Support for Polymorphism

• subsumption
  method defined on Numbers can work on Integers and Reals

```
method square(x:Number):Number { x * x }
```

• parametric polymorphism
  method works in a similar way for a variety of argument types

```
method fetch(a:Array of T, i:Integer):T {...}
```

OO languages that support parametric polymorphism:
- Trellis
- Theta
- Pizza, GJ, PolyJ, NextGen, ...
- C++
- Modula-3
- Eiffel
- Beta (kind of)
- ....
- Cecil

Parametric Polymorphism in Cecil

expressed with type variables, like ML:

```
forall T:
  method fetch(a:Array[T], i:Integer):T ...
forall T:
  method cons(elm:T, lst:List[T]):List[T] ...
```

more convenient syntax: `

```
method fetch(a:Array[`T], i:Integer):T ...
method cons(elm:`T, lst:List[T]):List[T] ...
```

objects parameterized, too:

```
forall T: template object Array[T];
```


Cf. Class-based Parameterization

class's type variable can be used in class's methods
- advantage: know what `T` is

```
class Array[`T] {
  method fetch(i:Integer):T ...;
  method store(val:T, i:Integer):void ...;
  ...
}
let a := new Array[String];
... a.fetch(0) ...     -- returns String
```

sometimes, need to parameterize each method

```
method create_table(
    values: Vector[`V],
    keys:   Vector[`K]
):Table[K,V]  { ... }
```


Instantiation and Inference of Type Parameters

When referencing a parameterized method or object, need to know the actual type parameters

often actuals can be inferred (*implicit* type parameters)

```
method fetch(a:Array[`T], i:Integer):T ...
let stringArr:Array[String] := ...;
... fetch(stringArr,5) ...   -- returns String
                             -- T is String
```

- more convenient

sometimes cannot infer

```
method new_array():Array[`T] ...
let my_array := new_array(); -- what is T ?
```

⇒ actuals to be specified by programmer
  (*explicit* type parameters)

```
let my_array := new_array[Bool](); -- T is Bool
```

Universal vs. bounded parametric polymorphism

as with ML andHaskell, want to know something about type variables, send some messages to values of those types

⇒ want to place constraints on type variables

Example

`print` method on `Array[T]` prints each element
works only if values of type `T` can be printed

how to express this constraint?

Approach 1: subtype bound

Declare a type that has all the desired operations

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

Constrain type variable to be subtype of `Printable` in `print` method

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

by-name subtyping: declared by programmer

Can use bounds to support conditional subtyping

```
extend Array[`T] subtypes Printable
  where T <= Printable;
```

"*Bounded polymorphism*" [Cardelli, Wegner 1985]

- Pizza *et al*, Cecil

Approach 2: signature bound

Express constraints directly as a required signature

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

similar to *structural subtyping*: inferred based on operations that types provide. *E.g.*,

```
type MyThing { ...; method print():void; ...}
```

structurally subtypes

```
type Printable { method print():void }
```

- Haskell
- Theta, PolyJ

Approach 3: check after instantiation

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

```
-- print message would not typecheck in Cecil
method print(a@:Array[`T]):void
{ a.do(&(elem:T){ print(elem) }); }
...
let a:Array[Foo] := ...;     -- macro-expand & check
                             -- body of print
print(a);
```

- C++, Modula-3

Approach 4: don't allow parameterized things

Do dynamic type casts from top type (`any` in Cecil, `Object` in Java) to desired/expected subtype when needed

```
method print(a@:Array):void
{
  a.do(&(elem:any){
    let pe:Printable := cast[Printable](elem);
    print(pe);
  });
}
```

- Java, C++, Beta, ...

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

Covariant Type Parameters

collection of more specific things can be used in place of read-only collection of more general things

```
object RO_Collection[`T];
method sum_elements(c:RO_Collection[Number]
                   ):Number ...

let intColl:RO_Collection[Integer] := ...;
...sum_elements(intColl)...   -- OK! returns Number
```

solution: *covariant* type parameter

```
extend
  RO_Collection[`T] subtypes RO_Collection[`S]
    where T <= S;
```

Eiffel, Java (built-in arrays) apply covariance to mutable collections as well — cannot guarantee safety statically

- Java - runtime checks
- Eiffel - unchecked (!)
  or expensive static data-flow analysis

Some languages provide way to specify co-, contra-, or non-variance of each type parameter

Polymorphism over binary methods

```
-- gt message send 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
{ ... gt(x:T,y:T) ... }
```

May prefer a subtype constraint over a signature constraint or language doesn't provide signature constraints

`sort()` with a subtype constraint

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

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

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

Problem: now the signature requires that we implement comparison (using `gt`) of a number and a string

```
... sort(a:Array[Ordered]) ...
-- a includes Numbers, Strings, ...
```


Solution: F-bounded subtype constraint

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

Now the extensions look like this:

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

Gives desired semantics:

- can compare a string with a string
  `... sort(a:Array[String]) ...`
- can compare a number with a number, or any two sublasses of number
  `... gt(x:Integer,y:Real) ...`
- statically disallows comparing a number with a string no longer can have `Array[Ordered]`
  `...gt(x:Integer,y:String)...` *TYPE ERROR*

Another example of F-bounded subtyping

```
method max(x:`T, y:`T):T
    where T <= Ordered[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
- inference of actual type parameter non-trivial (minimize result type)

Mutually-recursive Family of Classes

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

```
class Model {
  field views:List[Model];
}
class View {
  field myModel:Model;
}
```

Want to

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

For example

```
class DrawingModel isa Model { additional ops... };
class DrawingView isa View { additional ops... };
```

Problem: types of fields are not precise

```
let dview:DrawingView := ...;
... dview.myModel ...    -- want this to be DrawingModel
                         -- statically only expect Model
```

Solution: F-bounded Subtyping

parameterize each class by what other classes are in the family

```
forall M,V
where M <= Model[M,V],
      V <= View[M,V]
{
  class Model {
    field views:List[V];
  }
  class View {
    field myModel:M;
  }
}
class DrawingModel isa
    Model[DrawingModel,DrawingView];
class DrawingView isa
    View[DrawingModel,DrawingView];
```

parameterization provides mutual references

bounds guarantee references are (sub)types of `Model`,`View`

```
let dview:DrawingView := ...;
... dview.myModel ...   -- has type DrawingModel
  -- can send DrawingModel-specific messages to the result
```

Problem: complicated, verbose