## Cecil and Diesel

Purely object-oriented languages

- all data structures through user-defined objects
  - **classless** object model, but inheritance is static
  - type safe, garbage collected, implicit pointers, ...
- all operations & constructors through user-defined functions & methods
  - methods can dispatch on 0, 1, or several arguments ⇒ **multimethods**
- all instance & class variables through user-defined fields
- all control structures through user-defined code manipulating first-class, lexically-nested, anonymous closure objects
- invoke functions, fields, constructors, & closures uniformly through messages

Polymorphic static type checking, with F-bounded and signature-bounded polymorphism

- can omit type declarations ⇒ dynamically typed

Diesel adds module system, explicit (generic) functions vs. overriding methods, simpler inheritance & subtyping model



## Functions and variables

Use `fun` to define functions

- return result of last expression
- can overload for different numbers of arguments
  - (overriding later)

Use `let` to define (local and global) variables

- add `var` keyword to allow assignment, otherwise immutable
  - formals always immutable
- must initialize at declaration
- can infer type of immutable local var from initializer

```
let var count:int := 0;
fun foo(a:int, b:int, c:int):int {
  count := count + 1;
  let var d:int := a + b;
  let e := frob(d, c);   // infer type of e
  d := d + e;
  d + 5 }
fun frob(x:int, y:int):int { x - frob(y) + 1 }
fun frob(x:int):int { - x / 5 }
```



## Closures: first-class functions

Code bracketed in braces is a 0-argument function value (called a **closure** in Cecil-speak)

```
let closure := { factorial(10) + 5 };
```

Evaluation of closure body delayed until invoked by `eval`:

```
eval(closure)   → 3628805
```

To allow arguments to closure, add `&(`*`formals`*`)` prefix; invoke passing extra arguments to `eval`:

```
let closure2 := &(n:int){ factorial(n) + 5 };
eval(closure2, 10)  → 3628805
```

Closure type: `&(`*`formalTypes`*`):`*`resultType`*

- *`resultType`* can be inferred from body, as above

Like Self's blocks:

- anonymous
- lexically scoped
- first-class
- inherit additional behavior from library class
- invoked via a message send



## Warning: implementation limitation of closures

In current Vortex implementation of Cecil & Diesel, regular closures cannot safely outlive their lexically enclosing scope

- prevents currying, `compose`, closures in data structures, ...
- not checked: can crash compiled programs if violated!

To be able to return closures, use `&&` rather than `&`:

```
fun add_x(x:int):&(int):int {
  &&(y:int){ x + y } }

let add_2 := add_x(2);
let add_5 := add_x(5);

eval(add_2, 4)   → 6
eval(add_5, 4)   → 9
```

## Using closures in control structures

All traditional (and many non-traditional) control structures implemented as regular Cecil functions, with closures passed by callers for delayed evaluation

- like Smalltalk, Self

For simple lazy or repeated evaluation:

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

For iteration with arguments:

```
for(start, stop, &(i:int){ body })
do(array, &(elem:elemType){ body })
do_associations(table,
    &(key:keyType,value:valueType){ body })
```

For exception handling:

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

For continuation-passing style code:

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



## An example

```
-- this is a factorial method
fun factorial(n:int):int {
   if(n = 0,
      { 1 },
      { n * factorial(n - 1)}) }

-- call factorial here:
factorial(7)
```



## Non-local returns

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

- as in Smalltalk, Self
- like a `return` statement in C
- like a limited kind of continuation in Scheme

```
{ ...; ^ result }
{ ...; ^ }
```

Example (omitting types):

```
fun fetch(table, key, if_absent) {
  do_associations(table, &(k, v){
    if(k = key, { ^ v });
  });
  eval(if_absent) }
fun fetch(table, key) {
  fetch(table, key, {
    error("key " || print_string(key) ||
          " not found") }) }

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



## Classes

To define a new kind of ADT, can use `class` declaration

- can have 0, 1, or many superclasses
- no instance variables, constructors, or methods declared as part of the class!

```
class Point;
class ColoredPoint isa Point;
```

## Objects

Can make new objects using either
object declarations or object expressions

Object declarations look like class declarations:

```
object Origin isa Point;
```

- allow one-of-a-kind objects easily
- can inherit from a declared object, too

Object expressions look like `new ClassName`

- creates an instance of *`ClassName`*
- each instance is an object that inherits from *`ClassName`*
  - like **`object`** <*anon*> **`isa`** *`ClassName`*

This is a classless object model!

- no instantiation, just inheritance
- classes are just objects that can't be manipulated at run-time (akin to traits in Self)
- unlike Self:
  superclass must be declared, inheritance is immutable



## Fields

Use a `field` declaration to declare an instance variable

- one formal: type is class/object the field is part of
  - each argument value stores its own value of the field
  - any formal name can be omitted if unused in body
- a field can be given default initial value at declaration
- a field can be given initial value at object creation
- `var` keyword to allow assignment, otherwise immutable

```
class Point;
  var field x(:Point):int { 0 }
  var field y(:Point):int { 0 }

class ColoredPoint isa Point;
  -- each ColoredPoint instance has x & y too
  field color(:ColoredPoint):Color { Black }

object BlueOrigin isa ColoredPoint
  { color := Blue };

let p1 := new Point { x := 3, y := 4 };
let cp2 := new ColoredPoint { x := 5 };
```



## Constructors

Constructors are just regular functions that
return initialized objects

```
class Point;
  var field x(:Point):int { 0 }
  var field y(:Point):int { 0 }

  fun new_point(x:int, y:int):Point {
    new Point { x := x, y := y} }
```

Advantages:

- can give constructor functions appropriate names
  - no need to rely solely on static overloading to resolve different constructors
- constructor functions don't have to allocate a new object
  - they can cache and return a previously allocated object
  - they can return an instance of a subclass



## "Methods of a class"

As with constructors, methods of a class are supported using regular functions that take an instance of the class as an (explicit) argument

- no implicit `this`/`self` argument
- "inherited" simply by allowing subtypes as arguments

```
class Point;
  var field x(:Point):int { 0 }
  var field y(:Point):int { 0 }
  fun new_point(x:int, y:int):Point {
    new Point { x := x, y := y} }
  fun area2origin(p:Point):int { p.x * p.y }
  fun shift(p:Point, dx:int, dy:int):void {
    p.x := p.x + dx;
    p.y := p.y + dy; }
  fun draw(p:Point):void {
    Display.plot_point(p.x, p.y); }
```

Advantages:

- can easily add new "methods" to existing classes just by writing functions
  - likewise for adding new instance variables or constructors

## "Messages"

Messages are just function calls

Prefix and infix operations are just function calls, too

- any sequence of punctuation symbols can be a fun. name
- can specify precedence & associativity of operators

```
fun -(x:int):int { 0 - x }
fun +(x:int,y:int):int { ... int addition ... }
fun -(x:int,y:int):int { ... int subtraction ... }
precedence *,/,% left_associative above +,-;
```

For syntactic convenience,
any call can be written using dot notation:

```
p.area2origin   ⇔  area2origin(p)
p.x := p.x + 1  ⇔  set_x(p, x(p) + 1)
p.shift(3,4)    ⇔  shift(p, 3, 4)
```



## Field accessor functions

Field declarations implicitly produce 1 or 2 *accessor functions*:

- get accessor:
  given object, return field's value for object
- set accessor (for `var` fields):
  given object & new value, modify field's value for object
  - accessor function's name is `set_`*`fieldName`*

Manipulate field contents solely by invoking these functions

```
var field x(p:Point):int { 0 }
  ⇒
    fun x(p:Point):int {
      ... fetch p.x's contents, initially 0 ...}
    fun set_x(p:Point, new_value:int):void {
      ... update p.x to be new_value ... }

p.x := p.x + 1;  -- same as set_x(p, x(p) + 1);
```



## Overriding of methods

If want to override a function's implementation
when an argument is a subclass, then use a `method` declaration

- **specialize** the method's formal to the subclass,
  using `@`*`subclass`* in place of `:`*`superclass`*
  - method only applies to a call if run-time argument object is the same as or inherits from *`subclass`*
- adds a new "case" to the function (a "generic function")
  - `fun` declaration provides initial default unspecialized case

```
class Point;
  fun draw(p:Point):void {
    Display.plot_point(p.x, p.y); }

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

Function call does dynamic dispatch:
invokes unique *most-specific* case in callee function that is *applicable* to the run-time classes of the arguments



## 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)

Without `resend`:

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

With `resend`:

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

## Overriding of fields

Since fields accessed through accessor functions, can override accessor functions with regular methods

- field state still there ⇒ can be accessed via a resend

Conversely, can declare `field methods` that override existing functions with field accessor methods

```
class PolarPoint isa Point;
  var field rho(:PolarPoint):int { 0 }
  var field theta(:PolarPoint):int { 0 }
  method x(p@PolarPoint):int {
    p.rho * cos(p.theta) }
  method set_x(p@PolarPoint, x:int):void {
    ... set rho and theta from new x and old y ... }
  ... also override y and set_y functions ...
```

Because fields accessed through messages, like methods, clients can't tell how message implemented

- can change in subclasses
- can change in future versions of the program



## Abstract classes and methods

Can declare abstract classes

- disallows instantiation via new

To define "abstract methods", define functions without bodies

- concrete subclasses must override with methods

```
abstract class Shape;
  fun area(s:Shape):int;
  fun center(s:Shape):Point;

class Rectangle isa Shape;
  method area(r@Rectangle):int { ... }
  method center(r@Rectangle):Point { ... }

class Circle isa Shape;
  method area(c@Circle):int { ... }
  field method center(@Circle):Point;
```



## Multiple dispatching

Allow more than one argument of a method to be specialized ⇒ a **multimethod**

```
fun =(p1:Point, p2:Point):bool {
  p1.x = p2.x & { p1.y = p2.y } }

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

Dynamic dispatching rules:
invoke unique *most-specific* case in callee function that is *applicable* to the run-time classes of the arguments

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

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



## (Multi)method overriding

One (method/function) case overrides another if:

- for each argument position, the specializer/type of the first case is the same as or inherits from the specializer/type of the second case
- and for at least one argument position, the specializer/type of the first case strictly inherits from (is not the same as) the specializer/type of the second case

```
fun =(p1:Point, p2:Point)
```
overridden by
```
method =(p1@ColoredPoint, p2@ColoredPoint)

method foo(p1@Point, p2@Point)
```
overridden by
```
method foo(p1@Point, p2@ColoredPoint)

method bar(p1@ColoredPoint, p2:Point)
```
overridden by
```
method bar(p1@ColoredPoint, p2@ColoredPoint)
```

## Ambiguous methods

Two methods may be mutually ambiguous:
neither overrides the other

```
method baz(p1@ColoredPoint, p2@Point) { ... }
```
is ambiguous with
```
method baz(p1@Point, p2@ColoredPoint) { ... }
```

```
method qux(p1@Point, p2@Point) { ... }
```
is ambiguous with
```
method qux(p1@Point, p2@Point) { ... }
```

Dynamic dispatching rules:
invoke *unique* most-specific case in callee function that is applicable to the run-time classes of the arguments

Possible function call errors:
- no applicable cases: message-not-understood error
- no unique most-specific case: message-ambiguous error

(Static typechecking rules out the possibility of these errors)



## Resolving ambiguities

Can resolve ambiguities by defining an overriding method
- method can do its own thing, or
  it can use a **directed resend** to invoke one or more of the existing methods

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

method baz(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 on arguments

Multimethods support *dynamic overloading* of methods
based on the *dynamic* class of the arguments

*Static overloading* of methods is
based on the *static* class of the arguments

They're different...



## An example

In Diesel:
```
fun =(p1:Point, p2:Point):bool {
  p1.x = p2.x & { p1.y = p2.y } }
method =(p1@ColorPoint, p2@ColorPoint) {
  resend & { p1.color = p2.color } }
```

In Java:
```
class Point {
  ...
  boolean equals(Point arg) {
    return this.x = arg.x && this.y = arg.y; }
}
class ColorPoint extends Point {
  ...
  boolean equals(ColorPoint arg) {
    return super.equals(arg) &&
           this.color = arg.color; }
}
```

A client:
```
Point p1 = ...;   // might be Point or ColorPoint
Point p1 = ...;   // might be Point or ColorPoint
... p1.equals(p2) ...   // what might happen?
```

## The example, revised

In Java:

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

A client:

```
Point p1 = ...;   // might be Point or ColorPoint
Point p1 = ...;   // might be Point or ColorPoint
... p1.equals(p2) ...   // what might happen?
```



## Another version

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

Uses “typecase” idiom for argument dispatching

- what if add new subclasses of `Point`?



## A more extensible alternative

Can simulate multimethods with **double-dispatching**

- dispatch on receiver argument to find a class-specific method
- those methods send a message which encodes the class of the receiver to their argument
- integral to the usual `Visitor` design pattern

```
class Point {
  ...
  boolean equal(Point p2) {
    return p2.equalToPoint(this); }
  boolean equalToPoint(Point p0) {
    return p0.x = this.x && p0.y = this.y; }
  boolean equalToColorPoint(ColorPoint p0) {
    return equalToPoint(p0); }
}
class ColorPoint extends Point {
  ...
  boolean equal(Point p2) {
    return p2.equalToColorPoint(this); }
  boolean equalToColorPoint(ColorPoint p0) {
    return equalToPoint(p0) &&
           p0.color = this.color; }
}
```



## Aside: MultiJava

MultiJava: an extension of Java with

- optional dispatching on arguments
- “open classes”: add methods to existing classes

(The two things that Diesel multimethods support that Java methods don’t)

Example, in MultiJava:

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

Point p1 = ...;   // might be Point or ColorPoint
Point p1 = ...;   // might be Point or ColorPoint
... p1.equals(p2) ...   // does the “right” thing
```

## 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:

- `draw` 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 flags (captured by declared named objects), with different algorithms for different flags



## 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)
- by being written outside of their "receiver" class, naturally allow existing classes to be extended with new behavior
- mechanize double-dispatching, make it extensible w/o modifying existing code

Disadvantages:

- where to put code becomes less clear
- typechecking challenges, particularly if doing modular typechecking without knowing all the code



## Multiple inheritance

Can inherit from several parent objects:

```
abstract class Shape;
class Rectangle isa Shape;
class Rhombus isa Shape;
class Square isa Rectangle, Rhombus;

abstract class Stream;
abstract class InputStream isa Stream;
abstract class OutputStream isa Stream;
abstract class IOStream 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, e.g. if two superclasses define methods, neither of which overrides the other

```
abstract class Shape;
  fun area(s:Shape):int;
class Rectangle isa Shape;
  method area(r@Rectangle):int { ... }
class Rhombus isa Shape;
  method area(r@Rhombus):int { ... }
class 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):int {
  resend(s@Rectangle) }
```

## Diamond-shaped inheritance

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

- diamond-shaped hierarchies very common (if allowed)

```
abstract class Shape;
  fun area(s:Shape):int;
  field center(:Shape):Point;
  fun is_rectangular(:Shape):bool { false }
class Rectangle isa Shape;
  method is_rectangular(@Rectangle):bool{true}
  method area(r@Rectangle):int { ... }
class Rhombus isa Shape;
  method area(r@Rhombus):int { ... }
class Square isa Rectangle, Rhombus;

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

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



## Diesel semantics: inheritance as a partial ordering

Inheritance graph defines a partial ordering over classes
⇒ "subclasses override superclasses"

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

Rules for field accessors just like regular methods

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

Some alternatives:

- Smalltalk, Java, C#: no multiple (code) inheritance
- Self: just disambiguate `center`, not `is_rectangular`
- CLOS: totally order all superclasses (**linearization**)
- C++: two kinds of inheritance, virtual and non-virtual



## 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, no matter how many paths inherit them
  - used in Diesel, CLOS, C++ virtual inheritance
- top of diamond is duplicated
  - shared parents' fields duplicated along each path
  - used in C++ non-virtual inheritance



## Java & C#'s approach

Java & C# support two flavors of classes:
regular classes and interfaces

Interfaces include no implementation, just "abstract methods"

- no instance variables
- no method bodies

Allow multiple inheritance of interfaces

- a class can inherit from at most one regular class
- an interface can inherit only from interfaces

Benefits:

- no method bodies in interfaces ⇒ no ambiguities between implementations
- no instance variables in interfaces ⇒ no ambiguities in instance variable offset calculations
- still support some multiple "inheritance" idioms

Costs:

- loss of many MI idioms
- additional language complexity and library size

## Encapsulation

How to hide internal implementation details?

Traditional solution: each class encapsulates its members

- `public`: member can be accessed by clients
- `private`: member only visible in the body of this class
- `protected`: member only visible in this class *and subclasses*

But Diesel doesn't put members inside of classes, so how can it encapsulate internals of an ADT?



## Modules

Can wrap declarations in a `module`

- annotate declarations `public`, `private`, or `protected`
  - methods must have same annotation as their function
- `import` other modules, see only public decls
- `extend` other modules, also see their protected decls

```
module PointMod {
  public class Point;
  public get protected put
    var field x(:Point):int { 0 }
  public get protected put
    var field y(:Point):int { 0 }
  public fun new_point(...):Point { ... }
  ...
}
module ColorPointMod {
  public extend PointMod;
  public import ColorMod;
  public class ColorPoint isa Point;
  public field color(:ColorPoint):Color { ... }
  ...
}
```



## Multiple namespaces

Each module defines a separate namespace

- can have different functions, classes, etc. with the same name declared in different modules

```
module IntMod {
  fun +(x:int, y:int):int { ... }
}
module PointMod {
  fun +(p1:Point, p2:Point):Point { ... }
}
```

If reference a name, and multiple decls are in scope, which is meant?

- nested scopes take precedence over enclosing scopes
- can qualify names to resolve ambiguity: *`Module`*`$`*`Name`*
  - e.g. `PointMod$+`
- for function calls, use static argument types to disambiguate
  - Diesel's version of static overloading, but only across modules
  - still have dynamic multiple dispatching within a function

Can nest modules for nested namespaces & finer encapsulation boundaries



## Typechecking OO Languages

In OO language, want static checking to ensure the absence of:

- message-not-understood errors
- message-ambiguous errors

Simultaneously, want to allow subclasses to be used in place of superclasses

General strategy:

- define what the **types** are, what the **subtyping** relation is
- declare/infer types of variables, functions
  - check that assignments/initializations only store subtypes of variable's type
  - check that function calls only pass subtypes of function's argument types
  - check that function bodies only return subtypes of function's result type
- Check that overriding method cases have argument and result types that are *compatible* with overridden methods
- Check that method cases *completely* and *unambiguously* implement function's type

## What are the types?

Option 1: each class defines its own distinct type
- simple
- what most practical OO languages, including Diesel, do

Option 2: types are distinct from classes
- cleaner theoretically
- what most formal OO languages, Cecil do

In addition, there may be
- built-in types, e.g. `int`, `char`, `void`, `any`, `none`
- built-in type constructors,
  e.g. lists, tuples, records, functions



## Types vs. classes

A type is an *interface* to an object
- specifies what can be done to an object, not (necessarily) how it is implemented
  - e.g. a record of (function) types
- like an interface in Java, C#

```
type Point {
  fun x():int;
  fun y():int;
  fun area2origin():int;
  fun equals(Point):bool;
  ...
}
```

A class is a particular *implementation* of an object
- provides instance variables, method code, etc.

A class **conforms** to a type iff
all instances of the class support the interface of the type



## What is the subtyping relation?

(Write $\tau_1 \leq \tau_2$ for "$\tau_1$ is a subtype of $\tau_2$")

Main constraint:
if $\tau_1 \leq \tau_2$, then all values satisfying $\tau_1$ must also satisfy $\tau_2$
- subtyping is reflexive and transitive

Option 1: each subclass is a subtype
- **by-name** or **nominal subtyping**
- simple
- what most practical OO languages, including Diesel, do

Option 2: subtyping is distinct from subclassing
- **structural subtyping**
- cleaner theoretically
- what most formal OO languages, Cecil do

In addition, there may be
- built-in subtyping, e.g. `int` $\leq$ `real`, $\tau \leq$ `any`, `none` $\leq \tau$
- built-in subtyping for different instances of
  built-in type constructors, via structural subtyping



## Structural subtyping

Subtyping defined implicitly by properties of the types,
not explicitly by user declaration
- structural subtyping defines the maximal safe subtyping relation
- (safe) by-name subtyping is a subset of structural subtyping

## Record subtyping

Structural subtyping between immutable record types:
when is it safe for one to be a subtype of the other?
$\{I_1:\tau_1,...,I_n:\tau_n\} \leq \{I_1':\tau_1',...,I_m':\tau_m'\}$

Reasoning:

What are the operations on a record value?

- just field lookup (a.k.a. projection)

When will the values of one immutable record type support all the operations allowed by another immutable record type?

- a value of type $\{I_1':\tau_1',...,I_m':\tau_m'\}$ allows any of the $I_1'$ fields to be looked up
  $\Rightarrow$ subtype must have at least those fields
  - but can have more: **width subtyping**
- looking up the $I_i'$ field of a value of type $\{I_1':\tau_1',...,I_m':\tau_m'\}$ yields a value of type $\tau_i'$
  $\Rightarrow$ subtype's $I_i'$ field must yield a value of this type
  - but can yield a subtype: **depth subtyping**

Immutable tuple types also admit depth subtyping

- technically, could admit width subtyping, too



## Formalization

Can formalize structural subtyping rules using inference rules

[reflexive] $\dfrac{}{\tau \leq \tau}$

[transitive] $\dfrac{\tau_1 \leq \tau_2 \qquad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$

[record] $\dfrac{\tau_1 \leq \tau_1' \qquad ... \qquad \tau_m \leq \tau_m' \qquad m \leq n}{\{I_1:\tau_1, ..., I_n:\tau_n\} \leq \{I_1:\tau_1', ..., I_m:\tau_m'\}}$

[tuple] $\dfrac{\tau_1 \leq \tau_1' \qquad ... \qquad \tau_n \leq \tau_n'}{(\tau_1 * ... * \tau_n) \leq (\tau_1' * ... * \tau_n')}$



## Function subtyping

Structural subtyping between function types:
when is it safe for one to be a subtype of the other?
$\tau_a \rightarrow \tau_r \leq \tau_a' \rightarrow \tau_r'$

Reasoning:

What are the operations on a function value?

- just calling

When will the values of one function type support all the operations allowed by another function type?

- a value of type $\tau_a' \rightarrow \tau_r'$ can be called on an arg of type $\tau_a'$
  $\Rightarrow$ subtype must allow that too
  - but can allow a supertype: $\tau_a \geq \tau_a'$
- calling a value of type $\tau_a' \rightarrow \tau_r'$ yields a value of type $\tau_r'$
  $\Rightarrow$ subtype must yield a value of that type too
  - but can yield a subtype: $\tau_r \leq \tau_r'$



## Formalization

[function] $\dfrac{\tau_a' \leq \tau_a \qquad \tau_r \leq \tau_r'}{(\tau_a \rightarrow \tau_r) \leq (\tau_a' \rightarrow \tau_r')}$

Relation between return types varies in the *same* direction as relation between enclosing function types:
subtyping of functions is **covariant** in result type

Relation between argument types varies in the *opposite* direction as relation between enclosing function types:
subtyping of functions is **contravariant** in argument type

Contravariance is a curse!

- prevents many desired subtypings, as we'll see

## Ref subtyping

Structural subtyping between mutable reference types:
when is it safe for one to be a subtype of the other?
$\tau$ `ref` $\leq$ $\tau$' `ref`

Reasoning:

What are the operations on a ref value?

- `deref`: $\tau$ `ref` $\rightarrow \tau$
- `update`: $\tau$ `ref` * $\tau \rightarrow$ `unit`

When will the values of one `ref` type support all the operations allowed by another `ref` type?

- a value of type $\tau$' `ref` can be dereferenced, yielding a value of type $\tau$' $\Rightarrow$ subtype must too
  - but can yield a subtype: $\tau \leq \tau$'
- a value of type $\tau$' `ref` can be updated with a value of type $\tau$' $\Rightarrow$ subtype must be able to be too
  - but `update` can be called on a more general value, too: $\tau \geq \tau$'

These two opposing variance constraints require $\tau = \tau$'
$\Rightarrow$ `ref` is **invariant** in its argument type

- $\tau$ appears covariantly in `deref`, contravariantly in `update`



## Object subtyping

In traditional OO languages where classes contain their instance variables and methods, can use previous rules to decide when it's safe for one object type to be a subtype of another

View an object type as a record type

- each method is a function type
- each mutable instance variable is a ref type
- each immutable instance variable is a regular type
- (plus more for recursive types, etc.)

Then one object type is a subtype of another object type iff the first's record type is a subtype of the other's

Implications: in a subtype:

- type of an immutable instance variable can be changed to a subtype
- type of a mutable instance variable cannot be changed
- result type of a method can be changed to a subtype, and argument types of a method can be changed to a supertype
  - same constraint holds for overriding method in a subclass!



## Subtyping in Diesel

Fields, functions, and methods are not in classes, so object types aren't simple record types

Instead:

- assume each class is a subtype of its superclasses
- constrain method overriding to validate this assumption
  - implies rules for mutable & immutable fields via their accessors

A method can safely override a function if,
for arguments that are type-correct for the function,
if the method is applicable to the arguments,
the method can always be called safely in place of the function

- this is just function subtyping, restricted to the case that the method is applicable

Rules:

- an *unspecialized* method argument must be at least as general as the function's argument type (contravariant)
  - a *specialized* method argument is covariant!
- the method result must be at least as specific as the function's result type (covariant)



## Examples

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

let p:Point := ...; -- could be any Point subclass
let q:Point := p.copy;
... q.x ...

fun move(p:Point, n:num):void {
  ... }
method move(p@ColorPoint, i:int):void {
  ... }
method move(p@Point3D, j:any):void {
  ... }

let p:Point := ...; -- could be any Point subclass
move(p, 3.4);
```

## Signatures

Overriding method can have a more precise result type

Would like to let clients that know they have more specific arguments also know they have more specific results

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

let p:Point := ...;
let p2 := p.copy;   -- p2:Point

let cp:ColorPoint := ...;
let cp2 := cp.copy; -- would like cp2:ColorPoint
```

Add `signature` to method decl to enhance the function's type:

```
method signature copy(p@ColorPoint):ColorPoint
  { ... }
-- now copy has type
--  (Point):Point & (ColorPoint):ColorPoint;
```

Can write `signature` alone, e.g. for "abstract overrides"

```
signature copy(:Point3D):Point3D;
```



## Binary methods and typechecking

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

```
class Point;
  fun =(p1:Point, p2:Point):bool {
    p1.x = p2.x & { p1.y = p2.y } }

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

A client:

```
let p1:Point := new_point(3,4);
let p2:Point := new_colored_point(3,4,"Blue");
p1 = p1    -- what happens?
p1 = p2    -- what happens?
p2 = p2    -- what happens?
p2 = p1    -- what happens?
```



## Binary methods with multimethods

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

```
class Point;
  fun =(p1:Point, p2:Point):bool {
    p1.x = p2.x & { p1.y = p2.y } }

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

A client:

```
let p1:Point := new_point(3,4);
let p2:Point := new_colored_point(3,4,"Blue");
p1 = p1    -- what happens?
p1 = p2    -- what happens?
p2 = p2    -- what happens?
p2 = p1    -- what happens?
```

(What semantics for mixed colored & plain points is desired?)



## Checking method implementations

Last part of typechecking:

"check that method cases *completely* and *unambiguously* implement function's type"

Straightforward w/ single dispatching & monolithic classes:

- check at each concrete class that all abstract functions overridden with concrete methods
  - **completeness**
- check at each class that there are no method ambiguities from multiple inheritance
  - **unambiguity**

## 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 `fun`, `signature`, `method`, `field`, `class` decls):

- foreach function 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

Example:

Fun/signature & method declarations:

```
fun =(:Point,:Point):bool
method =(p1@Point, p2@Point):bool
method =(p1@ColorPoint, p2@ColorPoint):bool
```

Concrete classes subtyping `Point`:
`Point`, `ColoredPoint`, `Point3D`



## Some questions

How to make the check efficient?

- focus only on interesting combinations
- check completeness and unambiguity separately

How to make the check modular?

- put declarations into modules
- limit how one module can extend functions and classes of another module
  - still more flexible than both OO and functional styles
- key research focus in Dubious, MultiJava, EML languages



## Parameterized types

Want parameterized types, a.k.a. parametric polymorphism

An approach:

- add explicit type parameters on classes, functions, etc.
  - type variables treated as regular (but unknown) types in their scope
- instantiate type parameters with real types to use a parameterized thing

Example:

```
class Array[T] isa Collection[T];
  fun fetch[T](a:Array[T], i:int):T { ... }
  fun store[T](a:Array[T], i:int, v:T):void {.}
  fun new_array[T](size:int):Array[T] {
    new Array[T] { ... } }
  fun new_array[T](size:int, default:T)
    :Array[T] { new Array[T] { ... } }

let a:Array[string] :=
  new_array[string](10, "");
store[string](a, 5, "hi");
let s:string := fetch[string](a, 6);
```



## Implicit type parameters

Often, type parameter instantiations of called functions can be *inferred* from types of call arguments

- use `'T` to mark a function type parameter *T* that's inferred in this way
- clients don't instantiate explicitly; system infers instantiation

```
class Array[T] isa Collection[T];
  fun fetch(a:Array['T], i:int):T { ... }
  fun store(a:Array['T], i:int, v:T):void {...}
  fun new_array[T](size:int):Array[T] {
    new Array[T] { ... } }
  fun new_array(size:int, default:'T):Array[T]
    { new Array[T] { ... } }

let a:Array[string] := new_array(10, "");
store(a, 5, "hi");
let s:string := fetch(a, 6);
```

Inference of function parameters particularly important if functions are outside of their parameterized classes

## Universal vs. bounded parametric polymorphism

Just as with ML & Haskell, we want to place constraints on legal instantiations of type variables, so that we can do interesting things with values of that type

Example:
a `print_elems` function on `Array[T]`, given that elements can be printed

```
fun print_elems(a:Array[‘T]) {
  a.do(&(elem:T){
    -- illegal: print not necessarily defined on T!
    print(elem);
  });
}
```

How to express the constraint on the argument of `print_elems` such that values of type `T` are known to support `print`?



## Approach 1: subtype bound

Declare a type that has all the desired operations

```
abstract class Printable;
  fun print(:Printable):void;
```

Add a bound to type variables requiring them to be subtypes of the given type

```
fun print_elems(a:Array[‘T <= Printable]) {
  a.do(&(elem:T){ print(elem); }); }
```

Alternatively, can bound parameters of parameterized classes to require all instances to support operation(s)

```
class Array[T <= Printable] isa Collection[T];
fun print_elems(a:Array[‘T]):void {
  a.do(&(elem:T){ print(elem); }); }
```

Can declare *conditional subtyping*

```
extend class Array[‘T <= Printable]
    isa Printable;
  method print(a@Array[‘T <= Printable]):void {
    a.do(&(elem:T){ print(elem); }); }
```

(All features supported by Diesel, some by Java 1.5, C# 2.0)



## Approach 2: signature bound

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

```
fun print_elems(a:Array[‘T]):void
    where signature print(:T):void {
  a.do(&(elem:T){ print(elem); }); }
```

(Supported by Diesel, PolyJ)



## Approach 3: check after instantiation

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

- most expressive statically checked approach
- loses modular checking

```
-- [not legal Diesel]
fun print_elems(a:Array[‘T]):void {
  a.do(&(elem:T){ print(elem); }); }
...
let a:Array[Foo] := ...;
print(a); -- macro-expand & check body of print
```

(Used by C++, Modula-3)

## Approach 4: forgo parametric polymorphism

Use subtype polymorphism (plus dynamically checked downcasts) in place of parameteric polymorphism

- expressive, simple
- loses static guarantees

```
fun print_elems(a:Array['T]):void {
  a.do(&(elem:any){
    let e:Printable := cast[Printable](elem);
    print(e);
  });
}
```

(Used by Java 1.4 and earlier, C# 1.x)



## 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 required
- want to work for existing classes w/o adding new supertypes to them

Unspecified bounds more convenient if:

- hard to specify otherwise (e.g. supertype is a parameter)
- don't care about separate typechecking

No parameterization more convenient if:

- want simplest language
- don't care about fully static typechecking



## Parametric polymorphism and binary methods

An example, using a binary message ">":

```
fun sort(a:Array['T]):void {
  a.indices_do(&(i:int){
    a.indices_do(&(j:int){
      let a_i:T := fetch(a,i);
      let a_j:T := fetch(a,j);
      if(a_i > a_j, {      -- doesn't typecheck!
        store(a,i,a_j);
        store(a,j,a_i);
      });
    });
  });
}
```

Need to constrain T so that "$a_i$ > $a_j$" call is legal

Signature constraints work fine:

```
method sort(a:Array['T]):void
  where signature >(:T,:T):bool { ... }
```

But what if prefer a subtype constraint?



## `sort()` with a subtype constraint

Define a type for ordered things, along with its operations

- can include many useful default behaviors

```
abstract class Ordered;
  fun >(:Ordered, :Ordered):bool;
  fun <(x:Ordered, y:Ordered):bool { y > x }
  fun max(x:Ordered, y:Ordered):Ordered {
    if(x > y, { x }, { y }) }
  ...
```

Constrain type parameters using the new type:

```
fun sort(a:Array['T]):void where T <= Ordered {
  .. a_i:T .. a_j:T .. a_i > a_j .. }  -- typechecks!
```

## Implementing `Ordered`

Must provide implementations of `Ordered`

```
abstract class Ordered;
  fun >(:Ordered, :Ordered):bool;

class int isa Ordered;
  method >(i@int, j@int):bool { ... }

class string isa Ordered;
  method >(a@string, b@string):bool { ... }
```

Problem: > is incomplete!

Does this typecheck? run?

```
... 3 > "hi" ...
```



## Solution: F-bounded subtype constraint

Key idea: parameterize `Ordered` by the type of things that can be compared to

- then will need to be able to mention constrained type in its own bound (called an **F-bound**)

```
abstract class Ordered[T <= Ordered[T]];
  fun >(:'T <= Ordered[T], :'T):bool;
  fun max(x:'T <= Ordered[T], y:'T):T {
    if(x > y, { x }, { y }) }
  ...

fun sort(a:Array['T <= Ordered[T]]):void {
  .. a_i:T .. a_j:T .. a_i > a_j .. }  -- typechecks!

class int isa Ordered[int];
  method >(i@int, j@int):bool { ... }

class string isa Ordered[string];
  method >(a@string, b@string):bool { ... }
```

Can no longer compare ints and strings

> is now completely implemented



## Mutually-recursive F-bounded subtyping

Example: a framework of several classes whose instances refer to each other

Want to:

- write generic code for these classes
- refine the framework by creating subclasses

Problem: want to know *statically* that fields in subclasses store instances of *appropriate*, *corresponding* subclasses

```
-- framework
abstract class model[M,V]
    where M <= model[M,V], V <= view[M,V];
  field views(:model['M,'V]):list[V];
abstract class view[M,V]
    where M <= model[M,V], V <= view[M,V];
  field theModel(:view['M,'V]):M;

-- a refinement
class bitmap isa model[bitmap,bitViewer];
class bitViewer isa view[bitmap,bitViewer];
```

⇒ know that `bitViewer.theModel` returns `bitmap` *without rewriting the code*