Taming the Wildcards

Combining Definition- and Use-Site Variance

John Altidor[1], Shan Shan Huang[2], and Yannis Smaragdakis[1,3]

University of Massachusetts Amherst[1]
LogicBlox Inc.[2]
University of Athens, Greece[3]

Outline

- Motivation for Variance.
- Existing Approaches to Variance.
- Our Approach: Combine Def-Site and Use-Site Variance.
- Case Study and Results.
- Summary.

Motivation for Variance

- Generics have been added to mainstream languages (e.g. Java, Scala, C#) to support parametric polymorphism.
- Generics conflict with subtyping.
- `Dog` <: `Animal` does *not* imply `List<Dog>` <: `List<Animal>`.

```
List<Dog> ld = new ArrayList<Dog>();
List<Animal> la = ld;
la.add(new Cat());
Dog d = ld.get(0); // Assigning a Cat to a Dog!
```

Introduction to Variance

- Under what conditions for type expressions *Exp1* and *Exp2* is `C<`*Exp1*`>` a subtype of `C<`*Exp2*`>`?
- Four common flavors of variance:
  1. *Covariance*: $T <: U \implies$ `C<`T`>` $<:$ `C<`U`>`
  2. *Contravariance*: $T <: U \implies$ `C<`U`>` $<:$ `C<`T`>`
  3. *Bivariance*: `C<`T`>` $<:$ `C<`U`>` for all T and U.
  4. *Invariance*: `C<`T`>` $<:$ `C<`U`>` $\implies$ $T <: U$ and $U <: T$.
- Existing specifications: *Definition-Site* and *Use-Site* Variance

Definition-Site Variance

- As in Scala, the definition of generic class `C[X]` determines its variance.
- Each type parameter is declared with a variance annotation.
- It is safe to assume `RList[Dog]` <: `RList[Animal]` and `Func[Real, String]` <: `Func[Int, Object]`.
- The variances of each type parameter's positions should be at most the declared variance.

```
class RList[+X] { def get(i:Int):X }
class Func[-K, +V] { def apply(k:K):V }
class WList[-X] { def set(i:Int, x:X):Unit }
```

Use-Site Variance

- Clients declare desired variance at *use*-site.
- Java Wildcards.
  - `List<? extends T>` - covariant instantiation
  - `List<? super T>` - contravariant instantiation
  - `List<?>` - bivariant instantiation
  - `List<T>` - invariant instantiation
- `List<? extends Animal>` can call "`Animal get(int i)`" but not "`void set(int i, Animal a)`".

```
void mapSpeak(List<? extends Animal> animals) {
  for(int i = 0; i < animals.size(); i++)
    animals.get(i).speak();
}
```

Definition-Site: Pros and Cons

- Conceptual simplicity; class definition specifies its variance.
- Burden on library designers; not on users.
- Forces splitting the definitions of data types into co-, contra-, bi-, and invariant versions.
  - `scala.collection.immutable.Map[A, +B]`
    `scala.collection.mutable.Map[A, B]`
  - A generic with n type parameters can require 3^n interfaces (or 4^n if bivariance is allowed).

Use-Site: Pros and Cons

- Flexibility: co-, contra-, and bivariant versions on the fly.
- Design classes in natural way, but burden shifts to users.
- Type signatures quickly become complicated.
- Heavy variance annotations required for subtyping; from Apache Commons-Collections Library:

```
Iterator<? extends Map.Entry<? extends K,V>>
  createEntrySetIterator(
     Iterator<? extends Map.Entry<? extends K,V>>)
```

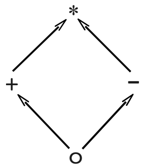
Our Approach: Combine Def-Site and Use-Site

- Take advantage of simplicity of def-site and flexibility of use-site variance.
- Flexibility of use-site removes need for redundant classes.
- Simplicity of def-site takes complexity burden off clients, and requires far fewer variance annotations for subtyping.
- Enable reasoning about classes with *both* def-site and use-site annotations.

Integrating Use-Site with Def-Site

```
class C[-X] { def set(arg1:X):Unit }
class D[+X] { def compare(arg2:C[+X]):Unit }
```

- `C[+X]` says to pass `X` to a version of `C` that is *at least* covariant.
- Use-site annotation corresponds to a *join* operation in the standard variance lattice.



- Variance of `X` in `C[`v_u`X]` is $v_c \sqcup v_u$, where v_c is def-site var of `C`.

Variance Composition

- What is variance of `X` in `A<B<C<X>>>`? (Ignore use-site)
- In general, what is variance of `X` in `C<`*E*`>`?
- Defined "transform" binary operator $\otimes$ to reason about variance of arbitrarily nested type expressions.
- $v_1 \otimes v_2 = v_3$
  If the variance of a type variable `X` in type expression E is v_2 and the def-site variance of class `C` is v_1, then the variance of `X` in type expression `C<`*E*`>` is v_3.

Deriving Transform Operator

- Example Case $+ \otimes - = -$
  Need to show $\texttt{C<}E\texttt{>}$ is contravariant wrt $\texttt{X}$ when generic $\texttt{C}$ is covariant in its type parameter and type expression E is contravariant in $\texttt{X}$. This holds because, for any T_1, T_2:

$$
\begin{aligned}
T_1 <: T_2 &\implies && \text{(by contravariance of } E\text{)} \\
E[T_2/\texttt{X}] <: E[T_1/\texttt{X}] &\implies && \text{(by covariance of } \texttt{C}\text{)} \\
\texttt{C<}E[T_2/\texttt{X}]\texttt{>} <: \texttt{C<}E[T_1/\texttt{X}]\texttt{>} &\implies && \\
\texttt{C<}E\texttt{>}[T_2/\texttt{X}] <: \texttt{C<}E\texttt{>}[T_1/\texttt{X}] & &&
\end{aligned}
$$

  Hence, $\texttt{C<}E\texttt{>}$ is contravariant with respect to $\texttt{X}$.
- See paper for remaining cases.

Summary of Transform

- Invariance transforms everything into invariance.
- Bivariance transforms everything into bivariance.
- Covariance preserves a variance.
- Contravariance reverses it.

Definition of variance transformation: $\otimes$

$+ \otimes + = +$	$- \otimes + = -$	$* \otimes + = *$	$o \otimes + = o$
$+ \otimes - = -$	$- \otimes - = +$	$* \otimes - = *$	$o \otimes - = o$
$+ \otimes * = *$	$- \otimes * = *$	$* \otimes * = *$	$o \otimes * = o$
$+ \otimes o = o$	$- \otimes o = o$	$* \otimes o = *$	$o \otimes o = o$

Def-Site Inference via Variance Calculus: *VarLang*

- Java Classes:

```
class C<X> {
  X    foo (C<? super X>   csx) { ... }
  void bar (D<? extends X> dsx) { ... }
}
class D<Y> { void baz (C<Y> cx) { ... }  }
```

- Translation to *VarLang*:

```
module C<X> { X+, C<-X>-, void+, D<+X>- }
module D<Y> { void+, C<oY>- }
```

Constraint Generation

```
module C<X> { X+, C<-X>-, void+, D<+X>- }
module D<Y> { void+, C<oY>- }
```

- Generate constraints from *VarLang* modules

foo return type $\Longrightarrow$ $c \sqsubseteq + \otimes \underbrace{+}_{\texttt{X}} = +$

foo arg type $\Longrightarrow$ $c \sqsubseteq - \otimes \underbrace{(c \sqcup -)}_{\texttt{C<-X>}}$

bar arg type $\Longrightarrow$ $c \sqsubseteq - \otimes \underbrace{(d \sqcup +)}_{\texttt{D<+X>}}$

baz arg type $\Longrightarrow$ $d \sqsubseteq - \otimes \underbrace{(c \sqcup o)}_{\texttt{C<oY>}}$

Constraints Enable Checking

```
class C<X> {
  X    foo (C<? super X>   csx) { ... }
  void bar (D<? extends X> dsx) { ... }
}
class D<Y> { void baz (C<Y> cx) { ... }  }
```

- `C` cannot be contravariant.
  - `foo` return type $\implies c \sqsubseteq +$ but $- \not\sqsubseteq +$
- Constraints correspond to checking def-site variance annotations.

Constraint Solving Enables Inference

`foo` return type	$\Longrightarrow$	$c \sqsubseteq +$
`foo` arg type	$\Longrightarrow$	$c \sqsubseteq - \otimes (c \sqcup -)$
`bar` arg type	$\Longrightarrow$	$c \sqsubseteq - \otimes (d \sqcup +)$
`baz` arg type	$\Longrightarrow$	$d \sqsubseteq - \otimes (c \sqcup o)$

- Trivial solution: $c = o$ and $d = o$.
- Most general solution: $c = +$ and $d = -$.
- Solve constraints by fix-point computation running in polynomial of the program size (# of constraints).

Case Study: Def-Site Inference for Java

- Mapped Java classes to *VarLang* modules.
  - Argument types map to contravariant positions.
  - Types of non-final fields map to covariant and contravariant.
  - etc.
- Applied inference to large, standard Java libraries.
- Example inferences: `java.util.Iterator<E>` is covariant and `java.util.Comparator<T>` is contravariant.

Sample Results from Inference

Library		# Type defs	# Gen defs	Type Definitions invar.	variant
java.*	classes	5550	99	69%	31%
	interfaces	1710	44	43%	57%
	total	7260	143	61%	39%
JScience	classes	70	25	76%	24%
	interfaces	51	11	55%	45%
	total	121	36	69%	31%
Apache Collec.	classes	226	187	66%	34%
	interfaces	23	22	55%	45%
	total	249	209	65%	35%
Google Guava	classes	204	101	90%	10%
	interfaces	35	26	46%	54%
	total	239	127	81%	19%

- Analysis was modular but conservative (e.g. ignored method bodies).
  - "`foo(List<Animal> arg)`" could have been "`foo(List<? extends Animal> arg)`".

Summary

- Combine def-site and use-site variance to reap their advantages and remove their disadvantages.
- Our reasoning enables adding def-site variance inference to Java and checking Scala classes with use-site variance annotations.
- Analysis over Java libraries shows potential impact even with a conservative analysis.
- See PLDI 2011 paper for further details.