure) $ 3628805
• To allow arguments, add &[x,y,z] prefix;
  invoke passing extra arguments to eval:
  let closure2 := &[n]{ factorial(n) + 5 };
  eval(closure2, 10) $ 3628805
• Like ML’s fn, Self’s blocks
  • anonymous, lexically scoped, first-class

Glitch: returning closures

• In current Cecil implementation, by default, closures cannot safely be returned out of
  their lexically enclosing scope
  • a glitch in the Vortex implementation, not the
    Cecil language
  • can crash Vortex mysteriously
  • prevents currying, compose, closures in data
    structures, ...
Avoiding the glitch

To allow a closure to be returned, use &&:

```c
def add_x(x) {
    &&(y) { x + y }
}
```

```c
let add_2 := add_x(2);
let add_5 := add_x(5);
eval(add_2, 4) // 6
```

Using closures in control structures

As in Self, all traditional (and many non-traditional) control structures are implemented as regular Cecil functions, with closures passed by callers supporting the necessary evaluation-only-on-demand

For simple lazy or repeated evaluation:

```c
if(test, { then_value }, { else_value })
```

```c
test1 & { test2 }
while(! test, { body })
```

More examples

For iteration with arguments:

```c
for(start, stop, &i) {
    body
}
```

```c
do(array, &elem) {
    body
}
```

For exception handling:

```c
do_associations(table, &key, value) {
    body
}
```

For 3-way branching:

```c
compare(i, j, {if_lt}, {if_eq}, {if_gt})
```

An example

```c
-- this is a factorial method

method factorial(n) {
    if(n = 0, { 1 },
        [ n * factorial(n - 1) ] )
    }
```

```c
-- call factorial here:

factorial(7)
```

Non-local returns

Support exiting a method early with a non-local return from a nested closure

```c
{ ...; ^ result }
{ ...; ^ } -- return void
```

Example

```c
method fetch(table, key, if_absent) {
    do(associations(table, &key, v) {
        if(key = key, {^ v});
    });
    eval(if_absent)
}
```

```c
method fetch(table, key) {
    fetch(table, key, {
        error("key " || print_string(key) || " not found")
    })
}
```

```c
fetch(zips, "Seattle", { 98195 })
```
Objects

- To define a new kind of ADT, use an object declaration
  
  ```
  object Point;
  ```
  
- No classes!

- To make a new "instance" of that ADT, use an object
  ```
  method new_point() {
    object isa Point
  }
  ```
  
- No special constructors!

Methods of objects

- To define a method "in" an object, write the method outside the object but **specialize** the method to the object by adding `@obj` after the first argument (which acts like the receiver argument)
  ```
  method area(@Point) {
    p.x * p.y
  }
  ```

Fields of objects

- To declare an instance variable, use a field declaration
  ```
  var field x(@Point) := 0;
  ```
  
- specialize the field to the object "containing" the field

- add `var` keyword to allow assignment, otherwise immutable

- fields can be given default initial values at declaration

- fields can be given initial values at object creation

- supports immutable, initialized fields!

- fields accessed by messages

  - Field declarations implicitly produce 1 or 2 accessor methods:
    - get accessor: given object, return field contents
    - set accessor (for `var` fields): given object & field's new contents, modify field

  - Manipulate field contents solely by invoking these methods

    ```
    var field x(@Point) := 0;
    method x(@Point) {
      ... fetch p.x's contents, initially 0 ...
    }
    method set_x(@Point, new_value) {
      ... update p.x to be new_value ...
    }
    ```

    -- increment p.x:
    ```
    set_x(p, x(p) + 1);
    ```

Syntactic sugar

- For syntactic convenience, any call can be written using dot notation:
  ```
  p.x x(p)
  p.x := p.x + 1 set_x(p, p.x + 1)
  p.shift(3, 4) shift(p, 3, 4)
  ```

- Infix & prefix operators (e.g. `+`) are really messages, too
  ```
  method +(p1Point, p2) {
    new_point(p1.x + p2.x, p1.y + p2.y)
  }
  ```

Inheritance

- Make new ADTs from old ones via `isa` inheritance clause
  ```
  object ColoredPoint isa Point;
  ```

- child/parent, a.k.a. subclass/superclass

- inherit all method & field declarations

- child has own field contents, unlike Self

- can add new methods & fields, specialized on child object

- can override methods & fields
Example

```
object ColoredPoint isa Point;
-- inherit all Point fields and methods
-- add some new ones:
field color:ColoredPoint;
method new_colored_point(x0, y0, c0) {
    object isa ColoredPoint {
        x := x0, y := y0, color := c0
    }
}
let p := new_colored_point(3,4,"Blue");
print(p.color); fi "Blue"
```

Overriding of methods

- Child can override inherited method by defining its own

```
object Point;
method draw(p@Point) { .. }
```

```
object ColoredPoint isa Point;
method draw(p@ColoredPoint) { .. }
```

```
let p := new_point(3,4);
p.draw;  -- invoke's Point's draw
```

```
let cp := new_colored_point(5,6,"Red");
cp.draw;  -- invokes ColoredPoint's draw
```

Resends

- Often, overriding method includes overridden method as a subpiece
- Can invoke overridden method from overriding method using `resend`
  - called `super` in some other languages

```
method draw(p@Point) {
    Display.plot_point(p.x, p.y);
}
method draw(p@ColoredPoint) {
    Display.set_color(p.color);
    resend;
}
```

Overriding of fields

- Since fields accessed through accessor methods, can override accessor methods with regular methods, & vice versa

```
object Origin isa Point;
method x(o@Origin) { 0 }
method y(o@Origin) { 0 }
```

Accessing fields

- Because fields accessed through messages, like methods, clients can’t tell how message implemented
  - can differ in different child objects
  - can change through program evolution & maintenance

```
let p : ...; -- Point or Origin object
print(p.x); -- how is x implemented?
```

Overloaded methods and dynamic dispatching

- Can overload methods two ways:
  - same name but different numbers of arguments
  - same name & number of arguments, but different specializer objects
- Specializer-based overloading resolved by using run-time class of receiver argument (a.k.a. dynamic dispatching, message sending)
  - unlike static overloading, which uses only the static type known at the call site
Multimethods

- Any argument, not just the receiver, can be specialized to an object
  
  ```
  method = (p1: Point, p2: Point) {
    p1.x = p2.x & { p1.y = p2.y } 
  }
  method = (cp1: ColoredPoint, cp2: ColoredPoint){
    cp1.x = cp2.x & { cp1.y = cp2.y } &
    { cp1.color = cp2.color } 
  }
  ```

- A message invokes the unique most-specific applicable method

Examples

```plaintext
method = (p1:Point, p2:Point) { ... }
method = (cp1:ColoredPoint, cp2:ColoredPoint) { ... }

let p1 := new_point(...);
let cp1 := new_colored_point(...);
print(p1 = p2); -- only Point-Point applies
print(cp1 = p2); -- ditto
print(cp1 = cp2); -- both apply, CP-CP wins
```

Method lookup rules

- Find all methods with the right name and number of arguments that apply
  
  - A method applies if the actual run-time objects are equal to or inherit from all the method's specializers, where present
  
  - Report "message not understood" if no applicable methods

- Pick the applicable method whose specializers are uniformly most specific
  
  - A specializer is more specific than another if it inherits from the other
  
  - A method overrides another if all of its specializers are at least as specific as the other's
  
  - Report "message ambiguous" if no single best method

Multimethod overriding

- One multimethod overrides another if
  
  - for all the other's specializers, the first method's corresponding specializers are equal to or inherit from the other's, and
  
  - either:
    
    - at least one of the first's specializers strictly inherits from the other's
    
    - one of the first's forms is specialized while the other's is not

```
method foo(p1:Point, p2:Point) { ... }
overridden by
method foo(p1:ColoredPoint, p2:Point) { ... }
```

```
method foo(p1:Point, p2:ColoredPoint) { ... }
overridden by
method foo(p1:ColoredPoint, p2:Point) { ... }
```

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

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) { ... }
```

Resolving ambiguities

- Can resolve ambiguities by defining an overriding method

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

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

```
method foo(p1:ColoredPoint, p2:ColoredPoint) { ... }
```
Directed resend

Overriding method can choose one or more ambiguously inherited methods using a directed resend.

```java
directed resend
method foo(p1 @ ColoredPoint, p2 @ Point) { ... }  
method foo(p1 @ Point, p2 @ ColoredPoint) { ... }  
method foo(p1 @ ColoredPoint, p2 @ ColoredPoint) {  
    -- invoke the ColoredPoint · Point one:  
    resend(p1, p2 @ Point);  
    -- invoke the Point · ColoredPoint one:  
    resend(p1 @ Point, p2);  
}
```

Multimethods vs. static overloading

- Multimethods support dynamic overloading: use dynamic class of arguments to resolve overloading.
- Static overloading is different: use static type of arguments known at call site to resolve overloading.
- Dynamic overloading is more powerful...

Example in Java

```java
class Point {  
    boolean equals(Point arg) {  
        return this.x = arg.x && this.y = arg.y;  
    }  
}  
class ColoredPoint extends Point {  
    boolean equals(ColoredPoint arg) {  
        return ... && this.color = arg.color;  
    }  
}  
```

```
Point p1 = ...; // might be a ColoredPoint  
Point p2 = ...; // might be a ColoredPoint  
... p1.equals(p2) ... // which method is invoked?
```

Second example in Java

```java
class Point {  
    boolean equals(Point arg) {  
        return this.x = arg.x && this.y = arg.y;  
    }  
}  
class ColoredPoint extends Point {  
    boolean equals(Point arg) {  
        return false;  
    }  
}  
```

```
Point p1 = ...; // might be a ColoredPoint  
Point p2 = ...; // might be a ColoredPoint  
... p1.equals(p2) ... // which method is invoked?
```

Third example in Java

```java
class Point {  
    boolean equals(Point arg) {  
        return this.x = arg.x && this.y = arg.y;  
    }  
}  
class ColoredPoint extends Point {  
    boolean equals(Point arg) {  
        if (arg instanceof ColoredPoint) {  
            ColoredPoint cpArg = (ColoredPoint) arg;  
            return ... && this.color = cpArg.color;  
        } else {  
            return false;  
        }  
    }  
}  
```

Example in MultiJava

- Allow arguments to have specializers

```java
class Point {  
    boolean equals(Point arg) {  
        return this.x = arg.x && this.y = arg.y;  
    }  
}  
class ColoredPoint extends Point {  
    boolean equals(Point @ ColoredPoint arg) {  
        return ... && this.color = arg.color;  
    }  
}  
```
Some uses for 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

Some more uses

- Multimethods useful for cooperative operations even over different types
- Examples:
  - `display` for various kinds of shapes on various kinds of output devices
  - standard default implementation for each kind of shape
  - overridden with specialized implementations for certain devices
  - `handleEvent` for various kinds of services for various kinds of events
  - operations taking flag constant objects, with different algorithms for different flags

Advantages of multimethods

- 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
- Avoid tedium & non-extensibility of `instanceof/cast`

Challenges of multimethods

- Objects don't contain their methods, so...
  - What's the programming model?
  - What's the encapsulation model?
- How to typecheck definitions and calls 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;
  - object Stream;
  - object InputStream isa Stream;
  - object OutputStream isa InputStream, OutputStream;

- 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
  - `object Rectangle isa Shape;
  - object Rhombus isa Shape;
  - object Square isa Rectangle, Rhombus;
  - method area(r:Rectangle) { ... }
  - method area(r:Rhombus) { ... }
  - object square := new_square(4);
  - area(r) ... E ambiguous!

- Can resolve ambiguities by adding overriding method, just as with MMs
  - `method area(s:Square) { resend(s:Rectangle) }

Multiple inheritance

- Can inherit from several parent objects:
  - `object Shape;
  - object Rectangle isa Shape;
  - object Rhombus isa Shape;
  - object Square isa Rectangle, Rhombus;
  - object Stream;
  - object InputStream isa Stream;
  - object OutputStream isa InputStream, OutputStream;

- MI can be natural in application domain
- MI can be useful for better factoring & reuse of code
- But MI introduces semantic complications....
Semantics of diamond-shaped inheritance?

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

let s := new_square(4);  
... is_shape(s) ... fi ambiguous?  
... is_rectangular(s) ... fi Rectangle's  
... area(s) ... fi ambiguous

Cecil semantics:  
inheritance as a partial ordering

object Shape;  
method is_shape(s @ Shape) { ... }  
object Rectangle isa Shape;  
object Rhombus isa Shape;  
object Square isa Rectangle, Rhombus;  

let s := new_square(4);  
... is_shape(s) ... fi Shape's  
... is_rectangular(s) ... fi Rectangle's  
... area(s) ... fi ambiguous

Semantics of inheritance of fields?

object Shape;  
field center(s @ Shape);  
object Rectangle isa Shape;  
object Rhombus isa Shape;  
object Square isa Rectangle, Rhombus;  

let s := new_square(4);  
... center(s) ... fi ambiguous

Other options

- Smalltalk, Java, C#: disallow MI  
  - sacrifices many practical examples  
- Self: like Cecil, but without partial order  
  - some "obvious" ambiguities not resolved  
- CLOS: linearize DAG into SI chain  
  - complex linearization rules, ambiguities always resolved  
- C++: two styles of MI  
  - non-virtual base classes (the default): replicate diamonds into trees  
  - virtual base classes: one shared copy  
  - very complex, bad default

Cecil semantics:  
fields are shared

In Cecil, fields are present once, independently of along how many paths they are inherited  
- field accessor methods are treated just like regular methods  
- field contents are stored once per inheriting object  
  ... center(s) ... fi s's contents of Shape's center field

Other options

- Self: slot (i.e., field contents) is shared  
  - leads to separating prototype & traits objects  
- C++: two styles of MI  
  - non-virtual base classes (the default): replicate instance variable  
  - virtual base classes: one shared copy (like Cecil)
Mixins

- MI enables new programming idioms, including mixins: highly factored abstract objects
- Typically, organize attributes along independent axes
- several possible implementations (mixins) for 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, mouse-click handler...
- Different mixin axes have common parent (e.g., Shape), leading to diamond-shaped inheritance

Java’s approach

- Java supports two flavors of classes: 
  - regular classes and interfaces
- Interfaces include no implementation, just “abstract methods”
  - no instance variables
  - no method bodies
- Allow multiple inheritance only of interfaces
  - a class can inherit from at most one regular class
  - an interface can inherit only from interfaces

Analysis of Java’s approach

- Benefits:
  - no method bodies in interfaces \(\Rightarrow\) no ambiguities between implementations
  - no instance variables in interfaces \(\Rightarrow\) no ambiguities in instance variable offset calculations
  - still support some multiple inheritance idioms
    - primarily for static type checking, not code reuse
- Costs:
  - no mixin-style programming
  - additional language complexity and library size

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 this doesn’t create errors

General strategy

- Declare (or infer) types and their subtyping relationships
- Declare (or infer) types of variables
  - Check that assignments/initializations/returns only store subtypes of variable’s type
- Declare signatures of operations
  - Check that messages with particular actual argument types find at least one matching signature
  - Check that methods & fields completely and unambiguously implement covering signatures

Points of variation

- What’s a type?
- What’s a subtype?
- What’s a signature?
One approach: explicit types and signatures

```plaintext
type Point;
signature x(Point):num;
signature y(Point):num;
signature set_x(Point, num):void;
signature set_y(Point, num):void;
signature shift(Point, dx: num, dy: num):num;
signature = (Point, Point):bool;
signature new_point(x:num, y:num):Point;

type ColoredPoint subtypes Point;
-- "inherits" signatures of supertype
signature color(ColoredPoint):Color;
signature set_color(ColoredPoint, Color):void;
signature new_colored_point(…):ColoredPoint;
```

Field signatures

```plaintext
n Syntactic sugar: a "field-like" pair of signatures can be specified with a single field signature declaration

```plaintext
signature x(Point):num;
signature set_x(Point, num):void;
field signature x(Point):num;
```

Using types and signatures

```plaintext
Legal:
let var cp:ColoredPoint :=
    new_colored_point(1, 2, Blue);
let var p:Point := new_point(3, 4);
p := cp;
cp.color := Red;
cp.shift(5, 6);
print(p = cp);

Illegal (static type errors):
 cp := p;
p.color := Green;
p.x := "hi there";
cp.shift(60);
print(p = 5);
```

Another option: "unify" types and classes/objects

```plaintext
Can merge types with classes/objects
a class/object declaration automatically creates a corresponding type declaration
an isa clause automatically creates a corresponding subtypes clause

```plaintext
object Point;
-- type Point;
object CartesianPoint isa Point;
-- type CartesianPoint subtypes Point;
```

"Unify" signatures and methods/fields

```plaintext
Signatures implied by method & field decls
add explicit argument and result types

```plaintext
var field x[p:Point.Point]:num := 0;
-- field signature x(Point):num;
var field y[p:Point.Point]:num := 0;
-- field signature y(Point):num;
method shift(p:Point.Point, dx:num, dy:num):void {...}
-- signature shift(Point, num, num):num;
method +(p1:Point.Point, p2:Point.Point):bool {...}
-- signature +(Point, Point):bool;
method new_point(x:num, y:num):Point {...}
-- signature new_point(num, num):Point;
```

Inheritance vs. subtyping

```plaintext
In theory, classes aren't types, and inheritance isn't subtyping:
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
```
Cecil's approach

- In Cecil, can program these separately:
  - type, subtypes, signature declarations for interfaces
  - representation, inherits, implementation declarations for implementation
  - subtypes declarations to conformance of implementations to interfaces

Example

```cpp
-- Point & ColoredPoint: types and signatures as before
representation PointImpl subtype Point;
var field impl' x(p@PointImpl:Point):num := 0;
var field impl' y(p@PointImpl:Point):num := 0;
impl' shift(p@PointImpl:Point,
  dx: num, dy: num):void {...}
impl' = (p1@PointImpl:Point,
  p2@PointImpl:Point):bool {...}
impl' new_point(x0: num, y0: num):Point {...}

representation ColoredPointImpl
  inherits PointImpl subtype ColoredPoint;
  ... more implementation declarations here ...
```

Syntactic sugar

- Common case: inheritance and subtyping are parallel
  - object defines representation & type
    - the representation subtypes the type
  - isa defines parallel inherits & subtypes
  - method defines implementation & signature
  - does parallel & and :
    - method = (p1@Point:Point, p2@Point:Point):...
    - method = (p1@Point, p2@Point):bool [...]

Benefits of separating inheritance and subtyping

- Clarity of thinking
- Sensible to implement interface w/o inheriting code
- Akin to Java's interfaces
- Sometimes sensible to reuse code w/o being a subtype
  - E.g. if ColoredPoint wants to inherit Point's code, but not allow ColoredPoints to mix with uncolored Points
- Sometimes the two relations are opposite
  - object deque subtypes stack;
  - object stack inherits deque;

Costs of separating inheritance and subtyping

- Verbosity in common case
  - ⇒ need syntactic sugar
- Complexity
- Subtyping w/o inheritance cannot provide default implementations
  - A weakness of Java's interfaces
- Difficult to typecheck safety of inheriting w/o subtyping

Overloaded/overriding signatures

- What if there are several signatures (implicit or explicit) with the same name and number of arguments?
  - signature = (Point, Point):bool;
  - signature = (ColoredPoint, ColoredPoint):bool;
  - signature = (num, num):bool;
  - signature = (string, string):bool;
  - ...
- What does this mean for clients?
- (When) is this legal?
Client view of signatures

- A message send is OK if there's at least one signature that says so
  - E.g. \( cp1 = cp2 \) is legal if there's some signature whose argument types are (supertypes of) ColoredPoint
- The client doesn't have to "choose" the right one, or do dispatching

Legality

- To make signatures legal, whatever promises they make to clients have to be guaranteed by method implementations
- If a client could pass certain types of arguments in a message, then
  - exactly one method has to be able to handle those arguments
  - the result type of the method has to be something that the client will expect
- Related to when one method can legally override another

Legality of method overriding

- 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
- Are relaxed conditions also safe?
  - can the result type be more precise (or more general) in overriding method?
  - can an argument type be more precise (or more general) in overriding method?

An example

- Which (if any) of the overrides are legal?

```java
method copy(p:Point):Point
method copy(p:ColoredPoint):ColoredPoint
method copy(p:Point3D):Object
let p:Point := ...; -- a Point, ColoredPt, or Point3D
let q:Point := p.copy;
... q.x ...
let cp:ColoredPoint := ...; -- a ColoredPoint
let cq:ColoredPoint := cp.copy;
... cq.color ...
```

Another example

- Which (if any) of the overrides are legal?

```java
method slide(p:Point, dx:num):void
method slide(p:ColoredPoint, dx:int):void
method slide(p:Point3D, dx:Object):void
let p:Point := ...; -- a Point, ColoredPt, or Point3D
slide(p, 3.4);
let cp:ColoredPoint := ...; -- a ColoredPoint
slide(cp, 5);
let p3d:Point3D := ...; -- a Point3D
slide(p3d, "hi");
```

Binary methods and typechecking

- Is this OK? What does it print?

```java
method = (p1:Point, p2:Point):bool {
  p1.x = p2.x & { p1.y = p2.y }
}
method = (p1:ColoredPoint, p2:ColoredPoint): {
  resend & { p1.color = p2.color }
}
let p:Point := new_point(1, 4);
let cp:Point := new_colored_point(1, 4, Blue);
print(p = p);
print(p = cp);
print(cp = p);
print(cp = cp);
Binary methods with multimethods

Is this OK? What does it print?

```c
method = (p1:Point, p2:Point):bool {
    p1.x = p2.x & { p1.y = p2.y }
}
method = (p1:ColoredPoint, p2:ColoredPoint):
    { p1.color = p2.color }

let p:Point := new_point(3,4);
let cp:Point := new_colored_point(3,4,Blue);
print(p = p);
print(p = cp);
print(cp = p);
print(cp = cp);```

Overriding fields

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

```c
class Point {
    var p:Point
    var color:Color
}

class ColoredPoint extends Point {
    var color:Color
}

field f(p:Point):A;
method f(p:ColoredPoint):A' {…}
var field g(p:Point):B;
method g(p:ColoredPoint):B' {…}
method set_g(p:ColoredPoint, v:B') {…}
```

What is the most flexible but still safe relationship between A and A’ and between B and B’?

Summary of overriding

Legal to override method in subtype if:

- result type same or a subtype (covariant)
- argument types same or supertypes (contravariant)
  - for undischarged arguments
  - dispatched arguments are replaced with subtypes

Contravariance is a pain in practice, but “It’s the Law” (for type safety, at least)

Checking signatures

In Cecil, allow arbitrary signatures and implementations

Need to ensure that each signature is completely and unambiguously implemented by one or more methods

Naive algorithm:

- for each combination of classes of arguments which is type-correct according to the signature
  - do method lookup
  - verify unique most-specific applicable method found

Efficiency? Modularity?

Abstract classes and methods

Most OO languages allow abstract classes, which can have abstract (unimplemented) methods

- Abstract methods OK as long as no instances of the abstract class can be created
- Cecil supports this idea through object role annotations
  - Used only during typechecking

Object roles

- abstract object: like an abstract class
  - cannot be manipulated directly
  - doesn’t have to have its signatures implemented
- template object: like a concrete class
  - cannot be manipulated directly
  - has to have its signatures implemented
- concrete object: like an instance
  - can be manipulated directly
  - has to have its signatures implemented
Example

abstract object List;
signature isEmpty(List):bool;

template object Cons isa List;
method isEmpty(a:Cons):bool { false }

concrete object Nil isa List;
method isEmpty(a:Nil):bool { true }

Parameterized types

- Simple approach:
  - add explicit type parameters on objects, methods
  - type parameters treated as regular (but unknown) types in their scope
  - instantiate when using a parameterized thing

Example:

```
template object Array[T] isa Collection[T];
method new_array[T](size:int):Array[T] {
   concrete object isa Array[T] {...} }
let a:Array[string] := new_array[string](10);
```

Parameterized methods

```
method fetch[T](a:Array[T], i:int):T { ... }
method store[T](a:Array[T], i:int, v:T):void {
   ... }
let a:Array[string] := new_array[string](10);
... store(a, 5, "hi");
... let s:string := fetch(a, 5);
```

Implicit type parameters

- Often, type parameter instantiations can be inferred from types of arguments to methods
- use `T` to mark a type parameter that's inferred in this way
- clients don't instantiate explicitly; system infers instantiation

```
method fetch(a:Array['T], i:int):T { ... }
method store(a:Array['T], i:int, v:T):void {
   ... }
let a:Array[string] := new_array[string](10);
... store(a, 5, "hi");
... let s:string := fetch(a, 5);
```

Universal vs. bounded parametric polymorphism

- Want to place constraints on legal instantiations of type variables, so that we can do interesting things with values of that type
  - ML has equality types
  - Wish ML had more flexible kinds of type for things that support `print`, `<`, etc.
- Example:
  - a `print` method on `Array[T]`, given that elements can be printed
  - how to express the constraint on `T` such that values of type `T` are known to be printable?

Approach 1: subtype bound

- Declare a type that has all the desired operations
  - type Printable;
  - signature print(Printable):void;
- Have some classes implement this type
  - template object string subtypes Printable;
  - method print(a:string):void {... }
- Add a `bound` to type variables requiring them to be subtypes of the given type
  - method print(a:Array[T <= Printable]):void{
  a.b(i:elem:T{ print(elem); }); }
- Can call this method on legal arguments
  - let a:Array[string] := ...
  ... print(a) ...
Bounds on parameterized objects

- Can place bounds on parameterized objects to require all instances to support operation(s)
  
  ```
  template object Array[T <= Printable]
  isa Collection[T]
  
  method print(a@Array[T]):void {...}
  
  Now can only create Arrays of things that are printable
  
  Supported by Cecil, Java 1.5, next C#
  ```

Approach 2: signature bound

- Express constraints directly as a required signature rather than indirectly as subtyping from something with the signature

  ```
  method print(a@Array[T]):void
  where signature print[T]:void
  {
    a.do(!elem:T) {[ print(elem); }];
  }
  
  Supported by Cecil, PolyJ
  ```

Approach 3: check after instantiation

- Could just write code, and check whether it works after instantiating with specific types

  ```
  ... [not legal Cecil]
  method print(a@Array):void {
    a.do(!elem:T) {[ print(elem); ]};
  
  let a:Array[Foo] := ...;
  print(a); -- macro-expand & check body of print
  
  Supported by C++, Modula-3
  ```

Approach 4: don't allow parameterized things

- Do dynamic type casts from any to desired/expected subtype when needed

  ```
  method print(a@Array):void {
    a.do(!elem:any) {
      let e:Printable := cast[Printable](elem);
      print(e);
    };
  }
  
  Supported by Java 1.4 and earlier, current C#
  ```

Comparison

- Subtype bounds more convenient if:
  - types already exist
  - many signatures required
  - want to encode semantics in types
- Signature bounds more convenient if:
  - few signatures
  - want to handle existing classes w/o adding new supertypes to them
- Unspecified bounds more convenient if:
  - hard to specify otherwise (e.g., superclass is a parameter)
  - don’t care about separate typechecking
- No parameterization more convenient if:
  - want simplest language
  - don’t care about fully static typechecking

Polymorphism over binary methods

- doesn't typecheck

  ```
  method sort(a:Array[T]):void {
    ... iterate over i,j ...
    let x:T := fetch(a,i);
    let y:T := fetch(a,j);
    if(gt(x,y),{ ... swap x and y ... });
  }
  
  Need to specify that send gt(x, y) is legal
  
  Signature constraints work fine:

  ```
  method sort(a:Array[T]):void
  where signature gt[T,T]:bool {
    ... }
  
  But what if prefer a subtype constraint?
First attempt

type Comparable;
signature gt(Comparable, Comparable):bool;

template object int subtypes Comparable;
method gt(n1:int, n2:int):bool { … }
template object string subtypes Comparable;
method gt(s1:string, s2:string):bool { … }

method sort(a:Array['T <= Comparable]):void {
... x:T ... y:T ... gt(x,y) ...
}

- sort now typechecks ✓
- gt isn't properly implemented ❌

Solution: F-bounded subtype constraint

type Comparable[T];
signature gt(Comparable[T], T):bool;

template object int subtypes Comparable[int];
method gt(n1:int, n2:int):bool { … }
template object string subtypes Comparable[string];
method gt(s1:string, s2:string):bool { … }

method sort(a:Array['T <= Comparable[T]]):void {
... x:T ... y:T ... gt(x,y) ...
}

- sort now typechecks ✓
- gt properly implemented ✓
- whaa?! ❌

In English...

- Comparable takes as a parameter the type of things that are being compared against
type Comparable[T];
signature gt(Comparable[T], T):bool;

- Implementations of Comparable specify the type of things that they can be compared against
object int subtypes Comparable[int];
object string subtypes Comparable[string];

- Sort takes an array of things that can be compared against themselves
method sort(a:Array['T <= Comparable[T]]):void { … }

Another example

method max(x:'T, y:'T):'T
where 'T <= Comparable['T]
{ if(gt(x,y), { x }, { y }) }

- max on strings returns a string
- max on ints returns an int
- a static type error to try to do max on a string and a number

Summary

- F-bounded polymorphism is required for many practical examples of OO polymorphism
  Supported in Cecil, Java 1.5, new C#
- Pretty tricky to learn how to define your own F-bounded classes and methods
- Signature-bounded polymorphism remains simple