An Interval-based Inference of Variant Parametric Types ⋆

Florin Craciun[1], Wei-Ngan Chin[2], Guanhua He[1], and Shengchao Qin[1]

[1] Department of Computer Science, Durham University, UK
[2] Department of Computer Science, National University of Singapore, Singapore

Abstract. Variant parametric types represent the successful integration of subtype and parametric polymorphism to support a more flexible subtyping for Java-like languages. A key feature that helps strengthen this integration is the use-site variance. Depending on how the fields are used, each variance denotes a covariant, a contravariant, an invariant or a bivariant subtyping. By annotating variance properties on each type argument to a parametric class, programmers can choose various desirable variance properties for each use of the parametric class. Although Java library classes have been successfully refactored to use variant parametric types, these mechanisms are often criticized, due to the difficulty of choosing appropriate variance annotations. Several algorithms have been proposed for automatically refactoring legacy Java code to use generic libraries, but none can support the full flexibility of the use-site variance-based subtyping. This paper addresses this difficulty by proposing a novel interval-based approach to inferring both the variance annotations and the type arguments. Each variant parametric type is regarded as an interval type with two type bounds, a lower bound for writing and an upper bound for reading. We propose a constraint-based inference algorithm that works on a per method basis, as a summary-based analysis.

1 Introduction

Recently, several mainstream object-oriented languages, such as Java and C#, have successfully integrated traditional *subtype polymorphism* and *parametric polymorphism* to support better type-safe reusable code with significant reduction of runtime cast operations. Subtype polymorphism is a nominal relation, based on a given class hierarchy. Parametric polymorphism allows a data or a function to be parameterized by types and supports structural subtyping [1]. In handling objects with mutable fields, a crucial feature that helps strengthen the integration of subtype and parametric polymorphism is the adoption of *variance*. Variance annotations predict the flow of values for fields and provide a richer subtyping hierarchy. Depending on how the fields are being accessed, each variance denotes a covariant, a contravariant, an invariant or a bivariant subtyping. Generics types of Java 5 (also called Wildcard Types) [23, 24, 12] are based on the variant parametric types (or VPTs) [14]. VPTs is based on *use-site variance* whereby each use of a class type is marked with suitable variances that indicate how the fields are to be accessed.

⋆ The work is supported in part by the EPSRC project EP/E021948/1.

Variant Parametric Types. Consider a variant parametric class `Pair` with two fields which are captured as type parameters:

$$\texttt{class Pair}\langle \texttt{A}, \texttt{B}\rangle \ \{\ \texttt{A fst};\ \ \texttt{B snd};\ \ \cdots\ \}$$

Assume three methods to retrieve the first field, to set the second field and to swap the two fields for a `Pair` object. In these methods, the parameter `this` is the `Pair` object whose variant parametric type must be provided with suitable variances. The type of the `this` parameter is specified prior to delimiter '|' (as in [4]):

$\texttt{Pair}\langle \oplus\texttt{A}, \circledast\rangle$ | $\texttt{C getFst}\langle \texttt{A}, \texttt{C}\rangle()$ `where` $\texttt{A}<:\texttt{C}$ { `return this.fst;` }

$\texttt{Pair}\langle \circledast, \ominus\texttt{B}\rangle$ | $\texttt{void setSnd}\langle \texttt{B}, \texttt{C}\rangle(\texttt{C y})$ `where` $\texttt{C}<:\texttt{B}$ { `this.snd=y;` }

$\texttt{Pair}\langle \odot\texttt{A}, \odot\texttt{A}\rangle$ | $\texttt{void swap}\langle \texttt{A}\rangle()$ { `A y=this.fst; this.fst=this.snd; this.snd=y;` }

As can be seen, four kinds of variance annotations (denoted by α) are possible: (i) $\alpha = \oplus$ captures a *flow-out* from the field to support covariant subtyping; (ii) $\alpha = \ominus$ captures a *flow-in* to the field to support contravariant subtyping; (iii) $\alpha = \odot$ captures both *flow-in* and *flow-out* to support invariant subtyping; and (iv) $\alpha = \circledast$ captures *no access* for the field to support bivariant subtyping. For simplicity, $\circledast\texttt{t}$ can be abbreviated as $\circledast$. More generally, given an object with variant parametric type $\texttt{c}_1\langle\alpha_1\texttt{t}_1\rangle$, we may pass it to a location with type $\texttt{c}_2\langle\alpha_2\texttt{t}_2\rangle$, in accordance with the following subsumption relations:

$$\frac{\texttt{c}_1<:\texttt{c}_2 \quad \alpha_1 t_1 <: \alpha_2 t_2}{\texttt{c}_1\langle\alpha_1 t_1\rangle <: \texttt{c}_2\langle\alpha_2 t_2\rangle} \qquad \frac{(\alpha_1 <: \odot) \quad t_1 = t_2}{\alpha_1 t_1 <: \odot t_2} \qquad \frac{}{\alpha_1 t_1 <: \circledast t_2}$$

$$\frac{(\alpha_1 <: \oplus \ \wedge t_1 <: t_2) \vee t_2 = \texttt{Object}}{\alpha_1 t_1 <: \oplus t_2} \qquad \frac{(\alpha_1 <: \ominus \ \wedge t_2 <: t_1) \vee t_2 = \bot}{\alpha_1 t_1 <: \ominus t_2}$$

The bottom of the class hierarchy is $\bot$ denoting the type of `null` value, while the top of the class hierarchy is `Object`. For simplicity, the first rule assumes that each class constructor has only a single inheritable type parameter. The above rules use nominal subtyping $\texttt{c}_1<:\texttt{c}_2$ from traditional class hierarchy and also a reflexive and transitive variance subtyping with a simple hierarchy: $\odot <: \oplus <: \circledast \quad \odot <: \ominus <: \circledast$. The $<:$ operator is overloaded to handle variance subtyping, nominal class subtyping and two VPT subtypings for `t` and α`t`, respectively. The above subsumption relations form the basis of the VPT system to provide a richer subtyping system. Two provisos highlighted in the above rules for parametric fields are (i) to allow each such field to be retrievable as an `Object`, and (ii) a null value (of $\bot$ type) to be written into any such field, regardless of its variant annotation. Types $\oplus$`Object` and $\ominus\bot$ are essentially equivalent to $\circledast$`t`.

Motivation. Although VPT mechanisms have now been validated in the full-scale implementation of Java 5 [12] and Java library classes have been successfully refactored to use variant parametric types, these mechanisms are often criticized, due to the difficulty of choosing appropriate variance annotations. By annotating variance properties on each type argument to a parametric class, programmers can choose various desirable variance properties for each use of the parametric class. For example, the types $\texttt{Pair}\langle\oplus\texttt{A}, \oplus\texttt{B}\rangle$ or $\texttt{Pair}\langle\odot\texttt{A}, \odot\texttt{B}\rangle$ are still correct types for the receiver of the above method `getFst`. However the best generic type is $\texttt{Pair}\langle\oplus\texttt{A}, \circledast\rangle$, since the first field is read and the second field is not accessed. In order to establish the most flexible correct variance annotations (those which do not restrict the code genericity) for a type declaration, the programmer has to analyse all the places where that type declaration is used in the program. Although several algorithms have been proposed for refactoring legacy Java code [9, 8, 7, 11], they are restricted either to parametric types [1] or to variant parametric types with known variance annotations. No one can support the full flexibility of the use-site variance-based subtyping. Moreover these algorithms require global analysis.

Contributions. We propose a novel approach to automatically inferring the variance annotations and the type variables for the variant parametric types of method parameters (including receiver), method result and method body's local variables. In addition, the expected value flow that may arise from the method body is captured as a precondition. The inference is designed as *a summary-based analysis* that works on a per method basis: the variant parametric types of a method are inferred only based on how they are used in the method body, while each call site is a specific instance of the method's type declaration. Our inference is guided by a dependency graph such that all the methods which are called by the current method have been already analyzed. Our inference also assumes that the generic class hierarchy is known. In order to support the full flexibility of the subtyping based on the use-site variance, our inference algorithm starts with unknown variance annotations. Each variant parametric type is represented as an *interval type* [2], namely two type bounds that allow us to distinguish a *read flow* from a *write flow* for each object's field. Based on a flow-based approach for VPTs [4], we reduce the problem of inferring variance annotations and type arguments to the problem of solving specialized *flow constraints*. To the best of our knowledge this is *the first algorithm* that decouples variance inference from the type inference itself. In order to allow more generic types for the method parameters we introduce *dual types* to support *unknown variance flow*. Dual types make a distinction between flow via an object, *object flow* and the flow via the object's fields, *field flow*. We also use *intersection and union types* to capture the *divergent flow* and *convergent flow*, respectively. A safe yet precise approximation is used to avoid disjunctive constraints. We also provide special solutions to handle runtime cast operations and method overriding.

Related Work. The task of introducing generics to an existing Java code [9, 8, 7, 11, 16] consists of two distinct problems, parameterization and instantiation. Class parameterization selects the class fields that can be promoted as class type parameters. Since class parameterization decisions may be quite hard to automate due to trade-offs in the possible design outcomes, our solution is to let programmers focus on high-level design decisions for parameterization, while leaving the more tedious annotations on value flows of methods to be automatically inferred. Previous algorithms for instantiation have been restricted to parametric types based on invariant subtyping [9, 8, 7, 11]. Although the most recent Java refactoring paper [16] claims being able to infer wildcard types, it conservatively assumes invariant subtyping even with wildcard types.

At each call site, Java compiler [12] performs a local inference of the method's type parameters. The algorithm follows the local type inference designed for parametric types [17] . Recently, a significant revision of Java local inference has been proposed in [21]. The new proposal has introduced two bounds for a type variable similar to our interval types. However it does not perform variance inference since the variance annotations are known. Our approach is more general and subsuming the local type inference.

Our variant parametric type inference algorithm produces subtyping (flow) constraints. To solve them, we work on a closed constraint graph employing techniques from [25, 18, 22, 10]. It seems also possible to formalize our constraint solver on a pretransitive graph [13] to have a more scalable implementation. In general the constraint solving techniques assume that the polarities of term constructors are known. However the inference of variant parametric types may generate term constructors with un-

known polarities (variances). Therefore our approach uses an interval type (a contravariant lower bound and a covariant upper bound) to represent each unknown polarity of a term constructor. The idea of using interval types for updatable values has already been applied to reference type [20, 19] and also in the context of object calculi [2]. An open problem (discussed in [2]) is whether the interval types can be used to infer types with variance information from non-annotated terms. Our variance inference provides a constraint-based solution to this open problem.

Outline. The following section presents our interval-based view of VPTs. Section 3 introduces the key features of our approach. Section 4 formalizes our inference algorithm. Section 5 solves the method overriding problem. A brief conclusion is then given.

2 Variant Parametric Types as Interval Types

The underlying idea behind our solution is to view each variant parametric type $\alpha\mathtt{X}$ as an interval (of types) with a low-bound $\mathtt{X.L}$ and a high-bound $\mathtt{X.H}$ such that $\mathtt{X.L}<:\mathtt{X.H}$. The low-bound variable captures each value of type $\mathtt{t}_1$ that may *flow into* $\alpha\mathtt{X}$ using the constraint $\mathtt{t}_1<:\mathtt{X.L}$, while the high-bound variable captures each value of type $\mathtt{t}_2$ that may *flow out* of $\alpha\mathtt{X}$ using $\mathtt{X.H}<:\mathtt{t}_2$. By default, it is always safe for each low-bound $\mathtt{X.L}$ to be bounded by $\bot <:\mathtt{X.L}$ and each high-bound can be bounded by $\mathtt{X.H}<:\mathit{Object}$. For example, given a variant parametric type $\mathtt{c}\langle\alpha\mathtt{X}\rangle$ (where $\mathtt{X}$ is a type variable) denoting a class with a field of type $\alpha\mathtt{X}$, it can always be translated into an interval type as follows:

$$\frac{\mathtt{X}=\mathtt{X.H}}{\mathtt{c}\langle\oplus\mathtt{X}\rangle\Longleftrightarrow\mathtt{c}\langle[\bot,\mathtt{X.H}]\rangle} \quad \frac{\mathtt{X}=\mathtt{X.L}}{\mathtt{c}\langle\ominus\mathtt{X}\rangle\Longleftrightarrow\mathtt{c}\langle[\mathtt{X.L},\mathit{Object}]\rangle} \quad \frac{}{\mathtt{c}\langle\odot\mathtt{X}\rangle\Longleftrightarrow\mathtt{c}\langle[\mathtt{X},\mathtt{X}]\rangle}$$

$$\frac{}{\mathtt{c}\langle\circledast\rangle\Longleftrightarrow\mathtt{c}\langle[\bot,\mathit{Object}]\rangle} \quad \frac{\mathtt{X.L}=\mathtt{fresh}() \quad \mathtt{X.H}=\mathtt{fresh}()}{\mathtt{c}\langle\alpha\mathtt{X}\rangle\Longrightarrow\mathtt{c}\langle[\mathtt{X.L},\mathtt{X.H}]\rangle}$$

Translation rules are bidirectional where the variance is known. The last rule is a key rule for variance inference, as it splits a type variable with an unknown variance into two type variables. Thus, field selection (reading) uses the type $\mathtt{X.H}$, while field updating (writing) is based on type $\mathtt{X.L}$.

The *interval type subtyping* subsumes VPT subtyping and is defined as a contravariant subtyping on low-bounds and a covariant subtyping on high-bounds, as follows:

$$\frac{\mathtt{c}_1<:\mathtt{c}_2 \quad \mathtt{t}_2.\mathtt{L}<:\mathtt{t}_1.\mathtt{L} \quad \mathtt{t}_1.\mathtt{H}<:\mathtt{t}_2.\mathtt{H}}{\mathtt{c}_1\langle[\mathtt{t}_1.\mathtt{L},\mathtt{t}_1.\mathtt{H}]\rangle<:\mathtt{c}_2\langle[\mathtt{t}_2.\mathtt{L},\mathtt{t}_2.\mathtt{H}]\rangle}$$

The annotations $.\mathtt{L}$ and $.\mathtt{H}$ make a flow-based distinction among the types, such that:

- $\mathtt{X.L}$ denotes a type that expects a *write flow* (flow in),
- $\mathtt{X.H}$ denotes a type that expects a *read flow* (flow out),
- $\mathtt{X}$ (without annotation) denotes a type that expects both *read* and *write* flows.

Using the flow expectations, we identified a special group of flow constraints that we called *closed flow constraints*. They denote a matching of a flow-out with a flow-in, namely a consumption of a read flow by a write flow.

Definition 1 (Closed Flow Constraint). *A closed flow constraint is a flow constraint that has one of the following forms:* $\mathtt{X}_1.\mathtt{H}<:\mathtt{X}_2.\mathtt{L}$, $\mathtt{X}_1.\mathtt{H}<:\mathtt{X}_2$, $\mathtt{X}_1<:\mathtt{X}_2.\mathtt{L}$, *and* $\mathtt{X}_1<:\mathtt{X}_2$, *where* $\mathtt{X}_1, \mathtt{X}_2$, *are different from* Object *and* $\bot$.

Proposition 1 (Variance Inference Rule-1). *If a low-bound type variable* $\mathtt{X.L}$ *does not occur in any closed flow constraint, it is resolved to be* $\bot$. *If a high-bound type variable* $\mathtt{X.H}$ *does not occur in any closed flow constraint, it is resolved to be* Object.

3 Inference of Variant Parametric Types

3.1 Main Algorithm

This section illustrates the main steps of our inference algorithm using the following method of a non-generic `Pair` class:

`Pair` | `Object move(Pair a) { Object y=a.getFst(); this.setSnd(y); return y; }`

Our goal is to infer its generic version that corresponds to the variant parametric class $\texttt{Pair}\langle\texttt{A},\texttt{B}\rangle$. Internally, our algorithm works with interval types to generate and solve the flow constraints. Therefore, we use the following interval type based specifications of the methods `getFst` and `setSnd` of the variant parametric class $\texttt{Pair}\langle\texttt{A},\texttt{B}\rangle$:

$\texttt{Pair}\langle[\bot,\texttt{A.H}],[\bot,\textit{Object}]\rangle$ | `C getFst`$\langle$`A.H, C`$\rangle$`() where A.H<:C {..}`
$\texttt{Pair}\langle[\bot,\textit{Object}],[\texttt{B.L},\textit{Object}]\rangle$ | `void setSnd`$\langle$`B.L, C`$\rangle$`(C y) where C<:B.L {..}`

Step 0. Decoration with Fresh Interval Types. This is a pre-processing step. It consists of the annotation with fresh type variables of the non-generic types and non-generic methods. We use the following naming conventions: the letters $\texttt{V}_\texttt{i}$ for the global type variables (visible outside the method), the letter `Y` for the method result, the letters $\texttt{N}_\texttt{i}$ for the arguments of new expressions, and the letters $\texttt{T}_\texttt{i}$ for other annotations:

$\texttt{Pair}\langle[\texttt{V}_1\texttt{.L},\texttt{V}_1\texttt{.H}],[\texttt{V}_2\texttt{.L},\texttt{V}_2\texttt{.H}]\rangle$ | `Y move(`$\texttt{Pair}\langle[\texttt{V}_3\texttt{.L},\texttt{V}_3\texttt{.H}],[\texttt{V}_4\texttt{.L},\texttt{V}_4\texttt{.H}]\rangle$ `a)`
$\{\texttt{T}_0$ `y=a.getFst`$\langle\texttt{T}_1\texttt{.H},\texttt{T}_2\rangle$`(); this.setSnd`$\langle\texttt{T}_3\texttt{.L},\texttt{T}_4\rangle$`(y); return y; }`

Step 1. Collect Flow Constraints. This step gathers the constraints from the method body using the type inference rules given in Section 4.1, as follows:

$\texttt{Pair}\langle[\texttt{V}_3\texttt{.L},\texttt{V}_3\texttt{.H}],[\texttt{V}_4\texttt{.L},\texttt{V}_4\texttt{.H}]\rangle<:\texttt{Pair}\langle[\bot,\texttt{T}_1\texttt{.H}],[\bot,\textit{Object}]\rangle\wedge\texttt{T}_1\texttt{.H}<:\texttt{T}_2\wedge\texttt{T}_2<:\texttt{T}_0\wedge\texttt{T}_0<:\texttt{T}_4\wedge$
$\texttt{T}_4<:\texttt{T}_3\texttt{.L}\wedge\texttt{Pair}\langle[\texttt{V}_1\texttt{.L},\texttt{V}_1\texttt{.H}],[\texttt{V}_2\texttt{.L},\texttt{V}_2\texttt{.H}]\rangle<:\texttt{Pair}\langle[\bot,\textit{Object}],[\texttt{T}_3\texttt{.L},\textit{Object}]\rangle\wedge\texttt{T}_0<:\texttt{Y}$

Step 2. Simplify Flow Constraints. This is a closure algorithm that iteratively decomposes the constraints into their elementary components. It primarily applies the interval subtyping rules with transitivity. The closure algorithm is invoked each time a new constraint is added to the set. For brevity, in the following examples, we omit the transitivity and the default constraints like $\bot$<:`X`, `X`<:*Object*, and `X.L`<:`X.H`. The result of this step is the following:

$$\texttt{V}_3\texttt{.H}<:\texttt{T}_1\texttt{.H}\wedge\texttt{T}_1\texttt{.H}<:\texttt{T}_2\wedge\texttt{T}_2<:\texttt{T}_0\wedge\texttt{T}_0<:\texttt{T}_4\wedge\texttt{T}_4<:\texttt{T}_3\texttt{.L}\wedge\texttt{T}_3\texttt{.L}<:\texttt{V}_2\texttt{.L}\wedge\texttt{T}_0<:\texttt{Y}$$

Step 3. Variance Inference. This step generates a set of closed flow constraints and then applies the variance inference rule from Section 2. Since $\texttt{V}_1\texttt{.L}$, $\texttt{V}_1\texttt{.H}$, $\texttt{V}_4\texttt{.L}$, $\texttt{V}_4\texttt{.H}$, $\texttt{V}_2\texttt{.H}$, $\texttt{V}_3\texttt{.L}$ do not occur in any closed flow constraint, they are accordingly solved as follows:

$$\texttt{V}_1\texttt{.L}=\bot\ \wedge\ \texttt{V}_1\texttt{.H}=\textit{Object}\ \wedge\ \texttt{V}_4\texttt{.L}=\bot\ \wedge\ \texttt{V}_4\texttt{.H}=\textit{Object}\ \wedge\ \texttt{V}_2\texttt{.H}=\textit{Object}\ \wedge\ \texttt{V}_3\texttt{.L}=\bot$$

Step 4. Type Variables Inference. This step solves the type variables $\texttt{T}_\texttt{i}$, $\texttt{N}_\texttt{i}$, and `Y` in term of the global type variables $\texttt{V}_\texttt{i}$ and ground types (which are types without type variables). It consists of three substeps:

1. Cycle elimination: This causes all type variables of a cycle to be equal. Note that there isn't a cycle in the current example.
2. Ordering: The type variables are ordered based on the number of constraints in which they appear as an upper bound.
3. Unification: Following the order defined before, the type variables are solved by equating to their low bounds. Type variables occuring in fewer constraints have a higher priority.

For our example, the result of the unification is summarized by the last column of the following table. The first column contains the constraints in which the type variables from the second column occur as upper bounds. Multiple type variables in the second column denotes type variables having the same priority.

Constraints	TVars	Result
$V_3.H<:T_1.H$	$\{T_1.H\}$	$T_1.H=V_3.H$
$V_3.H<:T_2 \wedge T_1.H<:T_2$	$\{T_2\}$	$T_2=V_3.H$
$V_3.H<:T_0 \wedge T_1.H<:T_0 \wedge T_2<:T_0$	$\{T_0\}$	$T_0=V_3.H$
$V_3.H<:T_4 \wedge V_3.H<:Y \wedge T_1.H<:T_4 \wedge T_1.H<:Y \wedge T_2<:T_4 \wedge T_2<:Y \wedge T_0<:T_4 \wedge T_0<:Y$	$\{T_4, Y\}$	$Y=T_4=V_3.H$
$V_3.H<:T_1.H<:T_2<:T_0<:T_4<:T_3.L$	$\{T_3.L\}$	$T_3.L=V_3.H$

Step 5. Result Refining. This step simplifies the inferred types of the method. The goal is to reduce the number of the global type variables using the residual flow constraint (namely the remaining flow constraints among the global type variables). The residual flow constraint of the current example is: $V_3.H<:V_2.L$. These type variables can be unified to a fresh type variable V, such that $V=V_3.H=V_2.L$. Since V stands for both low-bound and high-bound, it is not marked with either. The result of our inference (including the above refinements) is the following:

```
Pair⟨[⊥, Object],[V, Object]⟩ | V move⟨V⟩(Pair⟨[⊥, V],[⊥, Object]⟩ a)
                {V y=a.getFst⟨V, V⟩(); this.setSnd⟨V, V⟩(y); return y; }
```

Step 6. VPT Result. This step translates the inferred interval types into VPTs:

```
Pair⟨⊛,⊖V⟩ | V move⟨V⟩(Pair⟨⊕V,⊛⟩ a)
                {V y=a.getFst⟨V, V⟩(); this.setSnd⟨V, V⟩(y); return y; }
```

3.2 Interval Types versus Variant Parametric Types

The interval types are more expressive than variant parametric types, since they can support two different non-default bounds. A variant parametric type can only support two equal non-default bounds in the case of invariant subtyping $\odot$. Note that the default low-bound is $\perp$, while the default high-bound is *Object*. Considering the following code fragment, we like to infer the interval type of obj:

```
class Cell⟨A⟩ { A fst; ···                  class Integer extends Number{..}
    Cell⟨⊕A⟩ | A get⟨A⟩(){..}               class MyInt extends Integer{..}
    Cell⟨⊖A⟩ | void set⟨A⟩(A y){..}..}
        ...
Cell⟨[T.L, T.H]⟩ obj = new Cell⟨Integer⟩(new Integer(1)); // T.L<:Integer<:T.H
Number n = obj.get⟨T1⟩();                                 // T.H<:T1<:Number
MyInt m = new MyInt(2); obj.set⟨T2⟩(m);                   // MyInt<:T2<:T.L
```

Our algorithm can infer the interval type $\texttt{Cell}\langle[\texttt{MyInt}, \texttt{Number}]\rangle$ for obj. However this interval type cannot be translated into a variant parametric type, since it consists of two different bounds. In order to keep the equivalence between interval types and variant parametric types, we add one more rule to the variance inference:

Proposition 2 (Variance Inference Rule-2). *If both bounds* X.L *and* X.H *of an interval type occur in the closed flow constraints, then the default constraint of an interval type* X.L<:X.H *is strengthened to the equality* X.L=X.H.

In our example, adding T.L=T.H to the above set of constraints will generate a cycle such that T.L<:Integer<:T.H ∧ T.L=T.H. Cycle elimination generates T.L=Integer=T.H. Thus new inference result is the interval type $\texttt{Cell}\langle[\texttt{Integer}, \texttt{Integer}]\rangle$, that can be directly translated into the variant parametric type $\texttt{Cell}\langle\odot\texttt{Integer}\rangle$.

3.3 Main Flow and Conditional Flow

Cast operations give rise to *conditional flow constraints* (or dynamic subtype constraints in [9]). These constraints are conditional in the sense that they are only required to hold if the corresponding dynamic downcasts succeed at runtime. Our analysis separates the *main flow* gathered from the method body without the cast operations and the *conditional flow* corresponding to the cast operations. Conditional constraints use a different subtyping notation ($<:_c$). One benefit of our analysis is that it can guarantee that some of the cast operations are redundant, and therefore they can be safely eliminated at compile time. The number of the eliminated casts is used as an accuracy measure of generic type systems [8, 11, 4, 16]. The following example illustrates how our inference algorithm handles the cast operations:

Original code	Inference Result
$\texttt{Cell} \mid \texttt{void fill(Cell a)}$	$\texttt{Cell}\langle\oplus\texttt{V}\rangle \mid \texttt{void fill(Cell}\langle\oplus\texttt{Cell}\langle\ominus\texttt{V}\rangle\rangle\ \texttt{a)}$
$\{\texttt{Cell b} = (\texttt{Cell})\texttt{a.fst;b.fst} = \texttt{this.fst;}\}$	$\{\texttt{Cell}\langle\ominus\texttt{V}\rangle\ \texttt{b} = \texttt{a.fst;b.fst} = \texttt{this.fst;}\}$

Code annotated with Fresh Interval Types

$\texttt{Cell}\langle[\texttt{V}_1\texttt{.L}, \texttt{V}_1\texttt{.H}]\rangle \mid \texttt{void fill(Cell}\langle[\texttt{V}_2\texttt{.L}, \texttt{V}_2\texttt{.H}]\rangle\ \texttt{a)}$
$\{\ \texttt{Cell}\langle[\texttt{T}_1\texttt{.L}, \texttt{T}_1\texttt{.H}]\rangle\ \texttt{b} = (\texttt{Cell}\langle[\texttt{T}_2\texttt{.L}, \texttt{T}_2\texttt{.H}]\rangle)\texttt{a.fst;}\ \ \texttt{b.fst} = \texttt{this.fst;}\ \}$

1. Collect Constraints	$\texttt{Cell}\langle[\texttt{T}_2\texttt{.L}, \texttt{T}_2\texttt{.H}]\rangle <: \texttt{Cell}\langle[\texttt{T}_1\texttt{.L}, \texttt{T}_1\texttt{.H}]\rangle \wedge \texttt{V}_1\texttt{.H} <: \texttt{T}_1\texttt{.L}$ $\texttt{V}_2\texttt{.H} <:_c \texttt{Cell}\langle[\texttt{T}_2\texttt{.L}, \texttt{T}_2\texttt{.H}]\rangle$
2. Simplify	$\texttt{V}_1\texttt{.H} <: \texttt{T}_1\texttt{.L} <: \texttt{T}_2\texttt{.L} <: \texttt{T}_2\texttt{.H} <: \texttt{T}_1\texttt{.H} \quad \texttt{V}_2\texttt{.H} <:_c \texttt{Cell}\langle[\texttt{T}_2\texttt{.L}, \texttt{T}_2\texttt{.H}]\rangle$
3. Infer Variance	$\texttt{V}_1\texttt{.L} = \texttt{V}_2\texttt{.L} = \bot \wedge \texttt{T}_1\texttt{.H} = \texttt{T}_2\texttt{.H} = \textit{Object} \wedge \texttt{V}_1\texttt{.H} <: \texttt{T}_1\texttt{.L} <: \texttt{T}_2\texttt{.L}$ $\texttt{V}_2\texttt{.H} <:_c \texttt{Cell}\langle[\texttt{T}_2\texttt{.L}, \textit{Object}]\rangle$
4. Infer Type Vars	$\{\texttt{T}_1\texttt{.L}\} \quad \texttt{T}_1\texttt{.L} = \texttt{V}_1\texttt{.H}$ $\{\texttt{T}_2\texttt{.L}\} \quad \texttt{T}_2\texttt{.L} = \texttt{V}_1\texttt{.H}$
5. Solve Conditional	$\texttt{V}_2\texttt{.H} <:_c \texttt{Cell}\langle[\texttt{V}_1\texttt{.H}, \textit{Object}]\rangle \Rightarrow \texttt{V}_2\texttt{.H} <: \texttt{Cell}\langle[\texttt{V}_1\texttt{.H}, \textit{Object}]\rangle$
6. Refine Results	$\texttt{V} = \texttt{fresh}() \quad \texttt{V}_1\texttt{.H} = \texttt{V} \wedge \texttt{V}_2\texttt{.H} = \texttt{Cell}\langle[\texttt{V}, \textit{Object}]\rangle$

Though the conditional flow is kept separately, it is still used by the variance inference in Step 3. If Step 3 ignores the conditional flow, it infers the incorrect result $\texttt{V}_2\texttt{.H} = \textit{Object}$. A new step (Step 5) is added to the main algorithm. This step combines together the conditional flow and the (already solved) main flow in order to find a common solution. In our example, adding the conditional constraint to the main flow does not generate any contradiction as the type variables $\texttt{V}_2\texttt{.H}$ and $\texttt{V}_1\texttt{.H}$ are unconstrained in the main flow. However it is not always possible to find a common solution for the main and conditional flow, as illustrated by the following example:

$\texttt{V}_3\ \texttt{foo2(Cell}\langle[\texttt{V}_4\texttt{.L}, \texttt{V}_4\texttt{.H}]\rangle\ \texttt{obj)}\ \{$
$\texttt{if(...)}\ \{\texttt{... return (Integer)obj.fst;}\}\ \texttt{else}\{\texttt{... return (Float)obj.fst;}\}\}$
$//\texttt{Integer} <: \texttt{V}_3 \wedge \texttt{Float} <: \texttt{V}_3 \quad \texttt{V}_4\texttt{.H} <:_c \texttt{Integer} \wedge \texttt{V}_4\texttt{.H} <:_c \texttt{Float}$

In this example the conditional constraints can be added to the method precondition to be checked at each call site where the casts could be selectively eliminated (with the help of a polyvariant program specializer):

$\texttt{Number foo2(Cell}\langle[\bot, \texttt{V}_4\texttt{.H}]\rangle\ \texttt{obj)}\ \texttt{where}\ \texttt{V}_4\texttt{.H} <:_c \texttt{Integer} \wedge \texttt{V}_4\texttt{.H} <:_c \texttt{Float}$

3.4 Convergent Flow and Divergent Flow

Multiple low bounds denote a *convergent flow*, while multiple high bounds denote a *divergent flow*. Our analysis uses union types for multiple low bounds and intersection types for multiple high bounds. An union type $\texttt{t}_1|\texttt{t}_2$ represents the least upper bound

of t_1 and t_2, while an intersection type $t_1\&t_2$ is the greatest lower bound of t_1 and t_2. Some of their subtyping rules may generate disjunctions. In order to keep our analysis simple, we propose a safe yet precise approximation that avoids those disjunctions:

AND rules	OR rules	Our OR rules
$\frac{t_1 \mid t_2 <: t}{t_1 <: t \ \wedge\ t_2 <: t}$	$\frac{t <: t_1 \mid t_2}{t <: t_1 \ \vee\ t <: t_2}$	$\frac{t <: t_1 \mid t_2 \quad T_1 = \texttt{fresh}()}{t <: T_1 \wedge t_1 <: T_1 \wedge t_2 <: T_1}$
$\frac{t <: t_1 \& t_2}{t <: t_1 \ \wedge\ t <: t_2}$	$\frac{t_1 \& t_2 <: t}{t_1 <: t \ \vee\ t_2 <: t}$	$\frac{t_1 \& t_2 <: t \quad T_2 = \texttt{fresh}()}{T_2 <: t_1 \wedge T_2 <: t_2 \wedge T_2 <: t}$

where T_1 and T_2 are fresh type variables. Another solution to avoid disjunctions is the tautology $t_1\&t_2 <: t_1 \mid t_2$, but sometimes this approximation may lead to no solutions. One benefit of using union and intersection types is that they are more expressive so that more casts can be directly eliminated as the following example (from [8, 4]) can illustrate:

```
class B1 extends A implements I {..}; class B2 extends A implements I {..};
```

Original code	Code annotated with Fresh Interval Types
`void foo(Boolean b){`	`void foo(Boolean b){`
`Cell c1 = new Cell(new B1());`	`Cell⟨[T1.L, T1.H]⟩ c1 = new Cell⟨N1⟩(new B1());`
`Cell c2 = new Cell(new B2());`	`Cell⟨[T2.L, T2.H]⟩ c2 = new Cell⟨N2⟩(new B2());`
`Cell c = b?c1 : c2;`	`Cell⟨[T3.L, T3.H]⟩ c = b?c1 : c2;`
`A a = (A) c.get();`	`A a = (A) c.get⟨T4⟩();`
`I i = (I) c.get();`	`I i = (I) c.get⟨T5⟩();`
`B1 b1 = (B1) c1.get();`	`B1 b1 = (B1) c1.get⟨T6⟩();`
`B2 b2 = (B2) c2.get(); }`	`B2 b2 = (B2) c2.get⟨T7⟩(); }`

The following table contains the inference steps for the above code with interval types. At the step 4.4, $T_3.H$ is resolved as to the union type $B1|B2$ due to two distinct flows converging to it, $B1 <: T_3.H \wedge B2 <: T_3.H$. The solutions of the main flow can prove that all conditional constraints succeed, and therefore all casts can be eliminated.

1. Collect Constraints	$B1 <: N_1 \wedge \texttt{Cell}\langle[N_1, N_1]\rangle <: \texttt{Cell}\langle[T_1.L, T_1.H]\rangle \wedge$ $B2 <: N_2 \wedge \texttt{Cell}\langle[N_2, N_2]\rangle <: \texttt{Cell}\langle[T_2.L, T_2.H]\rangle \wedge$ $\texttt{Cell}\langle[T_1.L, T_1.H]\rangle <: \texttt{Cell}\langle[T_3.L, T_3.H]\rangle \wedge \texttt{Cell}\langle[T_2.L, T_2.H]\rangle <: \texttt{Cell}\langle[T_3.L, T_3.H]\rangle$ $\wedge T_3.H <: T_4 \wedge T_3.H <: T_5 \wedge T_1.H <: T_6 \wedge T_2.H <: T_7$ $T_4 <:_c A \wedge T_5 <:_c I \wedge T_6 <:_c B1 \wedge T_7 <:_c B2$
2. Simplify	$B1 <: N_1 \wedge T_1.L <: N_1 <: T_1.H \wedge B2 <: N_2 \wedge T_2.L <: N_2 <: T_2.H \wedge T_3.L <: T_1.L \wedge T_1.H <: T_3.H$ $\wedge T_3.L <: T_2.L \wedge T_2.H <: T_3.H \wedge T_3.H <: T_4 \wedge T_3.H <: T_5 \wedge T_1.H <: T_6 \wedge T_2.H <: T_7$ $T_4 <:_c A \wedge T_5 <:_c I \wedge T_6 <:_c B1 \wedge T_7 <:_c B2$
3. Infer Variance	$T_1.L = T_2.L = T_3.L = \bot \wedge B1 <: N_1 <: T_1.H \wedge B2 <: N_2 <: T_2.H \wedge$ $T_1.H <: T_3.H \wedge T_2.H <: T_3.H \wedge T_3.H <: T_4 \wedge T_3.H <: T_5 \wedge T_1.H <: T_6 \wedge T_2.H <: T_7$ $T_4 <:_c A \wedge T_5 <:_c I \wedge T_6 <:_c B1 \wedge T_7 <:_c B2$
4. Infer Type Vars	$\{N_1, N_2\}$ $N_1 = B1 \wedge N_2 = B2$ $\{T_1.H, T_2.H\}$ $T_1.H = B1 \wedge T_2.H = B2$ $\{T_6, T_7\}$ $T_6 = B1 \wedge T_7 = B2$ $\{T_3.H\}$ $T_3.H = B1 \mid B2$ $\{T_4, T_5\}$ $T_5 = T_4 = B1 \mid B2$
5. Solve Conditional	$B_1 \mid B_2 <:_c A \quad B_1 \mid B_2 <:_c I$ $B1 <:_c B1 \quad B2 <:_c B2$

3.5 Field Flow and Object Flow

A key feature of our approach is the distinction between the flow via an object, called *object flow* and the flow via the fields of that object, called *field flow*. We introduce a special type notation, that we called *dual type* to support these two views: (1) object as a black box, and (2) object as a glass box. For example, a dual type for a `Pair` is of the form $\mathtt{X} \doteq \mathtt{Pair}\langle[\mathtt{V}_1.\mathtt{L}, \mathtt{V}_1.\mathtt{H}], [\mathtt{V}_2.\mathtt{L}, \mathtt{V}_2.\mathtt{H}]\rangle$, where the type variable `X` (called *object part*) is used for the flow of the entire object, while $\mathtt{Pair}\langle[\mathtt{V}_1.\mathtt{L}, \mathtt{V}_1.\mathtt{H}], [\mathtt{V}_2.\mathtt{L}, \mathtt{V}_2.\mathtt{H}]\rangle$ (called *field part*) caters to the flow via its fields. This dualism can improve the genericity of our inference results. Specifically, given the following method `dup` (from [4, 14]):

$$\mathtt{Pair\ dup(Pair\ a)\ \{\ Pair\ p = new\ Pair(a, a);\ return\ p;\ \}}$$

Without using the dual types, our inference can get the following types:

$$\mathtt{Pair}\langle\circledast,\circledast\rangle\ \mathtt{dup(Pair}\langle\circledast, \circledast\rangle\ \mathtt{a)\{}$$
$$\mathtt{Pair}\langle\circledast,\circledast\rangle\ \mathtt{p=new\ Pair}\langle\mathtt{Pair}\langle\circledast, \circledast\rangle,\mathtt{Pair}\langle\circledast,\circledast\rangle\rangle(\mathtt{a}, \mathtt{a});\ \mathtt{return\ p;\ \}}$$

The type of the method result is too imprecise, but still correct as fields are not accessed (bivariant $\circledast$) in the method body. Using dual types our approach can get more precise types by inferring an intersection type for the method parameter `a`, namely:

$$\mathtt{Pair}\langle\odot\mathtt{X}_1, \odot\mathtt{X}_1\rangle\ \mathtt{dup}\langle\mathtt{X}_1\rangle(\mathtt{X}_1\&\mathtt{Pair}\langle\circledast, \circledast\rangle\ \mathtt{a})\ \{$$
$$\mathtt{Pair}\langle\odot\mathtt{X}_1, \odot\mathtt{X}_1\rangle\ \ \mathtt{p} = \mathtt{new\ Pair}\langle\mathtt{X}_1, \mathtt{X}_1\rangle(\mathtt{a}, \mathtt{a});\ \mathtt{return\ p;\ \}}$$

The type variable $\mathtt{X}_1$ plays an important role, it allows the unknown variance to flow unchanged, such that the variance annotations of the parameter `a` fields are preserved in the type of the method result. As can be seen below, the type variable $\mathtt{X}_1$ comes from the object part of the dual type:

$$\mathtt{Y\ dup}(\mathtt{X}_1 \doteq \mathtt{Pair}\langle[\mathtt{V}_1.\mathtt{L}, \mathtt{V}_1.\mathtt{H}], [\mathtt{V}_2.\mathtt{L}, \mathtt{V}_2.\mathtt{H}]\rangle\ \mathtt{a})\ \{$$
$$\mathtt{X}_2 \doteq \mathtt{Pair}\langle[\mathtt{T}_1.\mathtt{L}, \mathtt{T}_1.\mathtt{H}], [\mathtt{T}_2.\mathtt{L}, \mathtt{T}_2.\mathtt{H}]\rangle\ \ \mathtt{p} = \mathtt{new\ Pair}\langle\mathtt{N}_1, \mathtt{N}_2\rangle(\mathtt{a}, \mathtt{a});\ \mathtt{return\ p;\ \}}$$

1. Collect Constraints	$\mathtt{X}_1 \doteq \mathtt{Pair}\langle[\mathtt{V}_1.\mathtt{L},\mathtt{V}_1.\mathtt{H}],[\mathtt{V}_2.\mathtt{L},\mathtt{V}_2.\mathtt{H}]\rangle <: \mathtt{N}_1 \wedge \mathtt{X}_1 \doteq \mathtt{Pair}\langle[\mathtt{V}_1.\mathtt{L},\mathtt{V}_1.\mathtt{H}],[\mathtt{V}_2.\mathtt{L},\mathtt{V}_2.\mathtt{H}]\rangle <: \mathtt{N}_2$ $\wedge \mathtt{Pair}\langle[\mathtt{N}_1,\mathtt{N}_1],[\mathtt{N}_2, \mathtt{N}_2]\rangle <: \mathtt{X}_2 \doteq \mathtt{Pair}\langle[\mathtt{T}_1.\mathtt{L},\mathtt{T}_1.\mathtt{H}],[\mathtt{T}_2.\mathtt{L},\mathtt{T}_2.\mathtt{H}]\rangle$ $\wedge \mathtt{X}_2 \doteq \mathtt{Pair}\langle[\mathtt{T}_1.\mathtt{L}, \mathtt{T}_1.\mathtt{H}], [\mathtt{T}_2.\mathtt{L}, \mathtt{T}_2.\mathtt{H}]\rangle <: \mathtt{Y}$
2. Simplify Dual Types	$\mathtt{X}_1 <: \mathtt{N}_1 \wedge \mathtt{X}_1 <: \mathtt{N}_2 \wedge \mathtt{Pair}\langle[\mathtt{N}_1, \mathtt{N}_1], [\mathtt{N}_2, \mathtt{N}_2]\rangle <: \mathtt{X}_2$ $\wedge \mathtt{Pair}\langle[\mathtt{N}_1, \mathtt{N}_1], [\mathtt{N}_2, \mathtt{N}_2]\rangle <: \mathtt{Pair}\langle[\mathtt{T}_1.\mathtt{L}, \mathtt{T}_1.\mathtt{H}], [\mathtt{T}_2.\mathtt{L}, \mathtt{T}_2.\mathtt{H}]\rangle \wedge \mathtt{X}_2 <: \mathtt{Y}$
3. Simplify	$\mathtt{X}_1 <: \mathtt{N}_1 \wedge \mathtt{X}_1 <: \mathtt{N}_2 \wedge \mathtt{Pair}\langle[\mathtt{N}_1, \mathtt{N}_1], [\mathtt{N}_2, \mathtt{N}_2]\rangle <: \mathtt{X}_2 <: \mathtt{Y}$ $\wedge \mathtt{T}_1.\mathtt{L} <: \mathtt{N}_1 <: \mathtt{T}_1.\mathtt{H} \wedge \mathtt{T}_2.\mathtt{L} <: \mathtt{N}_2 <: \mathtt{T}_2.\mathtt{H}$
4. Infer Variance	$\mathtt{V}_1.\mathtt{L} = \mathtt{V}_2.\mathtt{L} = \mathtt{T}_1.\mathtt{L} = \mathtt{T}_2.\mathtt{L} = \bot \wedge \mathtt{V}_1.\mathtt{H} = \mathtt{V}_2.\mathtt{H} = \mathtt{T}_1.\mathtt{H} = \mathtt{T}_2.\mathtt{H} = \mathit{Object}$ $\mathtt{X}_1 <: \mathtt{N}_1 \wedge \mathtt{X}_1 <: \mathtt{N}_2 \wedge \mathtt{Pair}\langle[\mathtt{N}_1, \mathtt{N}_1], [\mathtt{N}_2, \mathtt{N}_2]\rangle <: \mathtt{X}_2 <: \mathtt{Y}$
5. Infer Type Vars	$\{\mathtt{N}_1, \mathtt{N}_2\}\ \mathtt{N}_1 = \mathtt{X}_1 \wedge \mathtt{N}_2 = \mathtt{X}_1$ $\{\mathtt{X}_2\}\ \mathtt{X}_2 = \mathtt{Pair}\langle[\mathtt{X}_1, \mathtt{X}_1], [\mathtt{X}_1, \mathtt{X}_1]\rangle$ $\{\mathtt{Y}\}\ \mathtt{Y} = \mathtt{Pair}\langle[\mathtt{X}_1, \mathtt{X}_1], [\mathtt{X}_1, \mathtt{X}_1]\rangle$
6. Refine Results	$\mathtt{X}_1 \doteq \mathtt{Pair}\langle[\bot, \mathit{Object}], [\bot, \mathit{Object}]\rangle \Rightarrow \mathtt{X}_1\&\mathtt{Pair}\langle[\bot, \mathit{Object}], [\bot, \mathit{Object}]\rangle$ $\mathtt{Pair}\langle[\mathtt{X}_1,\mathtt{X}_1],[\mathtt{X}_1,\mathtt{X}_1]\rangle \doteq \mathtt{Pair}\langle[\bot,\mathit{Object}],[\bot,\mathit{Object}]\rangle \Rightarrow \mathtt{Pair}\langle[\mathtt{X}_1,\mathtt{X}_1],[\mathtt{X}_1,\mathtt{X}_1]\rangle$

A new step (`Step 2`) is added to the main algorithm in order to simplify the dual types. The simplification rules always prefer the object flow over the field flow (e.g. first constraint of `Step 1` is reduced to $\mathtt{X}_1 <: \mathtt{N}_1$). However, when the type variables of the field part are used by the other constraints, both flows are generated (e.g. the third constraint of `Step 1` is decomposed into two constraints). The last step is adapted to refine the dual types. A dual type can be refined to an intersection type (e.g. first line of `Step 6`). Since an intersection type is the greatest lower bound of its parts, it could be further simplified (e.g. the second line of `Step 6`).

4 Inference Algorithm

Program

P ::= *def**

def ::= `class` $c\langle K^*\rangle$ `extends` $c_1\langle\pi_{1_1},..,\pi_{1_k}\rangle$ ... $c_n\langle\pi_{n_1},..,\pi_{n_k}\rangle$ $\{(\pi\ f)^*$ *mth*$^*\}$

mth ::= $\kappa \mid \kappa$ *mn*$\langle V^*\rangle$**(**$(\kappa\ v)^*$**)** `where` ψ $\{e\}$

e ::= `null` $\mid v \mid v.f \mid v = e \mid v.f = e \mid \{\kappa\ v = e_1\ ; e_2\} \mid e_1\ ; e_2 \mid (\kappa)v \mid$ `new` $c\langle\kappa^*\rangle$**(**v^***)** $\mid$ `if` v `then` e_1 `else` $e_2 \mid$ `while` v `do` $e \mid v_0.mn\langle\kappa^*\rangle$**(**$v^*$**)**

Variant Parametric Type (VPT)

π ::= $\alpha\kappa \mid K$

κ ::= $V \mid c\langle\pi_1,..,\pi_n\rangle \mid c \mid \kappa\&\kappa \mid \kappa|\kappa \mid \bot$

α ::= $\odot \mid \oplus \mid \ominus \mid \circledast$

VPT Subtyping Constraint

ψ ::= $\kappa_1{<:}\kappa_2 \mid \kappa_1{<:_c}\kappa_2 \mid \psi\wedge\psi \mid$ `true`

Fig. 1. SYNTAX OF VARIANT CORE-JAVA

We design our inference algorithm as a *summary-based analysis*, on a per method basis guided by a global method call graph. Our approach is flow-insensitive within each method, but context-sensitive across the methods. The algorithm takes as input a well-typed non-generic program and the VPT class hierarchy, before it outputs a program that uses VPTs.

We use two assumptions to avoid recursive constraints: (1) no F-bounded quantification over the VPT class hierarchy, and (2) no polymorphic recursion for the classes and the methods. Techniques for avoiding recursive constraints are presented in [4, 5]. Nevertheless, our algorithm can cope with F-bounds, as long as we use constraint solving techniques that support recursive constraints and inductive simplification (from [25, 18]). Our current approach can infer generic types for mutually-recursive methods under the monomorphic recursion assumption.

We formalize the algorithm on Variant Core-Java (Fig. 1), a core calculus for Java-like languages. Both input and output programs are encoded in Core-Java since VPTs can subsume non-generic types. For ease of presentation, the features related to static methods, exception handling, inner classes and overloading are omitted. Multiple interface inheritance is supported as in Java [12], each class may extend from a single superclass but may implement multiple interfaces. VPT's syntax is also shown in Fig. 1. There are two kinds of type variables: `K` denoting a variance and a type together, and `V` denoting only a type. For simplicity, primitive types (e.g. `bool`, `void`) are represented by their corresponding classes (such as `Bool`, `Void`). Specifically, for each method our analysis can be divided into two main steps: (1) gathering the flow constraints based on the type inference rules (Section 4.1), and (2) solving the flow constraints (Section 4.2).

4.1 Type Inference Rules

The inference process is driven by the following main rule for each method:

$$\frac{\begin{array}{c} G\vdash c_i \Rrightarrow_{dcr} t_i \qquad G;\{(v_i{:}t_i)_{i=2}^n, \texttt{this}{:}t_1\}\vdash e\Rrightarrow_e e'{:}t,\varphi_0 \\ Y{=}\texttt{fresh}() \qquad Q_1{=}\bigcup_{i=1}^n \texttt{fv}(t_i) \qquad G\vdash \varphi_0\wedge t{<:}Y;Q_1 \Rrightarrow_{solver} \varphi;Q;\sigma \\ \sigma t_1|\sigma Y\ mn\langle Q\rangle\texttt{((}\sigma t_i\ v_i)_{i=2}^n\texttt{)}\ \texttt{where}\ \varphi\ \{\sigma e'\}\Rrightarrow_{vpt}\kappa_1|\kappa\ mn\langle Q'\rangle\texttt{((}(\kappa_i\ v_i)_{i=2}^n\texttt{)}\ \texttt{where}\ \psi\ \{e''\}\end{array}}{G\vdash c_1|c_0\ mn\texttt{((}(c_i\ v_i)_{i=2}^n\texttt{)}\ \{e\}\ \Rrightarrow\ \kappa_1|\kappa\ mn\langle Q'\rangle\texttt{((}(\kappa_i\ v_i)_{i=2}^n\texttt{))}\ \texttt{where}\ \psi\ \{e''\}}$$

that takes a non-generic method and the VPT class hierarchy `G`, decorates the method parameters ($\Rrightarrow_{dcr}$) with fresh interval types, collects the flow constraints ($\Rrightarrow_{e}$) from the method body, and then passes the constraints to the constraint solver ($\Rrightarrow_{solver}$). The solver

$$
\begin{array}{ll}
\multicolumn{2}{c}{\underline{\textbf{Interval Type}}} \\
\tau & ::= t \mid t.L \mid t.H \\
t & ::= X \mid d \mid c\langle[\tau_{1L},\tau_{1H}],..,[\tau_{nL},\tau_{nH}]\rangle \\
 & \quad c \mid t_1 \& t_2 \mid t_1 | t_2 \mid \bot \mid \mathit{Object} \\
X & ::= V \mid T \mid N \mid Y \\
\multicolumn{2}{c}{\underline{\textbf{Dual Type}}} \\
\multicolumn{2}{c}{d ::= X \doteq c\langle[\tau_{1L},\tau_{1H}],...,[\tau_{nL},\tau_{nH}]\rangle}
\end{array}
\qquad
\begin{array}{l}
\underline{\textbf{Flow Constraint}} \\
\varphi ::= \tau_1 <: \tau_2 \mid \tau_1 <:_c \tau_2 \mid \varphi \wedge \varphi \mid \texttt{true} \\
\underline{\textbf{Substitution}} \\
\sigma ::= X = \tau \mid X.L = \tau \mid X.H = \tau \mid d = t \\
\underline{\textbf{Closed Flow Constraint}} \\
\phi ::= t_1.H <: t_2.L \mid t_1 <: t_2.L \mid t_1.H <: t_2 \\
\quad \mid t_1.H <:_c t_2.L \mid t_1 <:_c t_2.L \mid t_1.H <:_c t_2
\end{array}
$$

Fig. 2. INFERENCE TYPES AND FLOW CONSTRAINTS

(Section 4.2) returns the list of method type parameters $\mathtt{Q}$ and a substitution σ. The substitution maps the type variables (introduced by the decoration) either to ground types or to the type variables from $\mathtt{Q}$. Interval types (and dual types), their flow constraints and the substitutions σ are detailed in Fig. 2. The final step $\Rrightarrow_{vpt}$ translates the interval types inferred for the method into VPTs. We summarize below the main judgments employed by this phase of our analysis (a complete description is in [5]):

- $\mathtt{G} \vdash \mathtt{c} \Rrightarrow_{dcr} \mathtt{t}$ denotes the decoration with fresh type variables of a non-generic class $\mathtt{c}$ with respect to its parameterized version from VPT class hierarchy $\mathtt{G}$. The result $\mathtt{t}$ is either a dual type, or the class $\mathtt{c}$ (when $\mathtt{c}$ is not parameterized), or a type variable.
- $\rho \vdash \kappa \Rrightarrow_{\kappa t} \mathtt{t}$ and $\rho \vdash \pi \Rrightarrow_{\pi \tau} \tau$ denote the translation of a VPT into an interval type with respect to a substitution ρ. A substitution ρ maps a type variable $\mathtt{K}$ (denoting a type and a variance) into two bounds $[\tau_{\mathtt{L}}, \tau_{\mathtt{H}}]$.
- $\mathtt{G} \vdash \mathtt{t}, \mathtt{fn} \Rrightarrow_{fld} [\tau_{\mathtt{L}}, \tau_{\mathtt{H}}]$ returns the low-bound and high-bound $[\tau_{\mathtt{L}}, \tau_{\mathtt{H}}]$ of a field $\mathtt{fn}$ with respect to an interval type $\mathtt{t}$ and the VPT class hierarchy $\mathtt{G}$.
- $\mathtt{G} \vdash \mathtt{t}, \mathtt{mn} \Rrightarrow_{mth} \mathtt{mth}$ returns the interface $\mathtt{mth}$ (with fresh interval types) of a method $\mathtt{mn}$ with respect to a receiver $\mathtt{t}$ and the VPT class hierarchy $\mathtt{G}$.
- $\mathtt{G}; \Gamma \vdash \mathtt{e} \Rrightarrow_{\mathtt{e}} \mathtt{e}' : \mathtt{t}, \varphi$ denotes the type inference for the expression $\mathtt{e}$ with respect to the type environment Γ and the VPT class hierarchy $\mathtt{G}$. The inference result consists of the expression $\mathtt{e}'$ annotated with interval types, its interval type $\mathtt{t}$ and the derived flow constraint φ. The type environment Γ consists of the interval types generated by $\Rrightarrow_{dcr}$.

4.2 Constraint Solver

The constraint solver takes as input a flow constraint φ_0, a set of visible type variables $\mathtt{Q}_0$, a VPT class hierarchy $\mathtt{G}$ and performs the following sequence of steps:

$$
\frac{\begin{array}{c}\vdash \varphi_0 \Rrightarrow_{set} C_0 \quad \vdash C_0 \Rrightarrow_{tr} C_1 \quad G \vdash C_1 \Rrightarrow_{simplify} C_2 \quad \vdash C_2 \Rrightarrow_{dual} C_3; D \quad G \vdash C_3 \Rrightarrow_{simplify} C_4 \\ \vdash C_4; Q_0 \Rrightarrow_{variance} C_5; Q_1; \sigma_1 \quad \vdash C_5; Q_1; \Rrightarrow_{typvar} C_6; Q_2; \sigma_2 \quad G \vdash C_6 \Rrightarrow_{cond} C_7 \\ G \vdash C_7; D; Q_2; \sigma_2 \circ \sigma_1 \Rrightarrow_{refine} C_8; Q; \sigma \quad \vdash C_8 \Rrightarrow_{cnj} \varphi\end{array}}{G \vdash \varphi_0; Q_0 \Rrightarrow_{solver} \varphi; Q; \sigma}
$$

The goal is first to simplify the constraints φ_0 to atomic constraints among type variables and ground types and then to solve the type variables in term of the ground type and the visible type variables $\mathtt{Q}$. The result consists of a residual constraint φ among the visible type variables, a reduced set of type variables $\mathtt{Q}$ and the solution itself given as a substitution σ. Since our solver internally works with a set of constraints $\mathtt{C}$ instead of a conjunction φ, the judgments $\Rrightarrow_{set}$ and $\Rrightarrow_{cnj}$ make the corresponding translations. We summarize below the main steps of our solver (a complete description is in [5]).

Transitive Closure ($\Rightarrow_{tr}$). The constraint set is always closed by transitivity such that this step is performed each time a new constraint is added. The transitivity takes into account the conditional constraints, it generates a conditional constraint from a conditional constraint and non-conditional constraint. VPT subtyping (and also interval type subtyping) is transitive since the VPT class declarations are well-formed as in [15].

Simplification ($\Rightarrow_{simplify}$). It consists of a constraint decomposition $\Rightarrow_{\mathtt{s}}$ followed by a transitive closure:

$$\frac{G\vdash C_0 \Rightarrow_{s} C' \quad \vdash C' \Rightarrow_{tr} C}{G\vdash C_0 \Rightarrow_{simplify} C}$$

Constraint decomposition $\Rightarrow_{\mathtt{s}}$ is performed with respect to the class subtyping given by the VPT class hierarchy $\mathtt{G}$, the interval subtyping rule and the subtyping rules for intersection and union types. Using the mechanism presented before the intersection and union types constraints always decompose into conjunctions. A conditional constraint is decomposed into new conditional constraints. The step is performed until the constraint set remains unchanged. In the solver, the first call of $\Rightarrow_{simplify}$ step decomposes the outermost intersection and union types to reduce the complexity of the step $\Rightarrow_{dual}$.

Dual Types Simplification($\Rightarrow_{dual}$). It decomposes all the dual types from the input constraint set $\mathtt{C}_0$. The result consists of a new constraint set $\mathtt{C}$ and the list of the decomposed dual types $\mathtt{D}$:

$$\frac{\vdash C_0 \Rightarrow_{d} D;C_1 \quad \vdash C_1 \Rightarrow_{tr} C_1' \quad D\vdash C_1' \Rightarrow_{cd} C_2 \quad \vdash C_2 \Rightarrow_{tr} C}{\vdash C_0 \Rightarrow_{dual} C;D}$$

The process is performed in two stages. In the first stage ($\Rightarrow_{\mathtt{d}}$), all the flow constraints with dual types are decomposed. When it needs to choose, $\Rightarrow_{\mathtt{d}}$ prefers the flow through the object part of a dual type rather than that through the field part. In the second stage ($\Rightarrow_{cd}$), the flow through the field part is selectively added to the constraint set when it is required by the other constraints.

Variance Inference ($\Rightarrow_{variance}$). It computes the high-bound type variables and the low-bound type variables that do not occur in the closed flow constraints, and resolves them to their default values by the substitutions $\sigma_{\mathtt{H}}$ and $\sigma_{\mathtt{L}}$ respectively.

$$\frac{\begin{array}{c} L_H=\{V.H \mid V.H<:V_1.L\in C_0 \vee V.H<:V_2\in C_0 \vee V.H<:_c V_3.L\in C_0 \vee V.H<:_c V_4\in C_\} \\ L_L=\{V.L \mid V_1.H<:V.L\in C_0 \vee V_2<:V.L\in C_0 \vee V_3.H<:_c V.L\in C_0 \vee V_4<:_c V.L\in C_0\} \\ \sigma_H=[[V.H\mapsto \mathit{Object}] \mid V.H\in \mathtt{fv}(C_0) \wedge V.H\notin L_H] \\ \sigma_L=[[V.L\mapsto \bot] \mid V.L\in \mathtt{fv}(C_0) \wedge V.L\notin L_L] \\ \sigma_{HL}=[[V.L\mapsto V, V.H\mapsto V] \mid V.H\in L_H \wedge V.L\in L_L] \\ Q=Q_0\cup \mathtt{ran}(\sigma_{HL})\setminus(\mathtt{dom}(\sigma_L)\cup\mathtt{dom}(\sigma_H)\cup\mathtt{dom}(\sigma_{HL})) \;\; C=\sigma_H\circ\sigma_L\circ\sigma_{HL}C_0 \end{array}}{\vdash C_0;Q_0 \Rightarrow_{variance} C;Q;\sigma_H\circ\sigma_L\circ\sigma_{HL}}$$

The substitution $\sigma_{\mathtt{HL}}$ implements the second variance inference rule, making equal the bounds of an interval when both of them occur in the closed flow constraints. The initial list of the visible type variables $\mathtt{Q}$ can be affected by the variance inference. This step works on all constraints, either from conditional flow or from main flow.

Type Variables Inference ($\Rightarrow_{typvar}$). This step solves the non-visible type variables in terms of the visible type variables $\mathtt{Q}_0$:

$$\frac{\vdash C_0;Q_0 \Rightarrow_{cycle} C_1;Q_1;\sigma_1 \quad \vdash C_1;Q_1 \Rightarrow_{order} L \quad \vdash L;C_1 \Rightarrow_{unify} C;\sigma_2 \quad Q=Q_1\cup \mathtt{fv}(C)}{\vdash C_0;Q_0 \Rightarrow_{typvar} C;Q;\sigma_2\circ\sigma_1}$$

First substep ($\Rightarrow_{cycle}$) makes equal all the type variables of a cycle. This process may also affect the visible type variables, resulting in a new set $\mathtt{Q}_1$. We use techniques from [10]

to eliminate the cycles. The non-visible type variables are then solved ($\Rrightarrow_{unify}$) in an order given by the number of their low bounds ($\Rrightarrow_{order}$). The substep $\Rrightarrow_{order}$ iteratively computes the order taking into account the situations when the low bounds are class type parameterized with type variables. The substep $\Rrightarrow_{unify}$ unifies the type variables with their low-bounds producing a substitution σ_2. Multiple low-bounds are combined together as an union type. Non-visible type variables of final constraint set $\mathtt{C}$ are promoted as visible in $\mathtt{Q}$. Though this step works only on the main flow constraints, its computed substitutions are also applied on the constraints of the conditional flow.

Solving conditional constraints ($\Rrightarrow_{cond}$). This step translates the conditional constraints into non-conditional constraints if the non-conditional constraints hold. Since it is always safe to add more constraints on the method interface type variables and the constraint set is transitively closed, this step only checks ($\vdash_?$) the conditional constraints with the ground types with respect to the class hierarchy $\mathtt{G}$. First check is for ground constraints, while the last two are to verify if an intersection and an union type can exist in $\mathtt{G}$. If the checks do not hold, the conditional constraints are not translated.

$$\frac{\begin{array}{c} B{=}\{c_1{<:_c}c_2 \mid c_1{<:_c}c_2{\in}C_0\} \quad G{\vdash_?}B \\ \forall V{\in}\mathtt{fv}(C_0).G{\vdash_?}\{c \mid V{<:_c}c{\in}C_0\}\&\{c \mid V{<:}c{\in}C_0\} \\ \forall V{\in}\mathtt{fv}(C_0).G{\vdash_?}\{c \mid c{<:_c}V{\in}C_0\}|\{c \mid c{<:}V{\in}C_0\} \\ C'{=}\{\tau_1{<:}\tau_2|\tau_1{<:_c}\tau_2{\in}C_0\} \quad C_0'{=}C_0\backslash\{\tau_1{<:}\tau_2|\tau_1{<:_c}\tau_2{\in}C_0\} \end{array}}{G{\vdash}C_0{\Rrightarrow_{cond}}C'{\cup}C_0'}$$

Refining the results ($\Rrightarrow_{refine}$). The goal of this step is to reduce the number of visible type variables of a method interface. The first three substitutions are based on the closed flow constraints which are in $\mathtt{C}_0$. The last two substitutions are for the high bound (low bound) type variables occurring on low bound (high bound) positions. Dual types are also translated into intersection types by the substitution $\sigma_{\mathtt{d}}$.

$$\frac{\begin{array}{c} \sigma_1{=}[V_1.H{\mapsto}V, V_2.L{\mapsto}V|V_1.H{<:}V_2.L{\in}C_0 \wedge V{=}\mathtt{fresh}()] \;\; \sigma_2{=}[V.H{\mapsto}t|V.H{<:}t{\in}\sigma_1C_0] \\ \sigma_3{=}[V.L{\mapsto}t|t{<:}V.L{\in}\sigma_2{\circ}\sigma_1C_0] \;\; \sigma_4{=}[V{\mapsto}t|t{<:}V{\in}\sigma_3{\circ}\sigma_2{\circ}\sigma_1C_0 \vee V{<:}t{\in}\sigma_3{\circ}\sigma_2{\circ}\sigma_1C_0] \\ \sigma'{=}\sigma_3{\circ}\sigma_2{\circ}\sigma_1{\circ}\sigma_t \quad G{\vdash}\sigma'D{\Rrightarrow_{refinedual}}\sigma_d \\ \sigma_1'{=}[V.H{\mapsto}V|V_1.L{\mapsto}V.H{\in}\sigma' \wedge V{=}\mathtt{fresh}()] \;\; \sigma_2'{=}[V.L{\mapsto}V|V_1.H{\mapsto}V.L{\in}\sigma' \wedge V{=}\mathtt{fresh}()] \\ \sigma{=}\sigma_2'{\circ}\sigma_1'{\circ}\sigma_d{\circ}\sigma' \end{array}}{G{\vdash}C_0;D;Q_0;\sigma_0{\Rrightarrow_{refine}}\sigma C_0;\sigma Q_0;\sigma}$$

5 Method Overriding

Consider the following method overriding example, where the method `boo` of the class `Cell` is overridden by the subclass `Pair` (note that `class Pair extends Cell{..}`):

`Cell` | `Object boo(Cell a) {this.fst = a.fst; return a.fst; }`

`Pair` | `Object boo(Cell a) {a.fst = this.fst; this.snd = a.fst; return a.fst; }`

Applying our inference to each method, we obtain the following results:

$\mathtt{Cell}\langle{\ominus}\mathtt{P}\rangle$ | $\mathtt{P}$ `boo(`$\mathtt{Cell}\langle{\oplus}\mathtt{P}\rangle$ `a){..}` $\quad$ $\mathtt{Pair}\langle{\oplus}\mathtt{P}_1{,}{\ominus}\mathtt{P}_1\rangle$ | $\mathtt{P}_1$ `boo(`$\mathtt{Cell}\langle{\odot}\mathtt{P}_1\rangle$ `a){..}`

The method overriding is sound only if the overriding method is a subtype of the overridden method and the overriding method's receiver is a subtype of the overridden method's receiver [3]. As can be seen, this property does not hold for the above inferred methods:

$$\mathtt{Pair}\langle{\oplus}\mathtt{P}_1{,}{\ominus}\mathtt{P}_1\rangle{<:}\mathtt{Cell}\langle{\ominus}\mathtt{P}\rangle \quad \mathtt{Cell}\langle{\oplus}\mathtt{P}\rangle{<:}\mathtt{Cell}\langle{\odot}\mathtt{P}_1\rangle \quad \mathtt{P}_1{<:}\mathtt{P}$$

To ensure this property, we augment our inference algorithm with the following considerations: (i) we can strengthen the receiver type and the result type of the overriding method; (ii) we can strengthen the parameters types and the precondition of the overridden method. Thus the method overriding problem is solved as follows:

1. Infer the overridden method as: $\texttt{Cell}\langle[\texttt{P},\mathit{Object}]\rangle \mid \texttt{P boo}(\texttt{Cell}\langle[\bot,\texttt{P}]\rangle\ \texttt{a})$
2. Undo the variance of the overridden method parameters by using fresh interval type variables ($\texttt{P}_1\texttt{.H}$ and $\texttt{P}_1\texttt{.L}$) that keep the relation with the other type variables of the receiver and the result: $\texttt{Cell}\langle[\texttt{P},\mathit{Object}]\rangle \mid \texttt{P boo}(\texttt{Cell}\langle[\texttt{P}_1\texttt{.L},\texttt{P}_1\texttt{.H}]\rangle\ \texttt{a})\ \texttt{where}\ \texttt{P}_1\texttt{.H}<:\texttt{P}$
3. Do inference for the overriding method: The process starts with the sound overriding assumptions (Step 1):

$$\texttt{Pair}\langle[\texttt{V}_1\texttt{.L},\texttt{V}_1\texttt{.H}],[\texttt{V}_2\texttt{.L},\texttt{V}_2\texttt{.H}]\rangle \mid \texttt{Y boo}(\texttt{Cell}\langle[\texttt{V}_3\texttt{.L},\texttt{V}_3\texttt{.H}]\rangle\ \texttt{a})\ \{\texttt{a.fst=this.fst;this.snd=a.fst;return a.fst;}\}$$

1.Overriding Assumptions	$\texttt{Pair}\langle[\texttt{V}_1\texttt{.L},\texttt{V}_1\texttt{.H}],[\texttt{V}_2\texttt{.L},\texttt{V}_2\texttt{.H}]\rangle<:\texttt{Cell}\langle[\texttt{P},\mathit{Object}]\rangle \wedge \texttt{Y}<:\texttt{P} \wedge \texttt{Cell}\langle[\texttt{P}_1\texttt{.L},\texttt{P}_1\texttt{.H}]\rangle<:\texttt{Cell}\langle[\texttt{V}_3\texttt{.L},\texttt{V}_3\texttt{.H}]\rangle \wedge \texttt{P}_1\texttt{.H}<:\texttt{P}$
2.Collect Constraints	$\texttt{V}_1\texttt{.H}<:\texttt{V}_3\texttt{.L} \wedge \texttt{V}_3\texttt{.H}<:\texttt{V}_2\texttt{.L} \wedge \texttt{V}_3\texttt{.H}<:\texttt{Y}$
3.Simplify	$\texttt{P}<:\texttt{V}_1\texttt{.L} \wedge \texttt{Y}<:\texttt{P} \wedge \texttt{V}_3\texttt{.L}<:\texttt{P}_1\texttt{.L} \wedge \texttt{P}_1\texttt{.H}<:\texttt{V}_3\texttt{.H} \wedge \texttt{P}_1\texttt{.H}<:\texttt{P}$ $\texttt{V}_1\texttt{.H}<:\texttt{V}_3\texttt{.L} \wedge \texttt{V}_3\texttt{.H}<:\texttt{V}_2\texttt{.L} \wedge \texttt{V}_3\texttt{.H}<:\texttt{Y}$
4.Infer Variance	$\texttt{V}_2\texttt{.H}=\mathit{Object} \wedge \texttt{V}_1\texttt{.L}=\texttt{V}_1\texttt{.H} \wedge \texttt{V}_3\texttt{.L}=\texttt{V}_3\texttt{.H} \wedge \texttt{P}_1\texttt{.L}=\texttt{P}_1\texttt{.H}$
5.Infer Type Vars	$\texttt{P}=\texttt{P}_1\texttt{.H}=\texttt{P}_1\texttt{.L}=\texttt{V}_1\texttt{.H}=\texttt{V}_1\texttt{.L}=\texttt{V}_2\texttt{.L}=\texttt{V}_3\texttt{.H}=\texttt{V}_3\texttt{.L}$

4. The result of the previous step is applied on both overridden and overriding methods and we obtain the following sound result:

$$\texttt{Cell}\langle\ominus\texttt{P}\rangle \mid \texttt{P boo}(\texttt{Cell}\langle\odot\texttt{P}\rangle\ \texttt{a}) \qquad \texttt{Pair}\langle\odot\texttt{P},\ominus\texttt{P}\rangle \mid \texttt{P boo}(\texttt{Cell}\langle\odot\texttt{P}\rangle\ \texttt{a})$$

6 Conclusion

We have formalized a novel constraint-based algorithm to infer variant parametric types for non-generic Java code. In contrast to the previous refactoring algorithms [9, 8, 7, 11, 16] which mainly support invariant subtyping and are designed as whole program analyses, our approach offers full support for use-site variance based subtyping and it is designed as a summary-based analysis that works on a per method basis. The main technical novelty of our approach is a systematic variance inference based on interval types. With the full support for use-site variance based subtyping, our approach can generate better generic types than those derived by existing systems. For instance, none of the previous algorithms can automatically infer the examples from Section 3.4 and Section 3.5.

Although our inference algorithm internally works with sophisticated mechanisms, its output is expressed in terms of variant parametric types extended with restricted forms of intersection/union types as used in [4]. We have proven the soundness of our inference algorithm with respect to our variant parametric type system in [4]. However the completeness requirement is a difficult problem since the decidability of nominal subtyping with use-site variance is still an open problem as was discussed in [15].

We have built an inference prototype which works for a core subset of Java. Our previous VPT checker from [4] is used to validate the inferred results. In our initial experiments we have tested the quality of our inference system results on a small set of non-generic programs by comparing the inferred generic types with the best generic types that one can manually provide. In all the cases, our system was able to infer the same types as those manually provided. The inference time was less than one second for each test program. Currently we are working to extend our experiments to larger programs by using our translator of Java to a core subset [6].

References

1. G. Bracha, M. Oderski, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM OOPSLA*, 1998.
2. M. Bugliesi and S. M. P. Geertsen. Type inference for variant object types. *Information and Computation*, 177(1), 2002.
3. G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM TOPLAS*, 17(3), 1995.
4. W. N. Chin, F. Craciun, S.C. Khoo, and C. Popeea. A flow-based approach for variant parametric types. In *ACM OOPSLA*, 2006.
5. F. Craciun, W. N. Chin, , G. He, and S. Qin. An Interval-based Inference of Variant Parametric Types. Technical report, Department of Computer Science, Durham University, UK., December 2008. Available at http://www.durham.ac.uk/shengchao.qin/papers/VPTinfer.pdf.
6. F. Craciun, H. Y. Goh, and W. N. Chin. A framework for object-oriented program analyses via Core-Java. In *IEEE Internationl Conference on Intelligent Computer Communication and Processing*, 2006.
7. D. Dincklage and A. Diwan. Converting Java Classes to use Generics. In *ACM OOPSLA*, 2004.
8. A. Donovan, A. Kiezun, M. S. Tschantz, and M.D. Ernst. Converting Java Programs to Use Generic Libraries. In *ACM OOPSLA*, 2004.
9. D. Duggan. Modular Type-based Reverse Engineering of Parameterized Types in Java Code. In *ACM OOPSLA*, 1999.
10. M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *ACM PLDI*, 1998.
11. R. Fuhrer, F. Tip, A. Kiezun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.
12. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2005.
13. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *ACM PLDI*, 2001.
14. A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM TOPLAS*, 28(5), 2006.
15. A. Kennedy and B. Pierce. On Decidability of Nominal Subtyping with Variance. In *FOOL/WOOD*, 2007.
16. A. Kieżun, M. D. Ernst, F. Tip, and R.M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE*, 2007.
17. M. Odersky. Inferred Type Instantiation for GJ, January 2002. Notes, 2002.
18. F. Pottier. Simplifying Subtyping Constraints. In *ACM ICFP*, 1996.
19. F. Pottier. *Type inference in the presence of subtyping: from theory to practice*. PhD thesis, Universite Paris 7, 1998.
20. J. C. Reynolds. Preliminary design of the programming language Forsythe. Technical report, CMU-CS-88-159,Carnegie Mellon, 1988.
21. D. Smith and R. Cartwright. Java type inference is broken: Can we fix it? In *ACM OOPSLA*, 2008.
22. Z. Su, M. Fahndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *ACM POPL*, 2000.
23. M. Torgersen, E. Ernst, C. P. Hansen, P. von der Ahe, G. Bracha, and N. Gafter. Adding Wildcards to the Java Programming Language. *JOT*, 3(11), 2004.
24. M. Torgersen, E. Ernst, and C.P. Hansen. WildFJ. In *FOOL*, 2005.
25. V. Trifonov and S. Smith. Subtyping Constrained Types. In *SAS*, 1996.