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 implemented somehow
  - **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 multi-methods

With multi-methods 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):
- make sure each combination of classes "covered" by a signature invokes a single most-specific target method (is **complete** & **consistent**) that **conforms** to signature

Research questions:

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

Example

Signature & method declarations:

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

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

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;
```

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); }); }
```


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

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