UMassAmherst

Java Wildcards Meet Definition-Site Variance

John Altidor[1]

Christoph Reichenbach[2,1]

Yannis Smaragdakis[3,1]

[1]University of Massachusetts
[2]Google
[3]University of Athens

Outline

- Motivation for Variance.
- Brief History, Existing Approaches.
- What Is New Here:
  Combine Definition-Site and Use-Site Variance.
  - Both in a single language, each using the other
  - VarJ Formal Calculus
  - Insights on Formal Reasoning
- Summary.

Subtyping – Inclusion Polymorphism

- Promotes reusability.
- Example: Java inheritance.

```
class Animal {
  void speak() { }
}
class Dog extends Animal {
  void speak() { print(“bark”); }
}
class Cat extends Animal {
  void speak() { print(“meow”); }
}
```

Generics – Parametric Polymorphism

type parameter

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

- `List<Animal>` ≡ `List` of `Animal`s
- `List<Dog>` ≡ `List` of `Dog`s

Generics – Parametric Polymorphism

type parameter

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

write X

- **List<Animal>** ≡ **List** of **Animals**
- **List<Dog>** ≡ **List** of **Dogs**

Generics – Parametric Polymorphism

type parameter

```
class List<X>
{
   void add(X x) { … }

   X get(int i) { … }

   int size() { … }
}
```

write X

read X

- **List<Animal>** ≡ **List** of **Animals**
- **List<Dog>** ≡ **List** of **Dogs**

Generics – Parametric Polymorphism

type parameter

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

write X

read X

no X

- **List<Animal>** ≡ **List** of **Animals**
- **List<Dog>** ≡ **List** of **Dogs**

Generics – Parametric Polymorphism

type parameter

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

write X

read X

no X

- **List<Animal>** ≡ **List** of **Animals**
- **List<Dog>** ≡ **List** of **Dogs**

Customized Lists

Generics and Subtyping

- Dog **<:** Animal (Dog **is an** Animal).
- Cat **<:** Animal (Cat **is an** Animal).

- List<Dog> **<:** List<Animal> 

Generics and Subtyping

- Dog **<:** Animal (Dog **is an** Animal).
- Cat **<:** Animal (Cat **is an** Animal).

- List<Dog> **<:** List<Animal> ?

No!

- A List<Animal> can add a Cat to itself.
  A List<Dog> cannot.

Variance Introduction

- When is **C<***Expr1***>** a subtype of **C<***Expr2***>**?

```
class RList<X>
{
  X get(int i) { … }

  int size() { … }
}
```

Can **read** from but not **write** to.

- It is safe to assume **RList<Dog>** <: **RList<Animal>**.
- Why?

Flavors of Variance - Covariance

- Assuming Dog **<:** Animal (Dog **is an** Animal).

`Generic<Dog>` **<:** **`Generic<Animal>`**

Covariance

Flavors of Variance - Contravariance

- Assuming Dog <: Animal (Dog **is an** Animal).

```
Generic<Animal>   <:   Generic<Dog>
```

Contravariance

Four Flavors of Variance

Covariance:	T **<:** U	⇒ C<T> **<:** C<U>.
Contravariance:	T **<:** U	⇒ C<U> **<:** C<T>.
Bivariance:	C<T> **<:** C<U>,	for all T and U.
Invariance:	C<T> **<:** C<U>,	if T **<:** U and U **<:** T.

- How do programmers specify variance?

Definition-Site Variance (C#/ Scala)

- Programmer specifies variance **in definition** as in Scala and C#.
- Variance of a **type position**.
  - Return types: covariant.
  - Arguments types: contravariant.

```
class RList<+X> {
  X get(int i) { … }
  int size() { … }
  // no method to add
}
```

```
class WList<-X> {
  void add(X x) { … }
  int size() { … }
  // no method to get
}
```

Definition-Site Variance (C#/ Scala)

- Programmer specifies variance **in definition** as in Scala and C#.
- Variance of a **type position**.
  - Return types: covariant.
  - Arguments types: contravariant.

```
class RList<+X> {
  X get(int i) { … }
  int size() { … }
  // no method to add
}
```

```
class WList<-X> {
  void add(X x) { … }
  int size() { … }
  // no method to get
}
```

Definition-Site Variance (C#/ Scala)

- Programmer specifies variance **in definition** as in Scala and C#.
- Variance of a **type position**.
  - Return types: covariant.
  - Arguments types: contravariant.

```
class RList<+X> {
  X get(int i) { … }
  int size() { … }
  // no method to add
}
```

```
class WList<-X> {
  void add(X x) { … }
  int size() { … }
  // no method to get
}
```

Definition-Site Variance (C#/ Scala)

- Programmer specifies variance **in definition** as in Scala and C#.
- Variance of a **type position**.
  - Return types: covariant.
  - Arguments types: contravariant.

```
class RList<+X> {
  X get(int i) { … }
  int size() { … }
  // no method to add
}
```

```
class WList<-X> {
  void add(X x) { … }
  int size() { … }
  // no method to get
}
```

Use-Site Variance (Java Wildcards)

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

Use-Site Variance (Java Wildcards)

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

List<? extends X>

UMassAmherst

Use-Site Variance (Java Wildcards)

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

List<? super X>

Use-Site Variance (Java Wildcards)

```
class List<X>
{
  void add(X x) { … }

  X get(int i) { … }

  int size() { … }
}
```

List<?>

Brief History: Definition-Site Variance

- Variance has been subject of many (ECOOP) papers.
- Definition-site variance has long history.
  - Introduced in late 80's:
    - Cook, ECOOP '89.
    - America & van der Linden, ECOOP '90.
    - Bracha & Griswold, OOPSLA, '93.
  - Formalized for C#: Emir et al, ECOOP '06.
  - Decidability of subtyping w/ variance: Kennedy & Pierce, FOOL '07.
    - Undecidable in general. Decidable fragment.

Brief History: Use-Site Variance

- Introduced:
  Thorup & Torgersen, ECOOP ’99.
- Generalized and formalized:
  Igarashi & Viroli, ECOOP ‘02.
- Adopted by Java as Wildcards:
  Torgersen et al, SAC ’04.
- Soundness of Java Wildcards:
  Cameron et al, ECOOP ‘08.
- Decidability still open:
  Kennedy & Pierce, FOOL ‘07.
- Decidable fragment: Tate et al, PLDI ’11.

Definition-Site vs. Use-Site Variance

- Definition-Site Cons:
  - Redundant Types:
    - **`scala.collection.immutable.Map<A, +B>`**
    - **`scala.collection.mutable.Map<A, B>`**
    - Generic with *n* parameters ⇒ 3^n interfaces (or 4^n if bivariance is allowed).
- Use-Site Variance Cons:
  - Type signatures quickly become complicated.

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

Wildcards Criticism

- **"We simply cannot afford another wildcards" – Joshua Bloch.**
- **"Simplifying Java Generics by Eliminating Wildcards" – Howard Lovatt.**

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

Our Approach: Take Best of Both Worlds

- Take advantages. Remove disadvantages.
  - **Simpler type** expressions in Java (burden off clients).
  - **Less redundant** type definitions in C# and Scala.

- Adding **explicit** definition-site annotations to Java.

- VarJ Calculus.
  - Directly extends TameFJ with definition-site variance.
- Extends denotational approach:
  PLDI 2011 (Altidor, Huang, Smaragdakis)

Fewer Wildcard Annotations

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

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

Extending Java with Definition-Site Variance

- Both kinds of annotations: easier for programmer, harder to typecheck

```
class ROStack1<+X> {
  X pop() { ... }
  List<? extends X>  toList() {... }
}
```

Extending Java with Definition-Site Variance

- Both kinds of annotations: easier for programmer, harder to typecheck

```
class ROStack1<+X> {
  X pop() { ... }
  List<? extends X> toList() {... }
}
```

Extending Java with Definition-Site Variance

- Both kinds of annotations: easier for programmer, harder to typecheck

```
class ROStack1<+X> {
  X pop() { ... }
  List<? extends X>  toList() {... }
}
```

Extending Java with Definition-Site Variance

- Both kinds of annotations: easier for programmer, harder to typecheck

```
class ROStack1<+X> {
  X pop() { ... }
  List<? extends X>  toList() {... }
}
```

✓

```
class ROStack2<+X> {
  X pop() { ... }
  <Y extends X> List<Y> toList() { ... }
}
```

Method Type Parameter

Extending Java with Definition-Site Variance

- Both kinds of annotations: easier for programmer, harder to typecheck

```
class ROStack1<+X> {
  X pop() { ... }
  List<? extends X>  toList() {... }
}
```

```
class ROStack2<+X> {
  X pop() { ... }
  <Y extends X> List<Y> toList() { ... }
}
```

Method Type Parameter

VarJ: Java Calculus modeling Wildcards and Definition-Site Variance

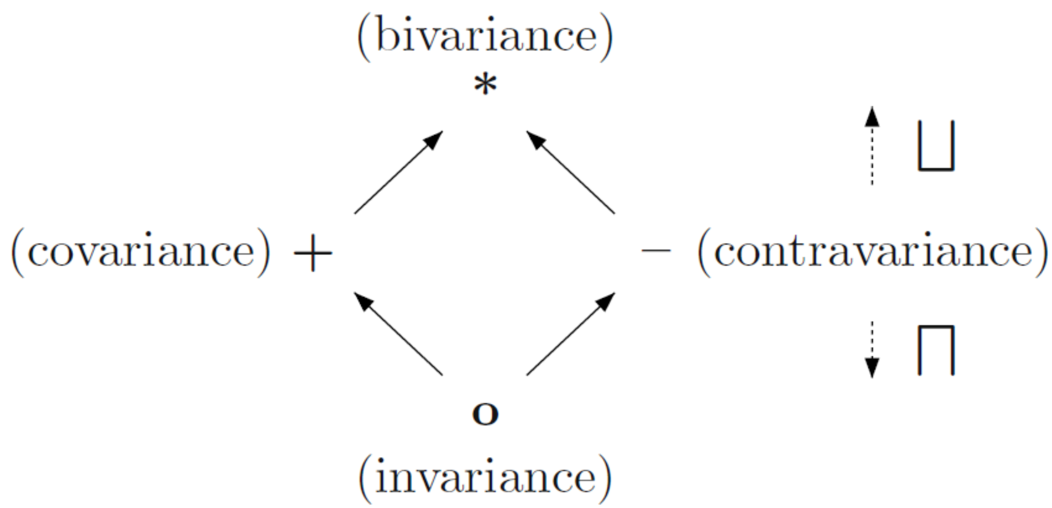
- Adding **explicit** definition-site annotations to Java.
  - Directly extends TameFJ with definition-site variance.
- Supports **full** complexities of the Java realization of variance.
  - Existential Types
  - Polymorphic Methods
  - Wildcard Capture
  - F-Bounded Polymorphism
- Type `Stack<? extends String>` modeled as `∃X->[⊥-String].Stack<X>`.

VarJ: Java Calculus modeling Wildcards and Definition-Site Variance

- Adding **explicit** definition-site annotations to Java.
  - Directly extends TameFJ with definition-site variance.
- Supports **full** complexities of the Java realization of variance.
  - Existential Types
  - Polymorphic Methods
  - Wildcard Capture
  - F-Bounded Polymorphism
- Type `Stack<? extends String>` modeled as `∃X->[⊥-String].Stack<X>`.

modeled as unknown (existential) type with lower and upper bound

Standard Modeling: Variance Lattice



- Ordered by subtype constraint
  (convention: also consider variances to be binary predicates).

- $+(T; T') \equiv T <: T'$
- $-(T; T') \equiv T' <: T$
- $o(T; T') \equiv +(T; T') \wedge -(T; T')$
- $*(T; T') \equiv true$

$$v \leq w \implies \left[v(T; T') \implies w(T; T') \right]$$

Variance of a Type

- When is `C<`*Expr1*`>` a subtype of `C<`*Expr2*`>`?
- What about existential types?
  `∃X->[⊥-String].Stack<X>`
- We answer more general question:
  When is `[U/X]T <: [U'/X]T`?
- **Key**: defined very general predicate:

  var(`X`; `T`)

  =

  variance of type `T` *with respect to type variable* `X`*.*

Variance of a Type

- When is `C<`*`Expr1`*`>` a subtype of `C<`*`Expr2`*`>`?
- What about existential types?
  `∃X->[⊥-String].Stack<X>`
- We answer more general question:
  When is `[U/X]T <: [U'/X]T`?
- **Key**: defined very general predicate:

  var(`X`; `T`)
  =
  variance of type `T` *with respect to type variable* `X`.

Variance of a Type

- When is `C<`*`Expr1`*`>` a subtype of `C<`*`Expr2`*`>`?
- What about existential types?
  ∃X->[⊥-String].Stack<X>
- We answer more general question:
  When is **[U/X]T <: [U′/X]T**?
- **Key**: defined very general predicate:

  var(**X**; **T**)

  =

  *variance of type **T** with respect to type variable **X**.*

Variance of a Type

- When is `C<`*Expr1*`>` a subtype of `C<`*Expr2*`>`?
- What about existential types?
  ∃X->[⊥-String].Stack<X>
- We answer more general question:
  When is **[U/X]T <: [U'/X]T**?
- **Key**: defined very general predicate:

  var(**X**; **T**)
  =
  variance of type ***T*** *with respect to type variable* ***X***.

Subtype Lifting Lemma

- We generalize Emir et al.'s subtype lifting lemma.
- Goal property of var.

If:

(a) $v \leq \text{var}(\mathbf{X}; \mathbf{T})$

(b) $v(\mathbf{U}; \mathbf{U'})$

Then: `[U/X]T <: [U'/X]T`

- var(`X; Iterator<X>`) = **+**
  and **+**`(Dog; Animal)` $\equiv$ `Dog <: Animal`

  implies `Iterator<Dog> <: Iterator<Animal>`

Subtype Lifting Lemma

- We generalize Emir et al.’s subtype lifting lemma.
- Goal property of var.

If:
(a) $v \leq \text{var}(\mathtt{X};\ \mathtt{T})$
(b) $v(\mathtt{U};\ \mathtt{U'})$
Then: $[\mathtt{U}/\mathtt{X}]\mathtt{T} <: [\mathtt{U'}/\mathtt{X}]\mathtt{T}$

- $\text{var}(\mathtt{X};\ \mathtt{Iterator<X>}) = +$
and $+(\mathtt{Dog};\ \mathtt{Animal}) \equiv \mathtt{Dog} <: \mathtt{Animal}$

implies `Iterator<Dog> <: Iterator<Animal>`

Subtype Lifting Lemma

- We generalize Emir et al.'s subtype lifting lemma.
- Goal property of var.

If:

(a) $v \leq \mathrm{var}(\mathbf{X}; \mathbf{T})$

(b) $v(\mathbf{U};\ \mathbf{U'})$

Then: $[\mathbf{U}/\mathbf{X}]\mathbf{T}$ `<:` $[\mathbf{U'}/\mathbf{X}]\mathbf{T}$

- var(**X**; `Iterator<X>`) = **+**

and **+**(`Dog; Animal`) $\equiv$ `Dog <: Animal`

implies `Iterator<Dog> <: Iterator<Animal>`

Subtype Lifting Lemma

- We generalize Emir et al.'s subtype lifting lemma.
- Goal property of var.

If

(a) v $\leq$ var(**X**; **T**)

(b) v(**U**; **U′**)

Then: `[U/X]T <: [U′/X]T`

- var(**X**; `Iterator<X>`) = **+**

and **+**`(Dog; Animal)` $\equiv$ `Dog <: Animal`

implies `Iterator<Dog> <: Iterator<Animal>`

Variance Composition

- Variance of variable **`X`** in type **`A<B<C<X>>>`**?
- In general, variance of variable **`X`** in type `C`<*E*>?

- $v_1 \otimes v_2 = v_3$. If:
  - Variance of variable **`X`** in type expression *E* is v_2.
  - The def-site variance of class **`C`** is v_1.
  - Then: variance of **`X`** in `C`<*E*> is v_3.

Variance Composition

- Variance of variable **X** in type **A<B<C<X>>>**?
- In general, variance of variable **X** in type **C**<*E*>?

- $v_1 \otimes v_2 = v_3$. If:
  - Variance of variable **X** in type expression *E* is v_2.
  - The def-site variance of class **C** is v_1.
  - Then: variance of **X** in **C**<*E*> is v_3.

Variance Composition

- Variance of variable **`X`** in type **`A<B<C<X>>>`**?
- In general, variance of variable **`X`** in type **`C`**<*E*>?

Transform Operator

- $v_1 \otimes v_2 = v_3$. If:
  - Variance of variable **`X`** in type expression *E* is v_2.
  - The def-site variance of class **`C`** is v_1.
  - Then: variance of **`X`** in **`C`**<*E*> is v_3.

UMassAmherst

Deriving Transform Operator

Example Case: **+** $\otimes$ **–** **=** **–**

- Class C is **covariant**.
- Type E is **contravariant in X**.
- Need to show C<E> is **contravariant in X**.
- For any T_1, T_2:

- Hence, C<*E*> **contravariant in X**.

Deriving Transform Operator

Example Case: **+ $\otimes$ − = −**

- Class C is **covariant**.
- Type E is **contravariant in X**.
- Need to show C<E> is **contravariant in X**.
- For any T_1, T_2:

- Hence, C<*E*> **contravariant in X**.

Deriving Transform Operator

Example Case: $+ \otimes - = -$

- Class C is **covariant**.
- Type E is **contravariant in X**.
- Need to show C<E> is **contravariant in X**.
- For any T_1, T_2:

- Hence, C<*E*> **contravariant in X**.

Deriving Transform Operator

Example Case: **$+ \otimes - = -$**

- Class C is **covariant**.
- Type E is **contravariant in X**.
- Need to show C<E> is **contravariant in X**.
- For any T_1, T_2:

- Hence, C<*E*> **contravariant in X**.

Deriving Transform Operator

Example Case: $+ \otimes - = -$

- Class C is **covariant**.
- Type E is **contravariant in X**.
- Need to show C<E> is **contravariant in X**.
- For any T_1, T_2:

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

- Hence, **C<*E*> contravariant in X**.

Summary of Transform

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

Definition of variance transformation: $\otimes$

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

Definition of **var** predicate

Variance of Types and Ranges: $var(\mathtt{X}; \phi)$, where $\phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta$

$$var(\mathtt{X}; \mathtt{X}) = + \qquad \text{(VAR-XX)}$$

$$var(\mathtt{X}; \mathtt{Y}) = *, \text{ if } \mathtt{X} \neq \mathtt{Y} \qquad \text{(VAR-XY)}$$

$$var(\mathtt{X}; \mathtt{C}\text{<}\overline{\mathtt{T}}\text{>}) = \textstyle\prod_{i=1}^{n} \left(\mathtt{v}_i \otimes var(\mathtt{X}; \mathtt{T}_i)\right), \text{ if } VT(\mathtt{C}) = \overline{\mathtt{vX}} \qquad \text{(VAR-N)}$$

$$var(\mathtt{X}; \bot) = * \qquad \text{(VAR-B)}$$

$$var(\mathtt{X}; \exists\Delta.\mathtt{R}) = var(\mathtt{X}; \Delta) \sqcap var(\mathtt{X}; \mathtt{R}), \text{ if } \mathtt{X} \notin dom(\Delta) \qquad \text{(VAR-T)}$$

$$var(\mathtt{X}; \overline{\mathtt{Y} \to [\mathtt{B}_L\text{-}\mathtt{B}_U]}) = \textstyle\prod_{i=1}^{n} \left(\left(- \otimes var(\mathtt{X}; \mathtt{B}_{Li})\right) \sqcap \left(+ \otimes var(\mathtt{X}; \mathtt{B}_{Ui})\right)\right) \qquad \text{(VAR-R)}$$

$$\left[var(\overline{\mathtt{X}}; \phi) = \overline{\mathtt{v}}\right] \equiv \left[\forall i, var(\mathtt{X}_i; \phi) = \mathtt{v}_i\right], \text{ where } \phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta \qquad \text{(VAR-SEQ)}$$

- See paper for further cases.

Definition of **var** predicate

Variance of Types and Ranges: $var(\mathtt{X};\phi)$, where $\phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta$

$var(\mathtt{X};\mathtt{X}) = +$ (VAR-XX)

$var(\mathtt{X};\mathtt{Y}) = *$, if $\mathtt{X} \neq \mathtt{Y}$ (VAR-XY)

$var(\mathtt{X};\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \prod_{i=1}^{n}(\mathtt{v}_i \otimes var(\mathtt{X};\mathtt{T}_i))$, if $VT(\mathtt{C}) = \overline{\mathtt{vX}}$ (VAR-N)

definition-site variance table

$var(\mathtt{X};\bot) = *$ (VAR-B)

$var(\mathtt{X};\exists\Delta.\mathtt{R}) = var(\mathtt{X};\Delta) \sqcap var(\mathtt{X};\mathtt{R}$ … (VAR-T)

$var(\mathtt{X};\overline{\mathtt{Y} \rightarrow [\mathtt{B}_L\mathtt{-B}_U]}) = \prod_{i=1}^{n}((- \otimes$ … $(\mathtt{X};\mathtt{B}_{U i})))$ (VAR-R)

$[var(\overline{\mathtt{X}};\phi) = \overline{\mathtt{v}}] \equiv [\forall i, var(\mathtt{X}_i;\phi) = \mathtt{v}_i]$, where $\phi ::= \mathtt{B} \mid \mathtt{R} \mid \Delta$ (VAR-SEQ)

- See paper for further cases.

Type Checking Variance

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{vX} \to [\ldots]}\texttt{>} \lhd \texttt{N} \{ \; \ldots \; \} \\ \boxed{\overline{\texttt{v}} \le var(\overline{\texttt{X}}; \texttt{T})} \quad \boxed{\overline{- \otimes \texttt{v}} \le var(\overline{\texttt{X}}; \overline{\texttt{Y} \to [\texttt{B}_L\texttt{-}\texttt{B}_U]})} \\ \boxed{\overline{- \otimes \texttt{v}} \le var(\overline{\texttt{X}}; \texttt{T}_i), \ \text{for each } i} \\ \ldots \end{array}}{\vdash \texttt{<}\overline{\texttt{Y} \to [\texttt{B}_L\texttt{-}\texttt{B}_U]}\texttt{> T m(}\overline{\texttt{T x}}\texttt{)} \ \{ \ \texttt{return e;} \ \} \ OK \text{ in } \texttt{C}}$$

(W-METH)

- Definition-site variance annotations are type checked using *var* predicate and assumed variances of each position.
- Soundness proof verifies variance reasoning is safe.

General Theory – Template for Adding Variance

- Variance composition:
  $v_1 \otimes v_2 = v_3$
- Variance binary predicate:
  $v(\mathbf{T};\ \mathbf{T}')$
- Variance lattice:
  $v_1 \leq v_2$
- Variance of a type:
  $var(\mathbf{X};\ \mathbf{T})$
- Relating variance to subtyping:
  Subtype Lifting Lemma
- Variance of a position: See paper

Summary of Contributions

- Generics and subtyping coexist fruitfully.
- Model full complexities of Java wildcards **and** interaction with definition-site.
- Generalizes all previous related work.
- Resolve central questions in the design of any language involving parametric polymorphism and subtyping.
  - Variance of various types (e.g. existential types).
  - Variance of a position (e.g. polymorphic type bounds).
- Paper explains a lot more.