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 { ... }
```

May prefer a subtype constraint over a signature constraint

`sort()` with a subtype constraint

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

Solution: F-bounded subtype constraint

```
method sort(a:Array[`T]):void
    where T <= Comparable[T] {
  ... gt(x:T, y:T) ... }
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 with a string
- a number with a number or a sublass of number

Cannot compare:

- a number with 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]:
    object model[M,V];
  field views(@:model[‘M,‘V]):list[V];
forall M,V where M<=model[M,V], V<=view[M,V]:
    object view[M,V];
  field themodel(@:view[‘M,‘V]):M;
-- an instance
object mymodel isa model[mymodel, myview];
object myview isa view[mymodel, myview];
```

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