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, ...

- 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
method add_x(x) { &&(y){ x + y } }
```

```c
let add_2 := add_x(2);
let add_3 := add_x(3);
```

```c
eval(add_2, 4) fi 6
eval(add_3, 4) fi 9
```

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

More examples

For iteration with arguments:

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

For exception handling:

```c
fetch(table, key, { if_absent })
```

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
like `^` in Self
like a return statement in C
```

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

Example

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

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 isa ... expression
  ```object isa Point
  method new_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) {```

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!
  ```var field x(@Point) := 0;
  var field y(@Point) := 0;
  method new_point(x0, y0) {```

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 ...
  }```

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,x(p)+1)
p.shift(3,4) shift(p, 3, 4)```
- Infix & prefix operators (e.g. +) are really messages, too
  ```method +(p1@Point, 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(cp@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"
p.shift(2,-2);  -- invoke inherited method
print(p.x);      fi 5

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

1. 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 } }

2. A message invokes the unique most-specific applicable method

Examples

   method = (p1@Point, p2@Point) { ... }
   method = (cp1@ColoredPoint, cp2@ColoredPoint) { ... }

   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

Method lookup rules

1. Find all methods with the right name and number of arguments that apply
   a. A method applies if the actual run-time objects are equal to
      or inherit from all the method’s specializers, where present
   b. Report "message not understood" if no applicable methods
2. Pick the applicable method whose specializers are uniformly most specific
   a. A specializer is more specific than another if it inherits from
      the other
   b. A method overrides another if all of its specializers are at
      least as specific as the other's
   c. Report "message ambiguous" if no single best method

Multimethod overriding

1. One multimethod overrides another if
   a. for all the other’s specializers, the first method’s corresponding specializers are equal to or inherit from the other’s, and
   b. at least one of the first’s specializers strictly inherits from the other’s, or one of the first’s formals is specialized while the other’s is not

   method foo(p1@Point, p2) { ... }
   overridden by
   method foo(p1@Point, p2@Point) { ... }

   method foo(p1@ColoredPoint, p2) { ... }
   overridden by
   method foo(p1@ColoredPoint, p2@ColoredPoint) { ... }

Ambiguous methods

1. 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

1. 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
method foo(p1:ColoredPoint, p2:Point) { ... }
method foo(p1:Point, p2:ColoredPoint) { ... }
method foo(p1:Point, 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; }
  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?
```

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;
method area(e@Rectangle) { ... }
object Rhombus isa Shape;
method area(e@Rhombus) { ... }
object Square isa Rectangle, Rhombus;
let s := new_square(4);
... area(s) ... is ambiguous!
```

- Can resolve ambiguities by adding overriding method, just as with MMs

```
method area(s@Square) { resend(s@Rectangle) }
```
Semantics of diamond-shaped inheritance?

object Shape;
method is_shape(s @ Shape) { ... }
method is_rectangular(s @ Shape) { ... }
object Rectangle isa Shape;
method is_rectangular(s @ 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 ambiguous
... area(s) ... fi ambiguous

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 their specializers
this partial ordering on methods defines the overriding relationship

... is_shape(s) ... fi Shape’s
... is_rectangular(s) ... fi Rectangle’s
... area(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

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 s’s contents of Shape’s center field

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 set_x(Point, num):void;
signature y(Point):num;
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;
```

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

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);
```

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

Signature = method/field/signature

```plaintext
Signatures extracted from method & field decls
  - add explicit argument and result types
  - use explicit signature decl for an abstract method

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 new_point(p:Point:Point):bool ...
  -- signature new_point(Point):bool;
method new_point(x:Point, y:Point):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

```
-- 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' n shift(p@PointImpl:Point, dx:num, dy:num) : void {...}
impl' n p@PointImpl:Point, p@PointImpl:Point : bool {...}
impl' n 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
  - it: does parallel & does:
    - method x(p@Point:Point, p@Point:Point)...:
    - method x(p@Point:Point, p@Point: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?
  - \[ \text{method } \text{copy}(p@Point):Point \]
  - \[ \text{method } \text{copy}(p@ColoredPoint):ColoredPoint \]
  - \[ \text{method } \text{copy}(p@Point3D):Object \]

Another example

- Which (if any) of the overrides are legal?
  - \[ \text{method } \text{slide}(p@Point, dx:min):void \]
  - \[ \text{method } \text{slide}(p@ColoredPoint, dx:int):void \]
  - \[ \text{method } \text{slide}(p@Point3D, dx:Object):void \]

Binary methods and typechecking

- Is this OK? What does it print?
  - \[ \text{method } \text{=(p@Point, p2:Point):bool} \{
    p1.x = p2.x & \{ p1.y = p2.y \} \}
  \]
Binary methods with multimethods

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(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?

```java
field f(p:Point):A;
method f(p:ColoredPoint):A' (...) 

d field g(p:Point):B;
method g(p:ColoredPoint):B' (...) 
method set_g(p:ColoredPoint, v:B'):void {...}
```

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