Variance in Type Systems and Variance-Based Parametric Types

Based on Igarashi and Viroli’s paper from ECOOP 2002 (excellent paper! Value more in taste, than in novelty)

• This is a mechanism that got integrated in Java generics with different syntax
Subtyping

• Roughly, when a type is a subset of another

• What does that mean for method signatures? (covariance/contravariance of arguments result types)

• Consider (which one really defines a subset?):

```java
interface I1 {
    Animal foo(Dog d);
}
interface I2 extends I1 {
    Dog foo(Animal d);
}
interface I3 extends I1 {
    Object foo(PrettyDog d);
}
interface I4 extends I2 {
    Dog foo(Dog d);
}
```
Variance Flavors

- Covariance: $R <: S \Rightarrow C<R> <: C<S>$
- Contravariance: $R <: S \Rightarrow C<S> <: C<R>$
- Bivariance: $C<R> <: C<S>$, for all $R$ and $S$
- Invariance: $C<R> <: C<S> \Rightarrow R = S$

Question: How Can We Have Safe Variance?

Two basic principles, applied in a variety of mechanisms:

- C covariant in X means that X should not be the type of a public (and writeable—e.g., non-final) field or an argument type of a public method

- C contravariant in X means that X should not be the type of a public, readable field, or the return type of a public method
Classical, Restrictive Approach

class Pair<X extends Object, Y extends Object> {
    private X fst;
    private Y snd;
    Pair(X fst, Y snd) {
        this.fst = fts;
        this.snd = snd;
    }
    void SetFst(X fst) {
        this.fst = fst;
    }
    Y getSnd() { return snd; }
}

• Pair is covariant in Y, contravariant in X
  - why don’t constructors matter?

• E.g. Pair<Object, Integer> can be used
  where Pair<String, Number> is expected

• (Integer <: Number <: Object)
Limitations of Classical Approach

Usually we use the type parameter both in covariant and in contravariant roles

class Vector<X> {
    private X[] ar;

    Vector(int size){ar = new X[size];}
    int size(){return ar.length;}
    X getElementAt(int i){return ar[i];}
    void setElementAt(X t,int i) {
        ar[i] = t;
    }
}

• Too conservative to infer variance from code
New Insight

• Instead of conservatism, disallow some uses of methods based on the statically known type information

• Think of the same single code for Vector as defining 4 classes:
  - the regular Vector
  - the covariant Vector (with only read-only methods)
  - the contravariant Vector (only write-only methods)
  - the bivariant Vector (no methods with Xs in their parameter or return list—“frozen” Vector)
Variance Annotations

Three kinds of annotations:

- \( + \) : covariance (think “const” or “read-only”)
- \( - \) : contravariance (think “write-only”)
- \( * \) : bivariance (think “contents not touched”)

Interpretation:

- \( C^{+T} \) : the union of all invariant types of the form \( C^S \), where \( S < : T \)
  - \( C \) with \( T \) used only to read from

- \( C^{-T} \) : the union of all invariant types of the form \( C^S \), where \( T < : S \)
  - \( C \) with \( T \) used only to write to

- \( C^{*} \) : all invariant types of the form \( C^S \)

(Note I say “union”—types are sets of values)
Rules

(For multiple type parameters, the rules apply by varying a single parameter and keeping all others the same)

\[ C<T> \prec C<+T> \]
- Vector<Integer> \prec Vector<+Integer>

\[ C<T> \prec C<-T> \]
- Vector<Integer> \prec Vector<-Integer>

\[ C<+T> \prec C<*> \]
- Vector<+Integer> \prec Vector<*>

\[ C<-T> \prec C<*> \]
- Vector<-Integer> \prec Vector<*>

\[ S \prec T \Rightarrow C \prec S \prec C<+T> \]
- Vector<+Integer> \prec Vector<+Number>

\[ S \prec T \Rightarrow C \prec S \prec C<-S> \]
- Vector<-Number> \prec Vector<-Integer>
Example Applications: Covariance

class Vector<X> {
  ...
  void fillFrom(Vector<X> v, int i) {
    for (int j=i; j<v.size(); j++)
      setElementAt(
        v.getElementAt(j-i),j);
  }
}

Fills a vector (beginning at position \(i\)) by reading the contents of another vector. \(v\) is read-only, the method is covariant

Vector<Number> vn =
  new Vector<Number>(20);
Vector<Integer> vi = new Vector<Integer>(10);
Vector<Float> vf = new Vector<Float>(10);

vn.fillFrom(vi,0);
vn.fillFrom(vf,10);
Example Applications: Covariance

class Vector<X> {
...
void fillFromVector(
    Vector<+Vector<+X>> vv) {
    int pos = 0;
    for (int i=0; i<vv.size(); i++) {
        Vector<+X> v = vv.getElementAt(i);
        if (pos+v.size() >= size()) break;
        fillFrom(v,pos);
        pos +=v.size();
    }
}
}

Fills a vector with the contents of all vectors in a vector-of-vectors

E.g. the Vector<X> object (this) can be Vector<Number>, while vv is a Vector<Vector<+Number>> (e.g., holding a vector of Integers and a vector of floats)
Example Applications: Contravariance

class Vector<X> {
    ...
    void fillTo(Vector<-X> v, int i) {
        for (int j=i; j<v.size(); j++)
            v.setElementAt(
                getElementAt(j), j-i);
    }
}

Fills vector v by reading the contents of another vector (beginning at position i). v is write-only, the method is contravariant

Vector<Number> vn =
    new Vector<Number>(20);
Vector<Integer> vi = new
    Vector<Integer>(10);
Vector<Float> vf = new
    Vector<Float>(10);

vi.fillTo(vn, 0);
vf.fillTo(vn, 10);
Example Applications: Bivariance

int countVec(Vector<Vector<*>>* vv) {
    int sz = 0;
    for (int i=0; i < vv.size(); i++) {
        sz += vv.getElementAt(i).size();
    }
    return sz;
}

We count all elements of members of a vector-of-vectors. The second level vectors are not touched, the vector-of-vectors is only read.

As another example, think of a vector of pairs, where only the first element of each pair is read and the Vector is not modified:
Vector<Pair<X,*>>*
Assessment

- The variance annotations (which could be inferred if all the code is available for analysis) yield more generic code

- Similar to parametric (template) methods, with bounds on the template parameters
  - but need lower bounds, in addition to the usual “X extends C” (upper bound)
  - the mechanisms are complementary—each can do some things better than the other (read the paper for details!)

- Informally, parametric types with variance are like bounded existential types: e.g.,
  Vector<+C> is like a type
  <exists X <: C> Vector<X>