Typechecking OO Languages

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

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

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

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

Simple idea: type = class

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

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

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


Interfaces

Previous declarations imply the following interfaces
- can also be declared explicitly if desired

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

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

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

Examples

Legal:

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

Illegal (static type error):

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

Method compatibility

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

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

Relaxed conditions (**covariance**):

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

Alternative relaxed conditions (**contravariance**):

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

Are either or both these relaxations safe?

Example

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

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

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

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


Overriding field types

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

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

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

-- an immutable field:
field bar(p@:Point):Bar;

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

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

What about fields overriding methods?

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

Binary methods and typechecking

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

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

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

Example of use:

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

Binary methods with multimethods

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

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

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

Example of use:

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

What semantics for mixed colored & plain points is desired?

Inheritance vs. subtyping

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

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

In Cecil, can program these separately:

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

Common case: inheritance and subtyping is parallel

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

Example

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

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

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


Benefits of separating inheritance and subtyping

Clarity of thinking

Sensible to implement interface w/o inheriting code

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

```
object Point;
  -- lots of Point methods here
  method =(p1@:Point, p2:Point):bool { ... }
object ColoredPoint inherits Point; --not subtype
  -- inherit most Point methods
  method =(p1@:ColoredPoint,
           p2:ColoredPoint):bool { ... }
```

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

Sometimes the two relations are opposite:

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

Costs of separating inheritance and subtyping

Verbosity in common case ⇒ need syntactic sugar

Complexity

Difficult to typecheck safety of inheriting w/o subtyping

Subtyping w/o inheritance cannot provide default implementations

Checking method implementations

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

Straightforward w/ single dispatching & monolithic classes:

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

Example

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

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

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


Checking multimethods

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

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

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

Research questions:

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

⇒ Dubious, MultiJava, EML

Example

Signature & method declarations:

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

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

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

Parameterized types

Want to add parameterized types

Simple approach:

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

```
template object Array[T] isa Collection[T];
  method fetch[T](a@:Array[T], i:int):T { ... }
  method store[T](a@:Array[T], i:int, v:T
              ):void { ... }
  method new_array[T](size:int):Array[T] {
    concrete object Array[T] { ... } }

let a:Array[string] := new_array[string](10);
...
store[string](a, 5, "hi");
....
let s:string := fetch[string](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 pass it explicitly; system infers it automatically

```
template object Array[T] isa Collection[T];
  method fetch(a@:Array[‘T], i:int):T { ... }
  method store(a@:Array[‘T], i:int, v:T
              ):void { ... }
  method new_array[T](size:int):Array[T] {
    concrete object Array[T] { ... } }

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


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 things with values of that type

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` can be printed?

Approach 1: subtype bound

Declare a type that has all the desired operations

```
type Printable;
  signature print(Printable):void;
```

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

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

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 {...}
```

Alternatively, can use bounds to support conditional subtyping

```
extend Array[`T <= Printable] isa Printable;
```

- parts in Java 1.5, new C#
- all in Cecil

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); }); }
```

- PolyJ, Cecil

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[`T]):void {
  a.do(&(elem:T){ print(elem); }); }
...
let a:Array[Foo] := ...;
print(a); -- macro-expand & check body of print
```

- 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);
  }); }
```

- 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 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

Most languages support only one of these approaches

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 a_i and a_j ... });
}
```

Need to guarantee 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?

`sort()` with a subtype constraint

```
method sort(a:Array[`T]):void
    where T <= Comparable {
  ... x:T ... y:T ... gt(x,y) ... }
signature gt(Comparable, Comparable):Bool;
```

For it to work, need the following declarations as well:

```
-- want to sort arrays of numbers or strings
extend Number isa Comparable;
extend String isa Comparable;
```

Problem: now the signature requires that we can compare (using `gt`) a number against a string

Solution: F-bounded subtype constraint

```
method sort(a:Array[`T]):void
    where T <= Comparable[T] {
  ... x:T ... y:T ... gt(x,y) ... }
signature gt(`T, `T):bool
  where T <= Comparable[T];
```

Now the extensions look like this:

```
extend Number isa Comparable[Number];
extend String isa Comparable[Number];
```

Can compare (using `gt`):

- a string against a string
- a number against a number or a sublass of number

Cannot compare:

- a number against a string

Another example of F-bounded subtyping

```
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 numbers returns a number

- can do `max` on mixed ints and floats, unlike Haskell

Mutually-recursive F-bounded subtyping

Example: a framework of several classes referring (storing in their fields instances of) to each other

Want to:

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

Problem: want to know *statically* that fields in subclasses store instances of subclasses, but not of the generic framework classes

```
-- framework
forall M,V where M<=model[M,V], V<=view[M,V]:
    abstract object model[M,V];
  field views(@:model[`M,`V]):list[V];
forall M,V where M<=model[M,V], V<=view[M,V]:
    abstract object view[M,V];
  field themodel(@:view[`M,`V]):M;

-- an instance
tem. object mymodel isa model[mymodel,myview];
tem. object myview isa view[mymodel,myview];
```

⇒ know that `mymodel.views` returns `list[myview]` *without rewriting the code*