A Flow-Based Approach for Variant Parametric Types

Wei-Ngan Chin Florin Craciun Siau-Cheng Khoo Corneliu Popeea

Department of Computer Science, National University of Singapore
{chinwn,craciunm,khoosc,corneliu}@comp.nus.edu.sg

Abstract

A promising approach for type-safe generic codes in the object-oriented paradigm is *variant parametric type*, which allows covariant and contravariant subtyping on fields where appropriate. Previous approaches formalise variant type as a special case of the existential type system. In this paper, we present a new framework based on *flow analysis* and *modular type checking* to provide a simple but accurate model for capturing generic types. Our scheme stands to benefit from past (and future) advances in flow analysis and subtyping constraints. Furthermore, it fully supports casting for variant types with a special reflection mechanism, called *cast capture*, to handle objects with unknown types. We have built a constraint-based type checker and have proven its soundness. We have also successfully annotated a suite of Java libraries and client code with our flow-based variant type system.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; Polymorphism; Constraints; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Object-oriented constructs; Type structure

General Terms Design, Languages, Theory, Verification

Keywords Genericity, Flow Analysis, Variant Parametric Types, Subtyping, Constraints

1. Introduction

Software reuse is an important aspect of software engineering. Traditionally, most mainstream object-oriented (OO) languages, such as Java, C++ and C#, have relied on class subtyping to support reuse (or genericity) via inclusion polymorphism. While this mechanism allows the convenient storage of objects via safe upcast into generic data structure, the converse process of retrieving objects from the same data structure requires downcast testing, which incurs runtime overheads and is possibly unsafe.

To address the shortcomings of inclusion polymorphism, there have been several recent proposals (amongst the Java [3] and C# [19] communities) for parametric types to be supported . Here, each class `c` is allowed to carry a list of type parameters for its fields, e.g., `c⟨t1,..,tn⟩`, whereby the type of each field can either be instantiated or left as a type variable. Below are two classes whose fields have been parameterised:

```
class Cell⟨A⟩ {
 A fst; ··· }
class Pair⟨A,B⟩ extends Cell⟨A⟩ {
 B snd; ··· }
```

With such parameterised class declarations, we may then define specialised instances, such as `Cell⟨Int⟩`, `Cell⟨Float⟩` or `Pair⟨Int,Num⟩`, which contain more specific type information for the fields of each class instance. Though parametric types can co-exist with class subtyping, pointwise matching of the respective fields is required. For example, the subtyping relation (denoted by `<:`) `Pair⟨t1,t2⟩ <:Cell⟨t3⟩` is allowed only when `Pair<:Cell` and `t1=t3`. The latter condition is for pointwise matching of the common field. Similarly, `Pair⟨t1,t2⟩<:Pair⟨t3,t4⟩` holds, provided `t1=t3` and `t2=t4`. Pointwise matching (invariant subtyping) is required because field reading and field writing are based on opposite flows that change the directions of subtyping. This requirement limits the reusability of programs based on parametric types.

To address this shortcoming, Igarashi and Viroli [17] developed a new variant parametric type system (or variant type, in short) to improve the subtyping of generic structures, depending on how the fields are being accessed. Let `c` denote a class with one type parameter. Let `o` denote an object of variant type $\texttt{c}\langle\alpha_1 t_1\rangle$ while `v` denotes a location of variant type $\texttt{c}\langle\alpha_2 t_2\rangle$, into which `o` is to pass. Each variant type $\texttt{c}\langle\alpha t\rangle$ has a variance α (see Section 3.1) attached to its field to indicate how the field is to be accessed. A field that is subject to read-only access via reference of `v` (denoted by $\alpha_2 = \oplus$) may be supported by covariant subtyping. That is, $\texttt{c}\langle\alpha_1 t_1\rangle\texttt{<:c}\langle\oplus t_2\rangle$ if $\alpha_1\texttt{<:}\oplus$ and $t_1\texttt{<:}t_2$. Conversely, a field that is subject to write-only access via reference of `v` (denoted by $\alpha_2 = \ominus$) may be supported by contravariant subtyping. That is, $\texttt{c}\langle\alpha_1 t_1\rangle\texttt{<:c}\langle\ominus t_2\rangle$ if $\alpha_1\texttt{<:}\ominus$ and $t_2\texttt{<:}t_1$. Also, a field that is subject to both read and write accesses via reference of `v` (denoted by $\alpha_2 = \odot$) must be supported by invariant subtyping. That is, $\texttt{c}\langle\alpha_1 t_1\rangle\texttt{<:c}\langle\odot t_2\rangle$ if $\alpha_1\texttt{<:}\odot$ and $t_1\texttt{<:}t_2 \wedge t_2\texttt{<:}t_1$. Lastly, if a field is not accessed via reference of `v` (denoted by $\alpha_2 = \circledast$), we can use bivariant subtyping. That is, we support $\texttt{c}\langle\alpha_1 t_1\rangle\texttt{<:c}\langle\circledast t_2\rangle$ for any t_1 and t_2.

Variant types give a much richer subtyping hierarchy than parameterised types do. Figure 1 illustrates some variant types for `Cell` objects and their places in the subtyping hierarchy. Note that $\rightarrow$ denotes a subtyping relation in the graph. Also, $\texttt{Cell}\langle\circledast\texttt{t}\rangle$, $\texttt{Cell}\langle\oplus\texttt{Object}\rangle$ and $\texttt{Cell}\langle\ominus\bot\rangle$ are equivalent to each other while $\texttt{Cell}\langle\odot\texttt{Num}\rangle$, $\texttt{Cell}\langle\odot\texttt{Float}\rangle$ and $\texttt{Cell}\langle\odot\texttt{Int}\rangle$ are unrelated. Note that $\bot$ denotes the type of $null$ values which can be assigned into any class type. However, each $\texttt{Cell}\langle\odot\texttt{t}\rangle$ is a subtype of both $\texttt{Cell}\langle\oplus\texttt{t}\rangle$ and $\texttt{Cell}\langle\ominus\texttt{t}\rangle$. Also, types of the form $\texttt{Cell}\langle\oplus\texttt{t}\rangle$ and $\texttt{Cell}\langle\ominus\texttt{t}\rangle$ have a subtyping hierarchy based on covariance and contravariance, respectively.

The benefits of variant typing have been known for some time. However, early proposals have attached access rights to the fields

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

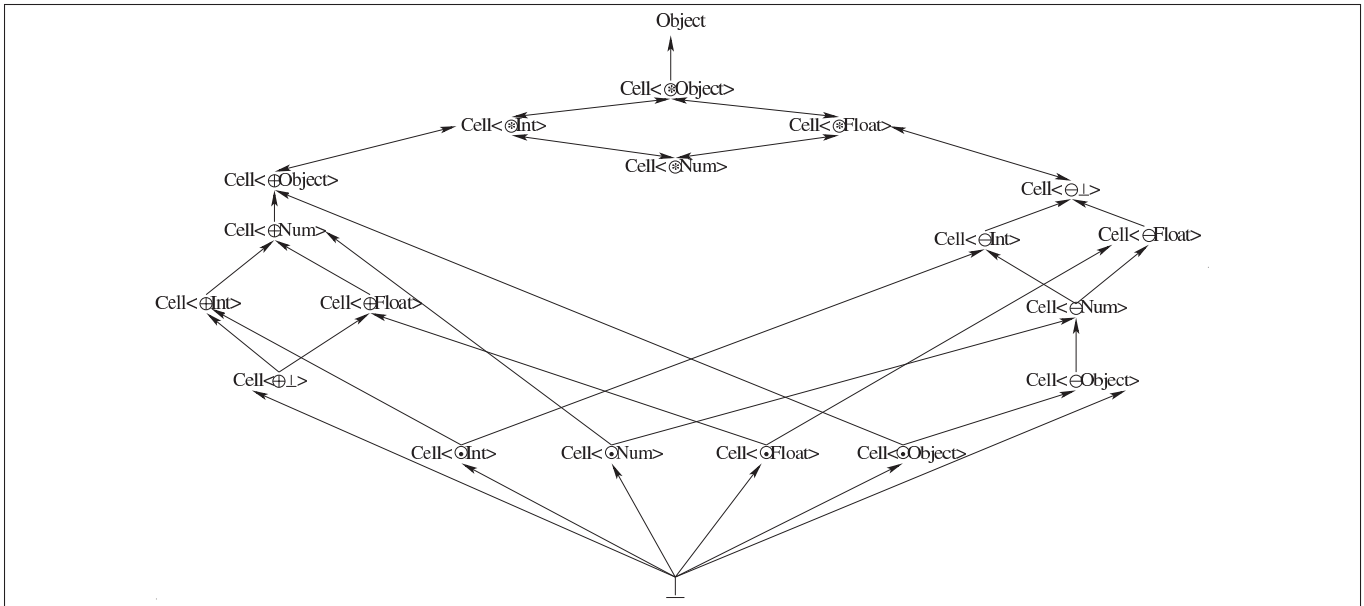

Figure 1. A Rich Subtyping Hierarchy

of each class declaration. This mechanism is known as *declaration-site variance* and is shown in the following example:

```
class DSCell⟨A⟩ {
 ⊕A fst;
 A getFst() { return fst; }
 void setFst(A x) { fst=x; } }
```

The field `fst` is declared read only using the variance $\oplus$. Consequently, the method `setFst` cannot be invoked. Using the concept of structured virtual type, Thorup and Torgersen [35] were the first to link access rights and covariant subtyping to the fields of each *use of a class* rather than the class declaration itself. This *use-site variance* mechanism is much more flexible than previous mechanisms based on *declaration-site variance*. In the following example, the access to the field `fst` is governed by the variance variable α. A reference of type `USCell`$\langle\oplus$`Int`$\rangle$ allows read-only access, while a reference of type `USCell`$\langle\odot$`Int`$\rangle$ allows read-write access to the field `fst`.

```
class USCell⟨αB⟩ {
 αB fst;
 B getFst() { return fst; }
 void setFst(B x) { fst=x; } }
```

Later, Igarashi and Viroli extended this concept to support contra- and bi-variance [17]. They formalised the variant type system by mapping it into a corresponding *existential type* system [7, 21, 22]. A recent proposal by Sun Microsystems for generics in Java 1.5 [37] supports *wildcard type* based on an improvement of Igarashi and Viroli's variant type system, but it is still viewed as a special case of the existential type system with subtyping. However, a more general version of existential type system, called System $F_{\leq}$, has undecidable subtyping [28], while the decidability of Igarashi and Viroli's variant type system, remains an open problem [18].

In this paper, we propose a new approach for the variant parametric type system that is based on the mechanism of flow analysis. Our flow analysis captures value flows via subtyping constraints. A major benefit of this approach is the considerable knowledge in flow analysis that has been accumulated in the recent past [25, 32, 38, 15, 16, 34, 26]. In particular, to support modular type-checking, we require non-structural subtype entailment of the form $\forall\overline{v}(C_1 \Longrightarrow \exists\overline{w}C_2)$, where C_1, C_2 are subtyping constraints while $\overline{v}, \overline{w}$ are sets of type variables. These constraints are non-structural as we use $\bot <: t <: \mathit{Object}$, to support the OO class inheritance mechanism. While the decidability of non-structural subtype entailment remains an open problem, there exist sound approximations that use constraint simplification and induction techniques [32, 38]. Our work is built on top of sound but practical solutions to subtyping (flow) constraints, and we have developed a systematic framework for the variant parametric type system with the following new features:

- Our framework is based on *flow analysis* which can concisely and intuitively capture flow of values on a per method basis (Section 3). We use variance annotations primarily to predict the flows of values, and not for access control. We also provide special considerations for two type values. A value of `Object` type can always flow out from any location while a null value of $\bot$ type can always flow into any location.
- We augment our generic type system with *intersection type* to help capture information flow more accurately. An intersection type `t1&t2` denotes a type with both the properties of `t1` and `t2`. Such types are important for languages with multiple inheritance (such as Java via its interface mechanism), and can accurately capture the flow of objects with their expected field accesses.
- Our approach is based on *modular type checking* (Section 5). Each method is specified with a flow constraint (and variant types) that is used to predict the value flows that may occur in the method's body. We verify each method separately to ensure that the predicted accesses, flow constraint and variant typings are efficiently and safely checked.
- We advocate the support of *downcast to arbitrary variant types* (Section 6). With this mechanism is a novel *cast capture* that uses a reflection technique to deal with values of unknown type. Cast capture has helped improve the generic implementation of several JDK 1.5 libraries.

- We present a soundness theorem and a *variant type checker*. We have successfully applied our prototype to a suite of Java libraries and client codes (Section 8). On average, we are able to eliminate 87.9% of the casts from non-generic Java 1.4 application code, that means 12.9% more casts than wildcard-generic Java 1.5 application code.

Our goal is to strive for type-safe OO programs with better genericity via a modular flow-based approach to variant parametric type system. Next, we explain our approach with the help of some examples.

2. Better Genericity

The main goal of genericity is to support highly reusable software components. To allow this to happen in a type-safe way, we should strive to provide type descriptions that are concise, understandable, general and accurate. Specifically, each well-typed generic program should be accurately identified where possible. As a side benefit, we are able to track type information in a precise manner, allowing redundant cast operations to be eliminated where possible. In this section, we examine the key aspects for which our approach based on flow analysis makes improvements over existing approaches based on existential types. Some of these improvements may not be peculiar to the flow-based approach, but they were gradually developed starting from a different view point.

2.1 Intersection Type

Parametric type systems use number of cast operations eliminated as a measure of accuracy [10, 14]. As it turns out, there may be competing decisions on what types to use for certain cast operations to be eliminated. The following example from [10] illustrates:

```
class B1 extends A implements I { ··· }
class B2 extends A implements I { ··· }
void foo(Boolean b) {
 Cell cb1 = new Cell(new B1());
 Cell cb2 = new Cell(new B2());
 Cell c = b ? cb1 : cb2;
 A a = (A) c.get();
 I i = (I) c.get();
 B1 b1 = (B1) cb1.get();
 B2 b2 = (B2) cb2.get();  }
```

This program contains four cast operations. With the help of parametric types, Donovan et al. [10] suggested three sets of possible types, each with a different subset of casts eliminated, as summarised below:

Types of Variables			Casts Eliminated			
cb1	cb2	c	(A)	(I)	(B1)	(B2)
Cell⟨B1⟩	Cell⟨B2⟩	Cell			√	√
Cell⟨A⟩	Cell⟨A⟩	Cell⟨A⟩	√			
Cell⟨I⟩	Cell⟨I⟩	Cell⟨I⟩		√		

Note that `Cell` denotes a raw type where nothing is known of its components. Hence, only `Object` values are statically retrievable from it. Raw type was originally proposed in [3] for backwards compatibility, and it is the basis for generic typing through inclusion polymorphism. However, none of the three proposed solutions are able to eliminate all four casts. This indicates that parametric typing is not expressive enough to capture generic type for such programs. There are two possible improvements. First, note that the fields of `cb1, cb2` and `c` are subject to read-only accesses, and not modified in the program fragment. We can therefore provide covariant annotations to the fields of these variables, and obtain two possible outcomes, each with three casts eliminated:

cb1	cb2	c	(A)	(I)	(B1)	(B2)
Cell⟨⊕B1⟩	Cell⟨⊕B2⟩	Cell⟨⊕A⟩	√		√	√
Cell⟨⊕B1⟩	Cell⟨⊕B2⟩	Cell⟨⊕I⟩		√	√	√

Second, both classes `B1` and `B2` have supertypes `A` and `I` in common. To exploit this, we can use an intersection type parameter in `Cell`⟨⊕`(A&I)`⟩ to describe the variable `c`. In a lattice of type values, an intersection type `A&I` essentially defines the greatest lower bound of `A` and `I`. With this, all four casts can now be eliminated in our new solution to genericity, as shown below:

cb1	cb2	c	(A)	(I)	(B1)	(B2)
Cell⟨⊕B1⟩	Cell⟨⊕B2⟩	Cell⟨⊕A&I⟩	√	√	√	√

Note that the above example cannot be coded in Java 1.5 syntax. Java 1.5 does not allow the use of intersection types for local variable declaration, field declaration or method argument/return types. Intersection types can be used only as upper bounds for a method type parameter.

2.2 Modular Flow Specification

Another important principle for better genericity is that type description should be designed in a *modular* fashion (on a per method basis). Type annotations appearing in the method header should depend only on the method body while each call site should be a specific instance of the method's type declaration. This principle is important for efficient type checking and ease of type annotation. Specifically, for each instance method, we provide the following method declaration:

t `|` t_0 `mn(`t_1 v_1`,...,`t_n v_n`) where` ψ `{...}`

A separate annotation "t `|`" is added at the beginning of each method's declaration to capture the variance of the implicit `this` parameter. This separate annotation (omitted in previous works, such as [17, 37]) allows us to capture the behaviour of each method, independent of its class declaration. Note that ψ captures the expected value flows of each method's body in terms of type of the parameters ($t_1, .., t_n$), result (t_0), and receiver (t). We support modular type checking by localising type variables which are not present in the type of parameters, result and receiver. A previous approach [17] relies on the existential open/close mechanism for the receiver parameter to determine if the receiver parameter is of suitable variance while other parameters are checked via subtyping. In contrast, we achieve uniform treatment for all parameters.

To illustrate the modular type annotation mechanism, consider three method declarations for the `Pair` class:

```
class Pair⟨A,B⟩ extends Cell⟨A⟩ {
 B snd;
 Pair⟨⊛,⊕Y⟩ | Y getSnd()
   {return this.snd;}
 Pair⟨⊛,⊖Y⟩ | void setSnd(Y v)
   {this.snd=v;}
 Pair⟨⊛,⊛⟩&W | Pair⟨⊙W,⊙W⟩ dup()
   {return new Pair⟨W,W⟩(this,this);} }
```

First, note that `getSnd` will read the second field while `setSnd` will write to it. Because of these effects, we may apply covariant (⊕) and contravariant (⊖) subtypings to the second component of the `Pair` object for `getSnd` and `setSnd`, respectively. Second, bivariant (⊛) subtyping is allowed for the unaccessed component of the `Pair` object for both methods. As a shorthand, we may write ⊛ to denote ⊛t since all bivariant types are equivalent. Note that `Y` from `getSnd` and `Y` from `setSnd` denote different type variables treated independently by our modular type checker.

The third method is an interesting application of intersection type. The method itself does not access the fields of the `this` parameter, which escapes into the two fields of the method's `Pair` result. To capture this value flow, we declare an intersection type $\texttt{Pair}\langle\circledast,\circledast\rangle\&\texttt{W}$ for the `this` parameter. The type $\texttt{Pair}\langle\circledast,\circledast\rangle$ is to acknowledge that we have a `Pair` object whose fields are not accessed by the current `dup` method. A type variable `W` helps indicate that this parameter will escape into the fields of the result with type $\texttt{Pair}\langle\odot\texttt{W},\odot\texttt{W}\rangle$. This flow allows the variant type of `W` to flow into the two fields of the output `Pair`. Hence, for a given receiver of type t, we have t<:$\texttt{Pair}\langle\circledast,\circledast\rangle$ and t<:`W`. Possible candidates for the type t are $\texttt{Pair}\langle\oplus\texttt{X},\oplus\texttt{Y}\rangle$ or $\texttt{Pair}\langle\oplus\texttt{X},\ominus\texttt{Y}\rangle$, etc. In contrast, if we use the following type suggested in [17]:

$$\texttt{Pair}\langle\odot\texttt{X},\odot\texttt{Y}\rangle \mid \texttt{Pair}\langle\odot\texttt{Pair}\langle\odot\texttt{X},\odot\texttt{Y}\rangle,\odot\texttt{Pair}\langle\odot\texttt{X},\odot\texttt{Y}\rangle\rangle\ \texttt{dup()}$$

we require t=$\texttt{Pair}\langle\odot\texttt{X},\odot\texttt{Y}\rangle$ or t=$\bot$, which restricts the possible uses of the method. One way to improve this situation is to duplicate the `dup` method for different scenarios, as shown below:

$$\texttt{Pair}\langle\oplus\texttt{X},\oplus\texttt{Y}\rangle \mid \texttt{Pair}\langle\odot\texttt{Pair}\langle\oplus\texttt{X},\oplus\texttt{Y}\rangle,\odot\texttt{Pair}\langle\oplus\texttt{X},\oplus\texttt{Y}\rangle\rangle\ \texttt{dup()}$$
$$\texttt{Pair}\langle\oplus\texttt{X},\ominus\texttt{Y}\rangle \mid \texttt{Pair}\langle\odot\texttt{Pair}\langle\oplus\texttt{X},\ominus\texttt{Y}\rangle,\odot\texttt{Pair}\langle\oplus\texttt{X},\ominus\texttt{Y}\rangle\rangle\ \texttt{dup()}$$
$$\texttt{Pair}\langle\odot\texttt{X},\ominus\texttt{Y}\rangle \mid \texttt{Pair}\langle\odot\texttt{Pair}\langle\odot\texttt{X},\ominus\texttt{Y}\rangle,\odot\texttt{Pair}\langle\odot\texttt{X},\ominus\texttt{Y}\rangle\rangle\ \texttt{dup()}$$

However, such duplications go against the goal of genericity. On the other hand, our solution with intersection types can improve genericity by allowing value flows to be accurately captured.

2.3 Avoiding F-Bounds where Possible

One feature that adds to the expressivity of bounded existential type is the use of F-bounds [5] which effectively capture recursive constraints of the form T<:$C\langle..,T,..\rangle$ where T is a type variable and C is a class name. While the designers of Java 1.5 consider this feature to be significant and useful [37], it is also a source of complication as reported recently in [20]. In particular, F-bound together with existential type is a source of undecidability for System $F_{\leq}$ which caused an earlier implementation of Java 1.5 to fail in accepting some programs with F-bounds that were actually type-safe (as first reported in [20]). Subsequent improvements in Java 1.6 have removed the reported errors, but the decidability of its type system remains an open problem.

While the flow-based approach that we advocate also supports recursive flow constraints (if the inductive mechanism of [32, 38] is used), our pragmatic philosophy is to avoid F-bounds whenever it is possible to do so.

As an example of F-bound, consider the following definition of the `Comparable` interface for Java 1.5:

```
interface Comparable⟨T⟩ {
 int compareTo(T o);  }
```

Here, class parameter `T` is being used to capture the parameter of the method `compareTo`. As this parameter is required to be a subtype of `Comparable` itself, F-bound of the form $\texttt{T<:Comparable}\langle\ominus\texttt{T}\rangle$ is usually needed when `Comparable` is used, as shown in the next example:

```
class Collections {
 ⟨T extends Comparable⟨? super T⟩⟩ static T max
  (Collection⟨? extends T⟩ col) { ··· }
}
```

In our flow-based approach, the current philosophy is to capture the value flows of each method independently. Hence, we have chosen to capture the value flow and subtyping relation directly for each method instead, as shown below for our definition of `Comparable`:

```
interface Comparable⟨A⟩ {
 Comparable⟨⊙T⟩ | int compareTo(T o);
}
```

Based on this definition, we can write the `max` method, as follows:

```
class Collections {
 static T max(Collection⟨⊕T⟩ col)
  where T<:Comparable⟨⊖T⟩ { ··· }
}
```

This alternative is equivalent to the earlier Java 1.5 definition.

We also support a simpler way, to express `Comparable` interface, as follows:

```
interface Comparable {
 S & Comparable | int compareTo(T o)
  where T<:Comparable ∧ T<:S ;
}
```

The use of this definition does not require any F-bound, but it is more restrictive than Java 1.5 definition of `Comparable` interface.

Another potential use of F-bound occurs for recursive fields of class declarations. An example is the following recursive `List` class:

```
class List⟨A,B⟩ extends Object
  where B<:List⟨A,B⟩ {
 A val;
 B next; ...
}
```

This solution uses an F-bound $\texttt{B<:List}\langle\texttt{A,B}\rangle$ that makes constraint solving more complex [32]. However, in our system we may choose to avoid the recursive constraint from the invariant of the class `List` by leaving the recursive `next` field with an incomplete variance $\oslash$, as follows:

```
class List⟨A⟩ extends Object {
 A val;
 ⊘List⟨A⟩ next; ...
}
```

The variance of the `next` field is incomplete at its declaration site and can be promoted to either $\odot$ or $\oplus$, depending on how its underlying type parameter $\texttt{List}\langle\texttt{A}\rangle$ is being instantiated at the use site. This type promotion process is elaborated later in Section 4.2, and can be used to avoid F-bound, where possible.

2.4 Avoiding Existential Type Always

It has been generally acknowledged that existential type is useful for describing data types whose implementation details can be made abstract. This aspect is closely related to the use of bivariant type $\circledast t$ where the underlying type t is unknown and may be assumed to be of any type. While no-access is one way to enforce bivariant type, it is also possible to use the open/close mechanism of existential type system to describe situations where implementation details can be made abstract. A typical example is the `copy` operation on two elements of a vector that was highlighted in [18], and reproduced below:

```
void copy(Vector⟨⊛⟩ x, int i, int j) {
 open x as [Y,y] in
  y.setElementAt(y.elementAt(i),j)
}
```

The above code opens the bivariant type of `x` as an object bound to variable `y` with an abstract type `Y`. As all elements of each vector are of the same `Y` type, we may safely copy a value from one position of the vector into another position, without knowing the

actual underlying type. The close correspondence between existential type and bivariant type is a primary reason why Igarashi and Viroli considered existential type system as the underlying model for their variant parametric type system.

However, the designers of Java 1.5 considered the open/close mechanism of existential type system to be somewhat restrictive [36]. They have therefore proposed a relaxation to open each expression as an existential type by associating it with a global type variable *without* a corresponding close operation. This use is similar to the flow-based approach where each parameter (or local variable) is regarded as a location where values may flow in and/or out. Nevertheless, in the context of existential type system, such relaxation might possibly be unsound since each existential type may in fact correspond to contradicting type values. This is possibly why correctness proof is yet to be completed (as of [36]), even though a full-scale implementation for wildcard type system has already been released for public use.

Furthermore, Java 1.5 relied on polymorphic (generic) type system for selected methods to capture situations where invariant type appears necessary, as shown by the following example:

```
⟨T⟩ void docopy(Vector⟨T⟩ x, int i, int j) {
 T tmp = x.elementAt(i);
 x.setElementAt(tmp,j)
}
```

Through a wildcard capture mechanism, it is possible to provide a method with bivariant parameter, as shown below:

```
void copy(Vector⟨?⟩ x, int i, int j) {
 docopy(x,i,j);
}
```

Note that wildcard type of `x` has been captured by the global `T` type variable. Again, the open/close mechanism is averted, even though the underlying system is still based on bounded existential type system.

Our current philosophy is to avoid existential type system altogether. To capture the effect of an unknown abstract type, we have introduced a casting mechanism that is able to capture the underlying type of an object via a fresh type variable. We refer to this as a *cast capture* technique which is elaborated in more details in Section 6. The same `copy` method can be re-written with a casting of the `x` parameter from bivariant type `Vector`$\langle\circledast\rangle$ to an invariant type (`Vector`$\langle\odot$`T`$\rangle$). In the process, `T` is used to capture the unknown type, as shown below:

```
void copy(Vector⟨⊛⟩ x, int i, int j) {
 Vector⟨⊙S⟩ w;
 {w = (Vector⟨⊙T⟩) x;
  w.setElementAt(w.elementAt(i),j) }
}
```

While this cast capture construct may look like a syntactic sugar for the open/close mechanism, we stress that it is part of a more general mechanism that can take an arbitrary type as source (instead of a bivariant type) for casting into another arbitrary type as target (instead of an invariant type). A cast for a c_1-object into an invariant type of the form $c_2\langle(\odot t)^*\rangle$ where $c_1{<:}c_2$ is always safe since every object is built using an invariant type. Furthermore, cast-capture is a runtime mechanism while open-close is a type-related operation to expose an obtained type at compile-time. Our cast capture mechanism using reflection is more general as it can capture type values at runtime, and also support a mix of cast capture and cast testing. In our formulation of variant parametric type system, the flow-based approach with casting has therefore avoided the need for existential type system altogether.

Some readers may contend that the casting mechanism is the prerogative of programmers and may be too burdensome to write. While this is so, we believe that there is still scope for automatic insertion of safe casts to invariant type (in a spirit similar to automatic type coercion) that is consistent with each user program.

3. Variance via Flow Analysis

A central feature of our proposed approach is the focus on flow analysis. Variance annotations are used to support the analysis of value flows to capture more accurate generic types, whereby suitable field subtypings (covariance and contravariance) are facilitated where possible.

We highlight the expressiveness of variant types through some more examples in Figure 2. Apart from a generic `Vector`$\langle$`A`$\rangle$ class declaration, we provide a number of static methods to highlight how flow analysis may assist in the formulation of generic types. In the `copyVec` method, the elements from a first vector `Vector`$\langle\oplus$`X`$\rangle$ are copied into a second vector `Vector`$\langle\ominus$`Y`$\rangle$, while a constraint `X<:Y` captures the direction of the value flow.

```
class Vector⟨A⟩ extends Collection⟨A⟩ {
 Vector⟨⊛⟩ | int size() {...}
 Vector⟨⊕X⟩ | X elementAt(int i) {...}
 Vector⟨⊖X⟩ | void setElementAt(X v, int i) {...}
}
void copyVec(Vector⟨⊕X⟩ v, Vector⟨⊖Y⟩ w,
   int start) where X<:Y {
   for(int i=0;i<v.size()&&i+start<w.size();i++)
    w.setElementAt(v.elementAt(i),i+start);
}
void copyNestVec(Vector⟨⊕Vector⟨⊕X⟩⟩v,
   Vector⟨⊖Y⟩ w) where X<:Y {
   int pos=0;
   for(int i=0; i<v.size();i++) {
    Vector⟨⊕Z⟩ s=v.elementAt(i);
    if (pos+s.size()<w.size())
     {copyVec(s,w,pos); pos +=s.size(); }
}}
void clearVec(Vector⟨⊖⊥⟩ v) {
   for(int i=0; i<v.size();i++)
    v.setElementAt(null,i);
}
Vector⟨⊙Z⟩ merge(Vector⟨⊕X⟩ v, Vector⟨⊕Y⟩ w)
   where X<:Z∧Y<:Z
 {...}
Vector⟨⊙Pair⟨⊙X,⊙Z⟩⟩ join(Vector⟨⊕Pair⟨⊕X,⊕Y⟩⟩ v,
    Vector⟨⊕Pair⟨⊕Y,⊕Z⟩⟩ w)
 {...}
void swap(Pair⟨⊙X,⊙Y⟩ p) where X<:Y∧Y<:X {
   T t=p.fst; p.fst=p.snd; p.snd=t;
}
```

Figure 2. Examples with Variant Types

Method `copyNestVec` copies from a nested vector of type `Vector`$\langle\oplus$`Vector`$\langle\oplus$`X`$\rangle\rangle$ into a second vector `Vector`$\langle\ominus$`Y`$\rangle$ with flow constraint `X<:Y`. This code remains highly generic as it uses covariant and contravariant subtypings. The next example shows how we use a special type $\perp$ to indicate that null values will be written into the vector. Given that `Vector`$\langle\ominus\perp\rangle$ is high up in the class hierarchy, this method is rather generic as we can supply *any* vector as its argument.

We also provide method headers for `merge` and `join`. From the type annotation of `merge`, we can tell that values from the first two vectors are retrieved, and then they flow into a new result vector. For the `join` method, we retrieve values from the two vectors `Vector`$\langle\oplus$`Pair`$\langle\oplus$`X`$,\oplus$`Y`$\rangle\rangle$ and `Vector`$\langle\oplus$`Pair`$\langle\oplus$`Y`$,\oplus$`Z`$\rangle\rangle$ before building a new vector `Vector`$\langle\odot$`Pair`$\langle\odot$`X`$,\odot$`Z`$\rangle\rangle$ that is joined

on the Y type. The result's invariant type offers a strong post-condition with read/write capability.

For the swap method, the two fields of a Pair object are swapped. Due to both reading and writing, we require the invariant type $\texttt{Pair}\langle\odot\texttt{X},\odot\texttt{Y}\rangle$ and the expected value flow: $\texttt{X<:Y}\wedge\texttt{Y<:X}$. Based on the flows from the three assignments of the swap body, we may obtain the following constraints: $\odot\texttt{X<:}\oplus\texttt{T}$, $\odot\texttt{Y<:}\oplus\texttt{X}$ and $\odot\texttt{T<:}\oplus\texttt{Y}$, where T is a local type variable (using type rules in Section 5.1). These constraints are simplified to obtain the following collected flow for the method body: $\texttt{X<:T}\wedge\texttt{Y<:X}\wedge\texttt{T<:Y}$. The swap method type checks as the expected flow implies the collected flow:

$\forall\texttt{X,Y.(X<:Y}\wedge\texttt{Y<:X} \implies \exists\texttt{T.(X<:T}\wedge\texttt{Y<:X}\wedge\texttt{T<:Y))}$

Note that the local type variable T is existentially quantified, while type variables X,Y from method parameters are universally quantified.

3.1 Improved Variant Subtyping

Variant parametric type τ consists of a variance α and a type t. Its grammar is introduced in Figure 4. We use variance annotations $\odot,\oplus,\ominus$ and $\circledast$, which correspond to read-write access, read-only access, write-only access, and no-access, respectively. These annotations are ordered by the following relation that is denoted by $<:_\alpha$ but abbreviated to $<:$ below:

$$\odot<:\oplus \quad \odot<:\ominus \quad \oplus<:\circledast \quad \ominus<:\circledast$$

$$\frac{\alpha_1<:\alpha_2 \quad \alpha_2<:\alpha_3}{\alpha_1<:\alpha_3} \quad \alpha<:\alpha$$

A type t is either a type variable v_t, a variant parametric class $c\langle\tau_1,\ldots,\tau_n\rangle$, the bottom type $\bot$ or an intersection type $t\&t$. The bottom type is used to hold the null value.

We allow finite intersections of types through the type operator $\&$. Semantically, $t_1\&t_2$ denotes the set of objects satisfying the interface specification of both t_1 and t_2. In a lattice of type values with partial order defined by class inheritance (through extends) and interface mechanism (through implements), $t_1\&t_2$ defines the greatest lower bound of t_1 and t_2. Our intersection types are similar to the compound types proposed in [4]. Specifically, they can be of the form $[t_1\&]t_2\&...\&t_n[\&W]$, where t_1 is a class, $t_2,...,t_n$ are interfaces, and W is a type variable.

In our system, variant parametric types are used to support flow analysis rather than access controls. As we focus on value flows at each method boundary, we apply variance annotations primarily to fields. The outermost variance of local variables is always $\odot$. For fields, variance annotations are used to support covariant or contravariant subtyping where possible.

The subtyping relations are denoted by $<:_\tau$ and $<:_t$, both abbreviated to $<:$ as follows:

$$\vdash \tau_1<:\tau_2\Rightarrow\psi \qquad \vdash t_1<:t_2\Rightarrow\psi$$

The resulting constraints ψ (see Figure 4 for their grammar) are kept in a disjunctive normal form. Instead of proving each subtyping directly, we collect a set of subtyping constraints ψ via $\tau{-}subtyping$ and $t{-}subtyping$ in Figure 3.

The first four τ-subtyping rules support contravariance, covariance, invariance and bivariance, respectively. The invariant case generates a constraint from the semantical equivalence of the two types ($t_1\equiv t_2$). Unlike the subtyping rule of Igarashi and Viroli [17], our improved mechanism handles two special values in the subtyping hierarchy, namely $\bot$ (for type of null) and *Object* (for top of class hierarchy). These two types are special in that it is always safe to write a null (of $\bot$ type) into any location (even if it has been marked for read-only access), and it is safe to read an *Object* value from any location (even if it has been marked for write-only access). We may also cast any type τ to either $\oplus$*Object* or $\ominus\bot$ as it is always safe to read an object or write a null value. This mechanism is implemented by the last two τ-subtyping rules.

$\tau-$subtyping

$$\frac{\alpha_1<:\ominus \quad \vdash t_2<:t_1\Rightarrow\psi}{\vdash \alpha_1 t_1<:\ominus\, t_2\Rightarrow\psi} \qquad \frac{\alpha_1<:\oplus \quad \vdash t_1<:t_2\Rightarrow\psi}{\vdash \alpha_1 t_1<:\oplus\, t_2\Rightarrow\psi}$$

$$\frac{\vdash t_1\equiv t_2\Rightarrow\psi}{\vdash \odot t_1<:\odot\, t_2\Rightarrow\psi} \qquad \vdash \tau<:\circledast t\Rightarrow\texttt{true}$$

$$\frac{\neg(\alpha_1<:\oplus)}{\vdash \alpha_1 t_1<:\oplus\, \mathit{Object}\Rightarrow\texttt{true}} \qquad \frac{\neg(\alpha_1<:\ominus)}{\vdash \alpha_1 t_1<:\ominus\, \bot\Rightarrow\texttt{true}}$$

t−subtyping

$$\vdash\bot<:t\Rightarrow\texttt{true} \quad \vdash t<:\mathit{Object}\Rightarrow\texttt{true} \quad \vdash t<:t\Rightarrow\texttt{true}$$

$$\frac{\vdash \tau_i<:\tau_i'\Rightarrow\psi_i,\ i=1..n}{\vdash c\langle\tau_i\rangle_{i=1}^n<:c\langle\tau_i'\rangle_{i=1}^n\Rightarrow\bigwedge_{i=1}^n\psi_i} \qquad \frac{\vdash t_1<:t_2\Rightarrow\psi_1 \quad \vdash t_2<:t_3\Rightarrow\psi_2}{\vdash t_1<:t_3\Rightarrow\psi_1\wedge\psi_2}$$

$$\frac{\texttt{class}\ c_1\langle V_i\rangle_{i=1}^m\ \texttt{extends}\ ...c_2\langle\tau_i'\rangle_{i=1}^n... \quad \rho=[V_i\mapsto\tau_i]_{i=1}^m \quad c_2\langle\rho\tau_i'\rangle_{i=1}^n\Rightarrow_p\ c_2\langle\rho\tau_i''\rangle_{i=1}^n}{\vdash c_1\langle\tau_i\rangle_{i=1}^m<:c_2\langle\rho\tau_i''\rangle_{i=1}^n\Rightarrow\texttt{true}}$$

$$\frac{\vdash t<:t_1\Rightarrow\psi_1 \quad \vdash t<:t_2\Rightarrow\psi_2}{\vdash t<:(t_1\&t_2)\Rightarrow\psi_1\wedge\psi_2} \qquad \frac{\vdash t_1<:t\Rightarrow\psi_1 \quad \vdash t_2<:t\Rightarrow\psi_2}{\vdash (t_1\&t_2)<:t\Rightarrow\psi_1\vee\psi_2}$$

$$\frac{t_1=v_t\vee t_2=v_t}{\vdash t_1<:t_2\Rightarrow t_1<:t_2} \qquad \frac{\vdash t_1<:t_2\Rightarrow\psi_1 \quad \vdash t_2<:t_1\Rightarrow\psi_2}{\vdash t_1\equiv t_2\Rightarrow\psi_1\wedge\psi_2}$$

Figure 3. Variant Subtyping

In the second part of Figure 3, the first two t-subtyping rules handle the bottom and top of the hierarchy. Subtyping between types of the same class is decomposed structurally by the fourth rule. The next two rules describe transitivity and the class inheritance relation. The class inheritance rule uses type promotion mechanism that is described later in Section 4.1 Intersection types satisfy the subtyping relations as in [29]. Subtyping relations that contain type variables are not simplified further and preserved in the resulting constraint. Semantic equality ($t_1\equiv t_2$) is given by the last t-subtyping rule. In summary, from the subtyping relations between types, we generate a set of subtyping constraints (on type variables). Note that in the following sections, we will use $\tau_1<:\tau_2$ as an abbreviation for ψ, where $\vdash \tau_1<:\tau_2\Rightarrow\psi$.

4. Core Language

We introduce a core language to ease the formulation of static and dynamic semantics. This language can be viewed as a result of translation from full Java language prior to type checking. For ease of presentation, we omit features that are related to static methods, exception handling, concurrency and inner classes. (Our implementation handles all features of the Java language.)

Our core language is named Variant CoreJava, and summarised in Figure 4. We use the suffix notation y^* to denote a list of (zero or more) distinct syntactic terms that are suitably separated. Both class and interface declarations are supported using the same syntactic grammar term *def*. As with Java, the main difference is that interface definitions do not have fields, and are defined using abstract methods (without body). Furthermore, while we support multiple inheritance, it is of the same restricted kind as that supported by the Java language. Each class may extend from only a single superclass but may implement multiple interfaces. In our language, the declaration $\texttt{class}\ c\langle V^*\rangle\ \texttt{extends}\ gc_1..gc_n$ assumes that gc_1 is a

Programs

$P ::= def^*$

$def ::= \texttt{class}\ c\langle V^*\rangle\ \texttt{extends}\ gc_1..gc_n\ \texttt{where}\ \psi_{inv}\ \{(\pi\ f)^*\ meth^*\}$

$gc ::= c\langle \pi_1, .., \pi_n\rangle$

$meth ::= t \mid t\ mn\texttt{(}(t\ v)^*\texttt{)}\langle v_t^*\rangle\ \texttt{where}\ \psi\ \{e\}$

$w ::= v \mid v.f$

$e ::= \texttt{null} \mid w \mid w = e \mid \{t\ v = e_1;\ e_2\} \mid e_1\ ;\ e_2 \mid \texttt{new}\ c\langle t^*\rangle\texttt{(}v^*\texttt{)} \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ v\ \texttt{do}\ e \mid v_0.mn\texttt{(}v^*\texttt{)}\langle t^*\rangle \mid (t)v \mid \{v_1 = (t)v\ ;\ e\}$

Variant Parametric Types

$\tau ::= \alpha t$

$t ::= v_t \mid c\langle \tau_1, .., \tau_n\rangle \mid t\&t \mid \bot$

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

Incomplete Variant Parametric Types

$\pi ::= V \mid \oslash\, s$

$s ::= c\langle \pi_1, .., \pi_n\rangle \mid s\&s \mid \bot$

Subtyping constraints

$\psi ::= t_1 <: t_2 \mid \psi \wedge \psi \mid \texttt{true}$

Class Invariant

$\psi_{inv} ::= V <:_i c\langle \tau^*\rangle \mid c\langle \tau^*\rangle <:_i V \mid \psi_{inv} \wedge \psi_{inv} \mid \texttt{true}$

Figure 4. Syntax of VARIANT COREJAVA

class while $gc_2..gc_n$ are essentially interfaces (`implements` is also represented by `extends` for easy presentation). Each class declaration captures a class invariant ψ_{inv} that is expected to hold for all newly constructed objects of the class. This is being used to specify suitable class lower and/or upper bounds for type variables. Since our system is based on use-site variance, the class fields types and the arguments of class inheritance have incomplete variance at declaration-site (denoted by π and V). Section 4.1 describes the annotations of class declarations with incomplete variant parametric types.

Each method declaration *meth* contains a constraint ψ which captures the expected value flows for its type variables. It also specifies method type parameters $\langle v_t^*\rangle$ in order to support modular type checking. This set of type variables is automatically inserted by our compiler.

We use an expression-oriented language, where method body is denoted by e. Local variable declaration is supported by block structure of the form: $\{t\ v = e_1; e_2\}$, with e_2 denoting its result. Each object is always built with an invariant type $c\langle\odot t^*\rangle$ via the construct `new` $c\langle t^*\rangle$**(**v^***)**. Our core language also supports a full casting mechanism via $(t)v$, where t can be an arbitrary variant type. In addition, we support a novel cast capture mechanism via $\{v_1 = (t)v\ ;\ e\}$, where t is an invariant type with unknown type variables that may be captured at runtime and used in e. These special features will be described in more detail in Section 6.

For simplicity of presentation, our core language represents primitive types (such as `void`, `bool`) by their corresponding classes (such as `Void`, `Bool`). In our implementation, we handle primitive types directly, as elaborated in Section 9. For soundness reasons, we treat arrays in the same way as other classes (unlike Java 1.5, which assumes arrays to be covariant).

In the subtyping constraints, disjunction is supported internally as it may be generated by subtyping relation for intersection types.

4.1 Class Parameterisation and Inheritance

For class declarations, an important decision is which fields are to be parameterised and how the class inheritance mechanism is to be supported. In general, each class declaration should be written in the following manner:

```
class c1⟨V1..Vn⟩ extends c2⟨π̂1..π̂s⟩ where ψinv {
 π1 f1;
 ...
 πm fm; ...
}
```

where each $\{V_i\}_{i=1}^n$ originates either from the fields of the current class $\{\pi_i\}_{i=1}^m$ or from the arguments of its superclass, $\{\hat{\pi}_i\}_{i=1}^s$. $\{V_i\}_{i=1}^n$ are variables corresponding to types with variance. For instance, the following non-generic declarations of `Cell` and `Pair` classes:

```
class Cell {
 Object fst; ··· }
class Pair extends Cell {
 Object snd; ···
}
```

can be parameterized as:

```
class Cell⟨A⟩ {
 A fst; ··· }
class Pair⟨A,B⟩ extends Cell⟨A⟩ {
 B snd; ···
}
```

The variance of the fields `fst` and `snd` is governed by the variables `A` and `B`. Given the type `Pair`$\langle\oplus$`Int,`$\ominus$`Int`$\rangle$, the field `fst` is covariant and the field `snd` is contravariant.

4.2 Type Promotion

There are some situations where the variance of a class field cannot be specified at use site. In the following example, the variance of the field `sndP` does not have any correspondence in the class parameters `A`,`B`,`C` and remains unknown after instantiation of these parameters.

```
class Triple⟨A,B,C⟩ extends Cell⟨A⟩ {
 Pair⟨B,C⟩ sndP; ···
}
```

The compiler inserts a special variance marker $\oslash$ to represent the unknown variance of field `sndP`:

```
class Triple⟨A,B,C⟩ extends Cell⟨A⟩ {
 ⊘Pair⟨B,C⟩ sndP; ···
}
```

Note that the source program does not contain any variance markers. We use them to explain how incomplete (or unknown) variance of variant parametric types are computed to either $\oplus$ or $\odot$. This process is known as *type promotion* and can be used for incomplete variant parametric types from field declarations and arguments of class inheritance.

The type promotion is defined using the relations

$$\rho \vdash \pi \Rightarrow_p \tau \qquad \rho \vdash s \Rightarrow_p t$$

where ρ is a substitution $[V \mapsto \tau]$ from class declaration parameters V to variant parametric types τ. The types π and s may contain unknown variance $\oslash$. The rules are described in Figure 5.

The second rule promotes the unknown variance $\oslash$ to either $\oplus$ or $\odot$ depending on the predicate *inv*(t) where t is the type obtained after substitution. Predicate *inv*(t) returns `true`, when all

$$\frac{}{\rho \vdash V \Rightarrow_p \rho V} \qquad \frac{\rho \vdash s \Rightarrow_p t \quad \alpha = \texttt{if}\ \mathit{inv}(t)\ \texttt{then}\ \odot\ \texttt{else}\ \oplus}{\rho \vdash \oslash s \Rightarrow_p \alpha t}$$

$$\frac{\rho \vdash \pi_i \Rightarrow_p \tau_i \quad i = 1,n}{\rho \vdash c\langle \pi_1, ..\pi_n\rangle \Rightarrow_p c\langle \tau_1, ..\tau_n\rangle} \qquad \frac{\rho \vdash s_i \Rightarrow_p t_i \quad i = 1,2}{\rho \vdash s_1 \& s_2 \Rightarrow_p t_1 \& t_2}$$

$$\mathit{inv}(\odot t) = \texttt{true} \qquad \frac{\alpha = \oplus \mid \ominus \mid \circledast}{\mathit{inv}(\alpha t) = \texttt{false}} \qquad \mathit{inv}(v_t) = \texttt{true}$$

$$\mathit{inv}(c\langle \tau_1, ..\tau_n\rangle) = \bigwedge_{i=1,n} \mathit{inv}(\tau_i) \qquad \mathit{inv}(t_1 \& t_2) = \bigwedge_{i=1,2} \mathit{inv}(t_i)$$

$$\mathit{inv}(c\langle\rangle) = \texttt{true} \qquad \mathit{inv}(\bot) = \texttt{true}$$

Figure 5. Type Promotion

variances from `t` (if any) are $\odot$ and `false` otherwise. Given `Triple`$\langle\oplus$`Int,`$\oplus$`Int,`$\oplus$`Int`$\rangle$, the type of field `sndP` is computed as follows: $\rho \vdash \oslash$`Pair`$\langle$`B,C`$\rangle \Rightarrow_p \oplus$`Pair`$\langle\oplus$`Int,`$\oplus$`Int`$\rangle$ where $\rho = [$`A` $\mapsto \oplus$`Int`,`B` $\mapsto \oplus$`Int`,`C` $\mapsto \oplus$`Int`$]$. As another example, given `Triple`$\langle\oplus$`Int,`$\odot$`Int,`$\odot$`Int`$\rangle$, the type of field `sndP` is computed as follows: $\rho \vdash \oslash$`Pair`$\langle$`B,C`$\rangle \Rightarrow_p \odot$`Pair`$\langle\odot$`Int,`$\odot$`Int`$\rangle$ where $\rho = [$`A` $\mapsto \oplus$`Int`,`B` $\mapsto \odot$`Int`,`C` $\mapsto \odot$`Int`$]$.

Another application of type promotion is for recursive fields of a class. The recursive field `next` of the class `List` has an incomplete variance $\oslash$ as follows:

```
class List⟨A⟩ extends Object {
 A val;
 ⊘List⟨A⟩ next; ...
}
```

The variance of the field `next` is incomplete at its declaration site and can be promoted to either $\odot$ or $\oplus$, depending on how its underlying type parameter `List`$\langle$`A`$\rangle$ is being instantiated at the use site. For example, when `A` is instantiated to $\ominus$`X`, the variance of the `next` field will be promoted to $\oplus$ via $\rho \vdash \oslash$`List`$\langle$`A`$\rangle \Rightarrow_p \oplus$`List`$\langle\ominus$`X`$\rangle$, where $\rho = [$`A` $\mapsto \ominus$`X`$]$. On the other hand, if `A` is instantiated to $\odot$`X`, then $\rho = [$`A` $\mapsto \odot$`X`$]$ and the variance of the `next` fields is instantiated to $\odot$`X` as follows: $\rho \vdash \oslash$`List`$\langle$`A`$\rangle \Rightarrow_p \odot$`List`$\langle\odot$`X`$\rangle$.

Our type promotion is a refinement of that proposed in [17]. First, we allow promotion to $\odot$ whenever possible while Igarashi and Viroli considered mainly the promotion of nested types with $\oplus$. Second, we consider type promotion for only field access and class inheritance where the outer variance is dependent on the variance of the underlying type. In contrast, Igarashi and Viroli focused on the promotion of nested types of arguments/result for method declarations, which need not be handled in our approach as these types are fully specified in our method declarations.

4.3 Class Invariant

The class invariant ψ_{inv} is used to capture the lower and upper bounds for the parameterised fields of each newly created object of the class. These bounds are of the form $\bigwedge c_1\langle\tau*\rangle <:_i V <:_i c_2\langle\tau*\rangle$. Class invariant may also support F-bounds when variable V occurs in the parameters of classes c_1 and c_2. If unspecified, the default lower and upper bounds are $\bot$ and `Object`, respectively. An upper bound invariant on a write-only field restricts the class of the object that can be written to the field to be subclasses of the bound, and a lower bound invariant on a read-only field restricts the class of the object that can be read from the field to be superclass of the bound.

We use the relation $\Rightarrow_{cinv}$ to reduce bounds from the class invariant to a constraint form: $\vdash [V_i \mapsto \tau_i]\psi_{inv} \Rightarrow_{cinv} \psi$, where τ_i are the current variant types for the class fields. The relation $\Rightarrow_{cinv}$ is defined in Figure 6. Note that this relation invokes the subtyping relations defined in Figure 3.

$$\vdash \circledast t <:_i t_1 \Rightarrow_{cinv} \texttt{true} \qquad \vdash t_1 <:_i \circledast t \Rightarrow_{cinv} \texttt{true}$$

$$\frac{\vdash t <: t_1 \Rightarrow \psi_1 \quad \vdash t_1 <: t \Rightarrow \psi_2}{\vdash \oplus t <:_i t_1 \Rightarrow_{cinv} \psi_1 \vee \psi_2} \qquad \frac{\vdash t <: t_1 \Rightarrow \psi_1 \quad \vdash t_1 <: t \Rightarrow \psi_2}{\vdash t_1 <:_i \ominus t \Rightarrow_{cinv} \psi_1 \vee \psi_2}$$

$$\frac{\alpha = \ominus \mid \odot \quad \vdash t <: t_1 \Rightarrow \psi}{\vdash \alpha t <:_i t_1 \Rightarrow_{cinv} \psi} \qquad \frac{\alpha = \oplus \mid \odot \quad \vdash t_1 <: t \Rightarrow \psi}{\vdash t_1 <:_i \alpha t \Rightarrow_{cinv} \psi}$$

$$\frac{\vdash \psi^i_{inv} \Rightarrow_{cinv} \psi^i}{\vdash \bigwedge \psi^i_{inv} \Rightarrow_{cinv} \bigwedge \psi^i}$$

Figure 6. Class Invariant

To illustrate the use of these bounded invariants, consider a class declaration for `Cell`$\langle$`X`$\rangle$ with an upper bound `X<:Num`. For declarations of the form `Cell`$\langle\ominus$`Int`$\rangle$ and `Cell`$\langle\ominus$`T`$\rangle$, the relation $\Rightarrow_{cinv}$ generates the `Int<:Num` and `T<:Num`, respectively. The first constraint reduces to `true`, while the second constraint contains a type variable and will be checked later for satisfiability. As another example, for `Cell`$\langle\ominus$`Object`$\rangle$ the relation $\Rightarrow_{cinv}$ fails as the upper bound is violated. Correspondingly, for read access, we support `Cell`$\langle\oplus$`Int`$\rangle$ and `Cell`$\langle\oplus$`Object`$\rangle$, but not `Cell`$\langle\oplus$`String`$\rangle$ since no `String` objects can be read from the `Num`-bounded field.

The class invariant is accumulated recursively from all the superclasses, as shown below:

$$[\textsc{cinv}]$$
$$\frac{\texttt{class}\ c\langle V_i\rangle^m_{i=1}\texttt{extends}(c_k\langle\pi_{ik}\rangle^{n_k}_{i=1})^s_{k=1}\texttt{where}\ \psi_{inv}\ \{..\} \in P \quad \rho = [V_i \mapsto \tau_i]^m_{i=1} \quad \rho \vdash c_1\langle\pi_{i1}\rangle^{n_1}_{i=1} \Rightarrow_p t \quad \vdash \rho\psi_{inv} \Rightarrow_{cinv} \psi}{\mathit{cinv}(c_1\langle\tau_i\rangle^m_{i=1}) = \psi \wedge \mathit{cinv}(t)}$$

5. Variant Type System

Variance annotations of programs are used to support flow analysis for more accurate generic types. We verify the flow of values through the following typing relation:

$$\Gamma; Q \vdash e :: \alpha t, \psi$$

The relation is for type checking, and assumes that Γ (type environment), Q (type variables in scope) and αt (type with expected variance) are given while ψ is the collected flow constraint. Syntax-directed rules for various language constructs are given in Figure 7.

Our type system is flow-insensitive as every location (variable, parameter and field) is given a type that never changes. In our type system, each object of type t_1 can be placed in a location of type t_2, provided $t_1 <: t_2$. The type of a location is therefore a particular *type view* of its object. This type view of an object may be changed by upcasting (via assignment or parameter passing) or by downcast operation that is checkable at runtime. The following rule shows how to type check an assignment expression:

$$[\textsc{assign}]$$
$$\frac{\alpha t = \mathit{GetType}(\Gamma, w) \quad \alpha <: \ominus \quad \Gamma; Q \vdash e :: \oplus t, \psi}{\Gamma; Q \vdash w = e :: \oplus\texttt{Void}, \psi}$$

Flow-in or write-only $\ominus$ is mandated on the left-hand side (w) while flow-out or read-only $\oplus$ is mandated on the right-hand side (e). To highlight how these flows are enforced, we present the rule for variable and field access (w stands for either v or $v.f$):

$$\frac{}{\Gamma;Q \vdash \texttt{null} :: \tau, \oplus\bot <: \tau}\ [\textsc{null}]$$

$$\frac{\Gamma' = \Gamma + \{v :: \odot t\} \quad \Gamma;Q \vdash e_1 :: \oplus t, \psi_1 \quad \Gamma';Q \vdash e_2 :: \tau, \psi_2}{\Gamma;Q \vdash \{t\ v = e_1;\ e_2\} :: \tau, \psi_1 \wedge \psi_2}\ [\textsc{local}]$$

$$\frac{\Gamma;Q \vdash e_1 :: \circledast t, \psi_1 \quad \Gamma;Q \vdash e_2 :: \tau, \psi_2}{\Gamma;Q \vdash e_1; e_2 :: \tau, \psi_1 \wedge \psi_2}\ [\textsc{seq}]$$

$$\frac{\Gamma(v) <: \oplus\ \texttt{Bool} \quad \Gamma;Q \vdash e_1 :: \tau, \psi_1 \quad \Gamma;Q \vdash e_2 :: \tau, \psi_2}{\Gamma;Q \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: \tau, \psi_1 \wedge \psi_2}\ [\textsc{cond}]$$

$$\frac{\Gamma(v) <: \oplus\ \texttt{Bool} \quad \Gamma;Q \vdash e :: \tau, \psi}{\Gamma;Q \vdash \texttt{while}\ v\ \texttt{do}\ e :: \oplus \texttt{Void}, \psi}\ [\textsc{while}]$$

$$\frac{\vdash_{def} \mathit{InheritanceOK}(\mathit{def}_i),\ i = 1..n \quad \vdash_{def} \mathit{def}_i,\ i = 1..n}{\vdash_{prg}\ \mathit{def}_{i=1..n}}\ [\textsc{prog}]$$

$$\frac{\rho = [V_j \mapsto t_j]_{j=1}^{k} \quad \tau'_i = \Gamma(v'_i)_{i=0}^{q} \quad \hat{t_0} \mid t\ mn((\hat{t_i}\ v_i)_{i=1}^{q})\langle V_{1..k}\rangle\ \texttt{where}\ \psi_{..} \in \tau'_0 \quad \psi_1 = \bigwedge_{i=0}^{q} \tau'_i <: \rho(\oplus \hat{t_i}) \wedge \rho(\oplus t) <: \tau}{\Gamma;Q \vdash v'_0.mn(v'_1, .., v'_q)\langle t_{1..k}\rangle :: \tau, \psi_1 \wedge \rho\psi}\ [\textsc{call}]$$

$$\frac{c_1 \vdash_{meth} meth_i, i = 1..q \quad vars\{\pi_i\}_{i=1}^{n} \cup (vars \bigcup_{k=1}^{s} \{\hat{\pi}_{ik}\}_{i=1}^{n_k}) \subseteq \{X_i\}_{i=1}^{m}}{\vdash_{def}\ \texttt{class}\ c_1\langle X_i\rangle_{i=1}^{m}\ \texttt{extends}\ (\hat{c}_k\langle\hat{\pi}_{ik}\rangle_{i=1}^{n_k})_{k=1}^{s}\ \texttt{where}\ \psi_{inv}\ \{(\pi_i\ f_i)_{i=1}^{n}\ meth_{i=1..q}\}}\ [\textsc{class}]$$

$$\frac{def = \texttt{class}\ c_1\langle V_i\rangle_{i=1}^{p}\ \texttt{extends}\ c_2\langle\hat{\pi}_i\rangle_{i=1}^{q}..\texttt{where}..\{fd^*\ meth_{1..p}\} \quad (\exists meth \cdot meth \in c_2\langle\hat{\pi}_i\rangle_{i=1}^{q}.. \wedge \texttt{name}(meth) = \texttt{name}(meth_i)) \Rightarrow \vdash \mathit{OverridesOK}(meth_i, meth) \quad i \in 1..p}{\vdash \mathit{InheritanceOK}(def)}\ [\textsc{inhc}]$$

$$\frac{meth_1 = t_0 \mid t\ mn((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \texttt{where}\ \psi_1\ \{e_1\} \quad meth_2 = \hat{t_0} \mid t\ mn((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \texttt{where}\ \psi_2\ \{e_2\} \quad V_L = vars(\hat{t_0}) - vars(t_0) \quad \vdash t_0 <: \hat{t_0} \Rightarrow \psi \quad \exists V_L \cdot (\psi \wedge \psi_2 \implies \psi_1)}{\vdash \mathit{OverridesOK}(meth_1, meth_2)}\ [\textsc{override}]$$

$$\frac{\texttt{class}\ c_1\langle V_i\rangle_{i=1}^{n}\ \texttt{extends}\ c_2\langle\hat{\pi}_i\rangle_{i=1}^{r}..\{(\pi'_i\ f_i)_{i=1}^{m}..\} \quad \rho = [V_i \mapsto \tau_i]_{i=1}^{n} \quad \rho \vdash \pi'_i \Rightarrow_p \tau'_i, i \in 1..m \quad \rho \vdash \hat{\pi}_i \Rightarrow_p \hat{\tau}'_i, i \in 1..r}{\mathit{fields}(c_1\langle\tau_i\rangle_{i=1}^{n}) = [(\tau'_i\ f_i)]_{i=1}^{m} + \mathit{fields}(c_2\langle\hat{\tau}'_i\rangle_{i=1}^{r})}\ [\textsc{fields}]$$

$$\frac{\tau = \Gamma(v)}{\tau = \mathit{GetType}(\Gamma, v)}\ [\textsc{GetType1}]$$

$$\frac{\alpha t = \mathit{GetType}(\Gamma, v) \quad t = c\langle\tau_i\rangle_{i=1}^{n} \quad (\tau f) \in \mathit{fields}(c\langle\tau_i\rangle_{i=1}^{n})}{\tau = \mathit{GetType}(\Gamma, v.f)}\ [\textsc{GetType2}]$$

$$\frac{\tau_1 = \mathit{GetType}(\Gamma, w) \quad \vdash \tau_1 <: \tau \Rightarrow \psi}{\Gamma;Q \vdash w :: \tau, \psi}\ [\textsc{var-field}]$$

Figure 7. Variant Type Rules

To retrieve the types of the variables and class fields, we use the auxiliary [GetType] rules from Figure 7. The current type τ_1 of w is retrieved from the type environment Γ. Further, the rule checks that τ_1 is a subtype of the expected variant type τ. This supports a flow-out from the variable w.

For object creation, we ensure that each object is constructed with an invariant type using $c\,\langle\odot t_i\rangle_{i=1}^{q}$. A type is said to be *invariant* if each variance on its immediate fields is marked with $\odot$. Note that the views of nested fields, namely $t_1, .., t_q$ from $c\,\langle\odot t_i\rangle_{i=1}^{q}$, may still be of variant types. Note that the variance of all class fields (including those which require type promotion) returned by *fields* is $\odot$.

$$\frac{vars\{t_i\}_{i=1}^{q} \subseteq Q \quad t_0 = c\langle\odot t_i\rangle_{i=1}^{q} \quad (\odot t'_i\ \ f_i)_{i=1}^{p} = \mathit{fields}(t_0) \quad \vdash \oplus t_0 <: \tau \Rightarrow \psi_0 \quad \Gamma;Q \vdash v_i :: \oplus t'_i, \psi_i \quad i = 1..p}{\Gamma;Q \vdash \texttt{new}\ c\langle t_i\rangle_{i=1}^{q}(v_1, .., v_p) :: \tau, \bigwedge_{i=0}^{p} \psi_i \wedge \mathit{cinv}(t_0)}\ [\textsc{new}]$$

For the purpose of constructing invariant types, the type variables in $\{t_i\}_{i=1}^{q}$ must be instantiated from Q. The class invariant $\mathit{cinv}(t_0)$ captures the specified upper/lower bounds on fields that must be satisfied for every object of the class. When such fields are updated, we statically ensure that their bounds are never violated. Given an instantiated class type, the rule [FIELDS] returns the variant types of the class fields using type promotion if necessary.

Local variable declaration v is marked for read-write access via $v :: \odot t$ as shown in the rule [LOCAL]. The rule for method call [Call] collects the flow-in for receiver and arguments, flow-out for the result and the method precondition.

5.1 Modular Flow Verification

We design a variant type system that can be verified in a modular fashion. Each method declaration is given suitable variant type annotations for its parameters, result and receiver. A "may" flow constraint ψ is specified at the header of each method declaration. This *may-flow* specification captures all possible flows that may occur in the method's body e. The type checking rule for a method is formalised as follows:

$$\frac{chkRecv(\texttt{cn}, t_0) \quad \Gamma = \{v_i :: \oplus t_i\}_{i=1}^{p} + \{\texttt{this} :: \oplus t_0\} \quad \psi_1 = \psi \wedge \bigwedge_{i=0}^{p} \mathit{cinv}(t_i) \wedge \mathit{cinv}(t) \quad \psi_1 \neq \texttt{false} \quad Q = \{V^*\} \quad vars(\psi) \subseteq Q \quad vars(\Gamma, t) \subseteq Q \quad \Gamma;Q \vdash e :: \oplus t, \psi_2 \quad V_I = vars(\psi_2) - Q \quad \psi_1 \implies \exists V_I \cdot \psi_2}{\texttt{cn} \vdash_{meth}\ t_0 \mid t\ mn((t_i\ v_i)_{i=1}^{p})\langle V^*\rangle\ \texttt{where}\ \psi\ \{e\}}\ [\textsc{method}]$$

We first construct an initial assumed flow constraint ψ_1 that is derived from the declared may-flow specification ψ, class invariants for each parameter, and result $\bigwedge_{i=0}^{p} \mathit{cinv}(t_i) \wedge \mathit{cinv}(t)$, The initial assumed flow constraint must be satisfiable, that is, $\psi_1 \neq \texttt{false}$. Furthermore, we collect the flow constraint of the method body using $\Gamma;Q \vdash e : \oplus t, \psi_2$, where ψ_2 captures all flows that may occur in the method body e. To prove the correctness of each declared flow constraint, we perform a subtype entailment on the flow constraint with V_I as local type variables using: $\psi_1 \implies \exists V_I \cdot \psi_2$. If this entailment holds, we have successfully verified the flow specification of a given method declaration. We also check if t_0, the given type of `this`, is compatible (no stupid cast) with the current class via the predicate $chkRecv(\texttt{cn}, t_0) = \texttt{cn}\langle\odot t^*\rangle <: t_0$.

Method overriding is checked by the [Override] rule. For safe function subtyping, we require each overriding method to have *weaker or equal* flow specification compared to the overwritten method.

5.2 Soundness

The soundness of our type system can be proven by relating to dynamic evaluation semantics of the form:

$$\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$$

where Π and ϖ denote runtime stack and heap, respectively. This evaluation may yield three possible runtime errors, namely $\mathbf{E} =$ **Error-Null** | **Error-Cast** | **Error-Type**. The second error is due to cast operations guarded by runtime checks inserted by the compiler. The third error is due to an object of the wrong type being written into a location with some expected static type. For well-typed programs, this last error can never happen. The progress theorem states that **Error-Type** cannot occur while the type preservation theorem shows that the type of an expression is preserved with each reduction step. We outline the two theorems below; details of proof may be found in Appendices A, B and C.

THEOREM 1 (Progress). *Let Γ be the environment mapping program variables to ground types. If $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$ and $\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$, then either:*

- *e is a value, or*
- *$\langle \Pi, \varpi \rangle [e] \hookrightarrow$* **Error-Null** | **Error-Cast**, *or*
- *there exist Π', ϖ', e' such that $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.*

Note that the type rules are extended to include store typing Σ. $\Gamma; \Sigma; \psi \models \Pi, \varpi$ denotes a consistency relation that relates static and dynamic semantics. The following theorem states the preservation of type during dynamic evaluation.

THEOREM 2 (Preservation). *Let Γ be an environment mapping program variables to ground types. If*

$$\Gamma; \Sigma; Q \vdash e :: \tau, \psi$$
$$\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$$
$$\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \hat{\Pi}, \hat{\varpi} \rangle [\hat{e}]$$

then there exists $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ such that

$$\Gamma - \mathit{diff}(e, \hat{e}) = \hat{\Gamma} - \mathit{diff}(\hat{e}, e)$$
$$\hat{\Sigma} \supseteq \Sigma$$
$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash \hat{e} :: \tau, \hat{\psi}$$
$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q}; \hat{\psi} \wedge \psi \models \hat{\Pi}, \hat{\varpi}.$$

Function $\mathit{diff}(e, e')$ returns a list of local variables that appear in e but not e' .

6. Casting and Cast Capture

While a key goal of a generic type system is to provide precise information to eliminate unnecessary downcasts, there remains always the need for cast operations to support the class subtyping mechanism. Furthermore, the introduction of generics and variance has complicated type casting as these operations must handle type variables and nested variant types. For example, cast operations may target nested types, such as $\texttt{Vector}\langle\odot\texttt{Vector}\langle\oplus\texttt{Num}\rangle\rangle$, or those with type variables, such as $\texttt{Vector}\langle\oplus X\rangle$.

However, existing solutions that support casting in Java 1.5 are restricted in that they use cast checks on the outermost type constructor only [37], and rely on unchecked warnings that may cause runtime errors (*e.g.*, when a cast to type variable occurs). The only system that supports cast operations fully (but for parametric types) was proposed by Viroli and Natali [39]. Their technique can be adapted to handle arbitrary variant types.

In the presence of single inheritance, we can classify each casting relation from t_0 to t into three categories: (1) safe upcast if $\vdash t_0 <: t$, (2) downcast with runtime check if $\vdash t <: t_0$, and (3) stupid cast if $\neg(\vdash t_0 <: t \vee \vdash t <: t_0)$. However, in the presence of multiple inheritance with interfaces, a class and an interface may be unrelated but a valid downcast is still possible if the actual type is a subtype of the two. Though it is possible to identify stupid cast with a more complex test, namely $\neg(\exists X \cdot X \neq \bot \wedge X <: t \wedge X <: t_0)$, we avoid it for simplicity. Instead, we only check to ensure that the type variables used in t have come from Q. Our type rule to support a variant cast operation is given below:

$$\frac{\alpha t_0 = \Gamma(v) \quad \alpha <: \oplus \quad \mathit{vars}(t) \subseteq Q}{\Gamma; Q \vdash (t)v :: \tau, \oplus t <: \tau} [\textbf{CAST}]$$

While casting is used to check a specific type for an object, we often have to deal with objects of unknown types. For example, we may have an object with a static type $\texttt{List}\langle\ominus\texttt{A}\rangle$, and we may be interested to know its actual invariant type $\texttt{List}\langle\odot\texttt{T}\rangle$, where `T` is an unknown type. To help identify the invariant type of a given object, we introduce a *cast capture* construct based on reflection mechanism: $\{v_1 = (t)v\,;\, e\}$ The following rule shows how to type check the capture construct:

$$\frac{\begin{array}{c}\alpha c\langle\tau_i\rangle_{i=1}^n = \Gamma(v) \quad \alpha <: \oplus \quad t = c\langle\odot V_i\rangle_{i=1}^n \quad \{V_i\}_{i=1}^n \cap Q = \{\} \\ \Gamma; Q \vdash v_1 :: \ominus t, \psi_1 \quad Q_1 = Q \cup \{V_i\}_{i=1}^n \quad \Gamma; Q_1 \vdash e :: \tau, \psi_2\end{array}}{\Gamma; Q \vdash \{v_1 = (t)v; e\} :: \tau, \psi_1 \wedge \psi_2} [\textbf{CAPTURE}]$$

Note that t is an invariant type of the form $c\langle\odot V_i\rangle$: c should have the same class type as v, while the *captured type variables* V_i stand for unknown types. Each V_i can be used in the expression e with its flow captured by the collected flow ($\psi_1 \wedge \psi_2$).

The flow of captured type variables should not cause additional restriction or generalization at the method boundary. We next present how the type system ensures the correct use of captured type variables.

The actual type t obtained via reflection is guaranteed to be more precise than `v`'s static type, $\Gamma(v)$. We call this guarantee *reflection assumption*. For each method, a relation $\Gamma \vdash e \Rightarrow_C V_C, \psi_C$ collects captured type variables, V_C, and their reflection assumptions, ψ_C as follows:

$$\frac{\begin{array}{c}\alpha t_0 = \Gamma(v) \quad \vdash t <: t_0 \Rightarrow \psi \\ X = \mathit{vars}(t) \quad \Gamma \vdash e \Rightarrow_C V, \psi_1\end{array}}{\Gamma \vdash \{v_1 = (t)v; e\} \Rightarrow_C V \cup X, \psi \wedge \psi_1}$$

The method judgement is modified to exclude captured type variables V_C from the local type variables V_I. Additionally, the expected flow ψ_1 is strengthened with reflection assumptions ψ_C.

$$\frac{\begin{array}{c}\mathit{chkRecv}(\texttt{cn}, t_0) \quad \Gamma = \{v_i :: \oplus\, t_i\}_{i=1}^p + \{\texttt{this} :: \oplus\, t_0\} \\ \Gamma \vdash e \Rightarrow_C V_C, \psi_C \quad \psi_1 = \psi \wedge \bigwedge_{i=0}^p \mathit{cinv}(t_i) \wedge \mathit{cinv}(t) \wedge \psi_C \\ Q = \{V^*\} \quad \mathit{vars}(\psi) \subseteq Q \quad \mathit{vars}(\Gamma, t) \subseteq Q \quad \psi_1 \neq \texttt{false} \\ \Gamma; Q \vdash e :: \oplus t, \psi_2 \quad V_I = \mathit{vars}(\psi_2) - Q - V_C \quad \psi_1 \Longrightarrow \exists V_I \cdot \psi_2\end{array}}{cn \vdash_{meth} \; t_0 \mid t\; mn((t_i\; v_i)_{i=1}^p)\langle V^*\rangle \;\texttt{where}\; \psi\; \{e\}} [\textbf{METHOD-WITH-CAPTURE}]$$

The proper flow of captured type variables is then ensured by the entailment from the above rule.

6.1 Cast Capture Examples

The cast capture mechanism can also be viewed as a downcast to the object's invariant type. Unknown types that are captured (via reflection) may be used in the program code, as shown in the example below:

```
void addNode(List⟨⊖A⟩ y, B z) where B<:A {
 List⟨⊙S⟩ v; List⟨⊙S⟩ w;
 {v = (List⟨⊙T⟩) y; w = new List⟨T⟩();
 w.val = z ; w.next = v.next ; v.next = w; } }
```

Though we do not know the exact type of y, we can use a cast capture on (List<⊙T>) to obtain its invariant type. Correspondingly, the reflection assumption is A<:T. We use the captured type T to build a List⟨⊙T⟩ node, write z to w.val, and also reconstruct pointers for the linked list in a type-safe and yet generic way. For this example, the initial assumed flow is $\psi_1 \equiv$(B<:A∧A<:T), whereby B<:A is from the flow specification and A<:T is the reflection assumption. This initial assumed flow implies the collected flow constraint $\exists$S·ψ_2, where $\psi_2 \equiv$(S<:T∧T<:S∧B<:S). Hence, modular verification holds for this example.

The same cast capture mechanism may also be used to capture an unknown invariant type, enabling a swap of elements within the same collection – without knowledge of its type. Consider:

```
void swapVec(Vector⟨⊛⟩ v,int i, int j) {
 Vector⟨⊙S⟩ w;
 {w = (Vector⟨⊙T⟩) v;
 S v1 = w.elementAt(i);
 S v2 = w.elementAt(j);
 w.setElementAt(v2,i); w.setElementAt(v1,j);} }
```

Note that input parameter v uses a bivariant type Vector<⊛>, which can be used to support an argument of an arbitrary Vector object. The initial assumed flow is $\psi_1 \equiv$true, while the collected flow is $\exists$S·ψ_2, where $\psi_2 \equiv$(S<:T∧T<:S). Hence, the entailment $\psi_1 \Longrightarrow \exists$S·$\psi_2$ holds.

An example of a method that does not type check is presented below:

```
Vector⟨⊕Y⟩ foo1(Vector⟨⊛⟩ v) {
 Vector⟨⊙S⟩ w; {w = (Vector⟨⊙T⟩) v; w} }
```

The initial assumed flow is $\psi_1 \equiv$true while the collected flow is $\psi_2 \equiv$T<:Y. Note that neither T (captured type variable) nor Y (global type variable) are existentially quantified from ψ_2. The entailment $\psi_1 \Longrightarrow \exists$S·$\psi_2$ does not hold, since the captured type variable T introduces an additional flow at method boundary. As another example, the following definition type checks as the collected flow from the method's body (after elimination of the local type variable S) is $\psi_2 \equiv$true:

```
Vector⟨⊛⟩ foo2(Vector⟨⊛⟩ v) {
 Vector⟨⊙S⟩ w; {w = (Vector⟨⊙T⟩) v; w} }
```

7. Implementation

We built a prototype for our variant parametric type system and carried out initial experiments to validate its feasibility. Our system was built using the Glasgow Haskell compiler [27], with a constraint solver (for handling subtyping constraints) implemented using Constraint Handling Rules (CHR) [13].

Our constraint solver employs a two-step algorithm to prove the non-structural subtype entailment of the form $\forall V_G \cdot (\psi_1 \Longrightarrow \exists V_I \cdot \psi_2)$. Note that ψ_1, ψ_2 are conjunctions of subtyping constraints , while V_G and V_I are sets of type variables. Even though the entailment from the [METHOD] rule may contain disjunctions, it can be reduced to entailments of the above form.

1. We eliminate the local type variables V_I (based on their upper and lower bounds) from ψ_2 to obtain $\psi_2' = \bigwedge_{i=1}^{n} X_i <: Y_i$ using techniques similar to [32, 38].

   To support the language's semantics a local type inference similar to [30, 23] is employed to identify appropriate instantiated types for local type variables or type parameters.

2. The resulting entailment $\forall V_G \cdot (\psi_1 \Longrightarrow \bigwedge_{i=1}^{n} X_i <: Y_i)$ is equivalent to $\bigwedge_{i=1}^{n} (\forall V_G \cdot (\psi_1 \Longrightarrow X_i <: Y_i))$. Each entailment can be proven by contradiction using the falsity of the formula $\forall V_G \cdot (\psi_1 \wedge \mathit{notsub}(X_i, Y_i))$, where $\mathit{notsub}(t_1, t_2)$ represents negation of subtyping relation.

Our constraint solver implements the variant subtyping rules (from Figure 3). Its deduction mechanism detects falsity based on pair of constraints of the form $t_1 <: t_2$ and $\mathit{notsub}(t_1, t_2)$. Our algorithm is a sound approximation of the subtype entailment problem.

The deduction mechanism can be further extended by the techniques of *case analysis* and *inductive proving*. However, from our experience working with large sets of Java library and application codes that have been annotated and checked with variant parametric types, we have yet to encounter real examples which require such extensions.

8. Experimental Results

To test the utility of our flow-based variant type system, we evaluated our prototype on a set of Java applications[1] as used in [10, 14]. These applications make use of library classes from package java.util, which we annotated with our variant parametric types. We counted each method declaration with flow specification, each class declaration with type parameters and each cast capture as a line of annotation. On average, these annotations constituted about 5.5% of the source code, which may be considered a reasonable price to pay for better reuse of type safe generic code. Due to modular type checking, the time needed to verify type-safe generic code was less than one second for each library code and less than 30 seconds for each application code. We expect that the time can be reduced by using a specialised constraint solver. Currently, our prototype is based on a meta constraint handling system written in CHR (which compiled to a Prolog program under IC-Parc's ECLiPSe system [2]).

Library	Prog.	Java 1.4	Java 1.5		VPT	
	Lines	Casts	Casts	Warnings	Casts	Warnings
AbstractList	909	1	1	0	0	0
AbstractSet	162	1	1	0	0	0
ArrayList	623	2	8	9	1	0
HashMap	1103	7	9	20	3	0
HashSet	231	2	4	3	1	0
Hashtable	1154	10	14	31	7	0
LinkedList	814	2	4	5	2	0
Properties	925	8	8	1	0	0
Vector	1062	2	9	9	0	0
Total	6983	35	58	78	14	0

Figure 8. Results for Library Code

Figures 8 and 9 show the experimental results for representative classes from the java.util package and application code (in terms of remaining casts). We counted the number of casts in Java 1.4 code (non-generic), Java 1.5 (annotated with wildcards) and our system (VPT - annotated with variant parametric types). The Java 1.5 compiler could not statically check some operations (especially those related to raw types and casts to type variables), and issued unchecked warnings since these operations cannot be verified by JVM runtime. Therefore, it is the programmer's responsibility to ensure that all unchecked operations are in fact safe.

[1] For more details: www.junit.org, www.cs.princeton.edu/~appel/modern/java/JLex/, www.cs.princeton.edu/~appel/modern/java/CUP/, www.spec.org/osg/jvm98/, vpoker.sourceforge.net, telnetd.sourceforge.net.

Application	Prog. Lines	Java 1.4	Java 1.5		VPT	
		Casts	Casts	Warnings	Casts	Warnings
DB	842	19	1	0	0	0
JUnit	5886	54	30	1	15	0
VPoker	6792	36	8	0	6	0
JLex	7260	69	12	3	0	0
Jess	10639	95	34	0	12	0
TelnetD	11314	46	8	0	6	0
JavaCup	11468	543	98	2	65	0
Total	54201	862	191	6	104	0

Figure 9. Results for Application Code

To summarize, our method can eliminate a significant portion (on average 87.9%) of the casts from non-generic Java 1.4 application code and 45.5% of the casts from wildcard-generic Java 1.5 application code. We have also made improvements for library code by eliminating about 60% casts from non-generic Java 1.4 code and about 75.8% casts from the wildcard-generic Java 1.5 code. Since our system fully supports casting for variant types, we can verify the unsafe operations for which the Java 1.5 compiler generates unchecked warnings. Note that Java 1.5 libraries contain more casts than Java 1.4 libraries do, since Java 1.4 containers are based on `Object` type instead of generic types. As expected, Java 1.4 application code requires more downcasts compared to Java 1.5 code.

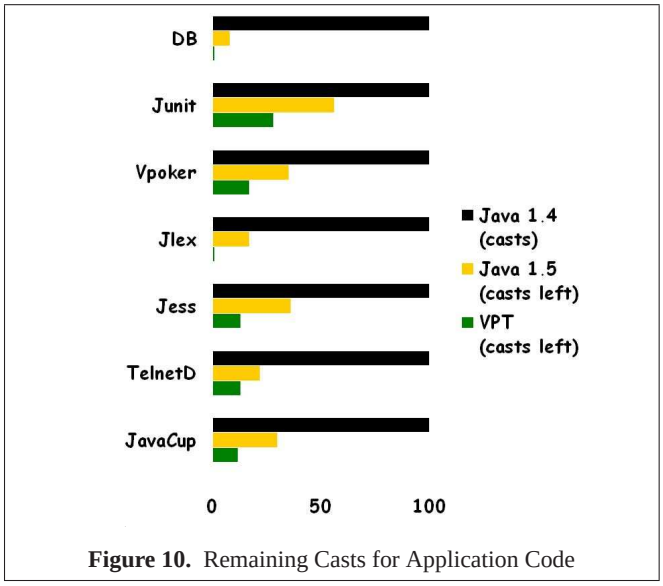

Figure 10. Remaining Casts for Application Code

Figure 10 shows a chart that visualises the percentage of remaining casts in each benchmark written in Java 1.4, Java 1.5 and our VPT. Java 1.4 which contains the casts from the original application code serves as reference.

Note that the casts eliminated using our type system measure the improvement in program safety. Current Java implementation (which translates parametric programs via *type erasure*) would reintroduce casts at the bytecode level. While such re-admitted casts may cause runtime overheads, they are known to be type safe and will never fail at runtime. Obviously, a better solution is to support variant parametric type at the bytecode level and we look forward to this scenario.

9. Extensions

In this section, we present some features omitted in the main presentation for brevity.

The hierarchy of primitive types forms a separate lattice from reference types. Furthermore, it is *not* the case that $\bot <: p <: \mathit{Object}$ for each primitive type p. Due to such differences, primitives are excluded from use as type arguments for generic classes in Java 1.5. Furthermore, the type erasure algorithm for the parametric program will transform each parametric field into an *Object* type for backwards compatibility. This is invalid if primitive types are used as type arguments.

We now show how primitive types can be used as type arguments for generic classes in our system. First, we need to add two constraints to distinguish reference and primitive types, as shown below:

$$\psi ::= \cdots \mid \mathtt{ref}(t) \mid \mathtt{prim}(t)$$

As these two families of types are disjoint, we add the following CHR irrevocable rule:

$$\mathtt{ref}(t) \wedge \mathtt{prim}(t) \quad \Leftrightarrow \quad \mathtt{false}$$

Second, we need to consider primitive types in the new variant subtyping mechanism. In the new subtyping hierarchy, $\circledast t$ denotes any type (reference or primitive) while $\oplus \mathit{Object}$ and $\ominus\bot$ denote only reference types (that are still equivalent, namely $\oplus \mathit{Object} \equiv \ominus\bot$). The subtyping relation is changed accordingly: $\oplus \mathit{Object} <: \circledast t$ still holds while $\circledast t <: \oplus \mathit{Object}$ does not hold anymore. Furthermore, we allow $\bot <: t$ and $t <: \mathit{Object}$ if and only if t is not a primitive type. To support these changes, we modify the main variant subtyping rules from Figure 3 to the following:

$$\frac{\alpha \neq \circledast}{\vdash \alpha t <: \oplus \mathit{Object} \Rightarrow \mathtt{ref}(t)} \qquad \frac{\alpha \neq \circledast}{\vdash \alpha t <: \ominus\bot \Rightarrow \mathtt{ref}(t)}$$

$$\frac{\alpha_1 \neq \circledast \quad \neg(\alpha_1 <: \oplus) \quad \vdash \mathit{Object} <: t_2 \Rightarrow \psi}{\vdash \alpha_1 t_1 <: \oplus t_2 \Rightarrow \psi \wedge \mathtt{ref}(t_1)} \qquad \frac{\alpha_1 \neq \circledast \quad \neg(\alpha_1 <: \ominus) \quad \vdash t_2 <: \bot \Rightarrow \psi}{\vdash \alpha_1 t_1 <: \ominus t_2 \Rightarrow \psi \wedge \mathtt{ref}(t_1)}$$

$$\vdash \bot <: t \Rightarrow \mathtt{ref}(t) \qquad \vdash t <: \mathit{Object} \Rightarrow \mathtt{ref}(t)$$

$$\vdash t <: \bot \Rightarrow t <: \bot \wedge \mathtt{ref}(t) \qquad \vdash \mathit{Object} <: t \Rightarrow \mathit{Object} <: t \wedge \mathtt{ref}(t)$$

Programmers often make use of the `instanceof` test on the runtime type of an object prior to some operations. Due to flow and path insensitivity, the type system is currently unable to take advantage of such runtime testing of types. To help eliminate more cast operations, our compiler translates each program fragment of the form:

```
if v.instanceof(t) then e1 else e2
```

to use a special program construct with fresh $\mathtt{v}_0$ variable:

```
if v.instanceof(t) then let v0::t=v in [v↦v0]e1
else e2
```

This construct is part of our core intermediate language, and it is generated prior to type checking. It is valid on the proviso that any assignment into `v` is a subtype of the more specific `t`. A type rule corresponding to the new language construct is shown below:

$$\frac{\begin{array}{c}[\textsc{let-instanceof}] \\ e_1 \equiv (\mathtt{let}\ v_0 :: t = v\ \mathtt{in}\ e) \\ \Gamma' = \Gamma + \{v_0 :: \odot t\} \quad \Gamma'; Q \vdash e :: \tau, \psi_1 \quad \Gamma; Q \vdash e_2 :: \tau, \psi_2\end{array}}{\Gamma; Q \vdash \mathtt{if}\ v.\mathit{instanceof}(t)\ \mathtt{then}\ e_1 \mathtt{else}\ e_2 :: \tau, \psi_1 \wedge \psi_2}$$

Flow-sensitivity may also cause some loss in type precision (such that some downcasts cannot be statically verified) when the same local variable is used for objects with different variant parametric types. To rectify this, we could use Static Single Assignment (SSA) intermediate form [8] which is known to give better flow-sensitive analysis results. Conversion of programs to SSA form can be handled in a preprocessing step, prior to type checking.

These techniques for incorporating path and flow sensitivity are quite standard, and were also explored in [41].

10. Conclusion

Software reuse has received much research interest for its boost to software development and maintenance productivities. Recently, generic type has become a main thrust in supporting software reuse. In reusing Java code, several works have proposed for refactoring legacy Java programs into those that use generic versions of popular container classes [10, 11, 14, 40].

Other works try to achieve precise Java type inference results in the presence of parametric polymorphism and data polymorphism in order to reduce the redundant cast operations [31, 1, 41]. The precision typically comes at the price of a whole program analysis. Every time when an application code is analysed, the reachable library code must also be re-analysed.

Variant parametric types attempt to increase language expressivity and code reuse by introducing another subtyping scheme, based on the notion of variance. This idea originated from the structured virtual types proposed by Thorup and Torgersen [35]. Their work is the first to link access rights and covariant subtyping to the fields of each use of a class rather than the class itself. Igarashi and Viroli extended this concept to support contra- and bi-variance [17]. They also formalised the variant type system by mapping it into a corresponding *existential type* system [17, 18] for Featherweight Java. While Igarashi and Viroli's design faithfully models the existential type system, it has been found to be too restrictive by the designers of Java 1.5. One improvement made in Java 1.5 is to allow each wildcard type to be opened without a corresponding close operation. This provides more flexibility for writing generic code, but weakens the link to the traditional pack/unpack mechanism of the existential type system. Hence, even though a full-scale language system has been implemented, the soundness of the wildcard type system is still under development (as of [36]). Other than Java, a recently developed language Scala [24] supports variance for parametric polymorphism. In contrast with our approach, Scala uses variance at *declaration-site*. However, an earlier version of Scala has experimented with the use-site variance mechanism that is consistent with the original system of Igarashi and Viroli but without the flexibility of the wildcard capture. This was considered to be too restrictive before the authors abandoned the approach. Recently, generic types of C# [12] were extended with *declaration-site* variance following the design adopted for the language Scala.

Theoretical foundations of the variance have also been studied in the context of typed λ-calculi, where type operators are equipped with a polarity property [6, 33, 9]. These foundations have even been extended to handle higher-order functions, but are closer in nature to declaration-site variance, and have mostly been formalised in only a theoretical setting, without practical implementations.

In our paper, we have proposed a new approach based on flow analysis to support the variant parametric type system. We leverage prior knowledge that has been accumulated for flow analysis and entailment for non-structural subtyping constraints. Palsberg and O'Keefe [25] show the equivalence of flow analysis and non-structural subtyping. Subtype entailment is known to be hard even for simple subtyping constraints. Rehof and Henglein determined the complexity of structural subtype entailment: for simple types, it is coNP-complete [15] and for recursive types it is PSPACE-complete [16]. Furthermore, they showed that non-structural subtype entailment is PSPACE-hard and is conjectured PSPACE-complete [16]. Su et al. [34] show the decidability of the first-order theory of non-structural subtyping with unary function symbols. Algorithms for non-structural subtype entailment (sound, but incomplete) were developed in Pottier [32], Trifonov and Smith [38]. While the decidability of non-structural subtype entailment remains an open problem, we use sound techniques based on these previous algorithms.

Our new approach is practically driven and can give better generic types. We have also augmented it with intersection types to support Java-like multiple (interface) inheritance. We have built a prototype system based on a set of syntax-directed type rules. This prototype is supported by a constraint-solver for variant subtyping, customised using CHR. Furthermore, our system supports full casting for variant types. Through a new cast capture mechanism, we can use reflection to handle objects with unknown types in a type-safe way. Experimental evaluation indicates that more downcasts can be eliminated by our approach, even when it is compared against the state-of-the-art type system from Java 1.5. Our flow-based approach to variant parametric type system is another step towards better genericity for type-safe OO programs.

Acknowledgments

We are grateful to Atsushi Igarashi for clarifying many questions regarding the variant type system. We also thank Alex Aiken, Greg Morrisett and Martin Rinard for providing useful feedback on this work. Shengchao provided useful technical comments on the paper while Hong Yaw implemented the Java-to-CoreJava translator and hand-annotated a suite of Java libraries and application code. We also thank Alexia Leong for her proof-reading efforts. This work is supported by research grant R-252-000-151-112 and a gift from Microsoft Singapore.

References

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26, London, UK, 1995. Springer-Verlag.

[2] IC-Parc at Imperial College. ECLiPSe Constraint Logic Programming. http://www.icparc.ic.ac.uk/eclipse/.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[4] Martin Buchi and Wolfgang Weck. Compound types for Java. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 362–373, New York, NY, USA, 1998. ACM Press.

[5] Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-Bounded polymorphism for object-oriented programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[6] Luca Cardelli. Notes about $F^{\omega}_{<:}$. 1994. Available at http://research.microsoft.com/Users/luca/Notes/FwSub.ps.

[7] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[9] Adriana Compagnoni Dominic Duggan. Subtyping for object type constructors. In *Foundations of Object-Oriented Languages (FOOL 1999)*, 1999.

[10] Alan Donovan, Adam Kiezun, Matthew S. Tschantz, and Michael D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 15–34, New York, NY, USA, 2004. ACM Press.

[11] Dominic Duggan. Modular type-based reverse engineering of parameterized types in Java code. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, pages 97–113, 1999.

[12] Burak Emir, Andrew J. Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[13] Thom Fruhwirth and et al. Constraint Handling Rules. http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/.

[14] Robert Fuhrer, Frank Tip, Adam Kiezun, Julian Dolby, and Markus Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, July 2005.

[15] Fritz Henglein and Jakob Rehof. The complexity of subtype entailment for simple types. In *Proceedings of 12th Symposium on Logic in Computer Science (LICS '97)*, pages 352–361, June 1997.

[16] Fritz Henglein and Jakob Rehof. Constraint automata and the complexity of recursive subtype entailment for simple type. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98*, pages 616–627, 1998.

[17] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 441–469, 2002.

[18] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 2006.

[19] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2001.

[20] Karl Mazurak and Steve Zdancewic. Type inference for Java 5: Wildcards, F-Bounds, and Undecidability. 2006. A note available at http://www.cis.upenn.edu/~stevez/note.html.

[21] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[22] Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin-Lof's Type Theory*. Oxford University Press, 1990.

[23] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 41–53, 2001.

[24] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*, pages 41–57, 2005.

[25] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 367–378, 1995.

[26] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, 1998.

[27] Simon Peyton-Jones and et al. Glasgow Haskell Compiler. http://www.haskell.org/ghc.

[28] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.

[29] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[30] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, 1998.

[31] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA*, pages 324–340, 1994.

[32] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, 1996.

[33] Martin Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universitat Erlangen-Nurnberg, 1997.

[34] Zhendong Su, Alexander Aiken, Joachim Niehren, Tim Priesnitz, and Ralf Treinen. The first-order theory of subtyping constraints. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–216, 2002.

[35] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP'99 - Object-Oriented Programming, 13th European Conference*, pages 186–204, 1999.

[36] Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL 2005)*, Long Beach, CA, January 2005.

[37] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, and Neal M. Gafter. Adding Wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, 2004.

[38] Valery Trifonov and Scott F. Smith. Subtyping constrained types. In *Static Analysis, Third International Symposium, SAS'96*, pages 349–365, 1996.

[39] Mirko Viroli and Antonio Natali. Parametric polymorphism in java: an approach to translation based on reflective features. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000)*, pages 146–165, 2000.

[40] Daniel von Dincklage and Amer Diwan. Converting Java classes to use generics. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–14, 2004.

[41] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 99–117, 2001.

A. Dynamic Semantics

The dynamic operational semantics of Variant CoreJava is described in small steps. Notations used are defined as follows.

Locations :	ι	$\in$	*Location*
Primitives :	k	$\in$	$prim =$ `int` $\uplus$ `bool` $\uplus$ `float` $\uplus$ `null` $\uplus$ `void`
Values :	δ, ν	$\in$	$Value = (TyPrim \times prim) \uplus Location$
Subs :	μ, ρ	$\in$	$Subs = TVar \rightarrow_{fin} Type$
Store :	ϖ	$\in$	$Store = Location \rightarrow_{fin} ObjVal$
Variable Env :	Π	$\in$	$VEnv = Var \rightarrow_{fin} Value$
Object values :	η	$\in$	$ObjVal = Type \times (Fd \rightarrow_{fin} Value)$
Type :	t	$\in$	*Type*

TyPrim consists of primitive types. A type t maintained at runtime does not contain any variant information. If need be, it will be treated as one with invariant annotation $\odot$. A runtime environment Π is a finite map from program variables to their associated values.

A value can be a location referencing an object or a pair containing a primitive value and a primitive type.

A runtime store ϖ is a finite map from locations to object values. An object value is comprised of its type and its field values. We write $\eta.f$ to denote the value of the field f of an object η. When the object is referred by its location ι, we also write $\iota.f$ to refer to the value of its field f.

We overload the function *type* to accept (1) primitive value and return the primitive type; (2) location and return the type of the dereferenced object; (3) object and return the object type; and (4) object field and return the field type.

The variable environment Π is such a stackable mapping. We write $\Pi[\nu/v]$ to denote an update of the value of the latest variable v in Π to ν. We write $\Pi + \{v \mapsto \nu\}$ to denote an extension of Π to include a binding of ν to v, while $\Pi - \{v^*\}$ removes a subset of the mappings. Similar notations are used for the update and enhancement of object values and stores. In the case of store, we also provide an abbreviated notation $\varpi[\nu/\iota.f] =_{df}$ $\texttt{let}\ (t,\xi) = \varpi(\iota)\ \texttt{in}\ \varpi[(t, \xi[\nu/f])/\iota]$. Given an object value, $\eta = (t, \xi)$, we have $\mathit{Flds}(\eta) =_{df} \xi$.

We require some intermediate expressions for the dynamic semantics to follow through. Our syntax is thus extended from the original expression syntax as follows:

$$e ::= \cdots \mid \eta \mid \iota \mid \nu \mid \texttt{ret}_\texttt{d}(v^*, e) \mid \texttt{ret}_\texttt{m}(Q, v^*, \tau, e)$$

The expression $\texttt{ret}_\texttt{d}(v^*, e)$ is used to capture the result of evaluating a local block, and $\texttt{ret}_\texttt{m}(Q, v^*, \tau, e)$ captures the result of method invocation. The set of variables v^* occurring in both result structures contain the local names and method parameters when entering local body and method body respectively. They are dropped at the end of the local/method body's evaluation. τ captures the type of the result of method invocation, whereas Q captures the set of type variables declared in the method header. Q is an instrument used to facilitate our soundness proof.

The dynamic evaluation rules are of the following form.

$$\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$$

We shall formulate the rules using an exception-style semantics with three possible errors, namely

$$\mathbf{E} = \textbf{Error-Null} \mid \textbf{Error-Cast} \mid \textbf{Error-Type}.$$

Whenever one such error is raised, the evaluation aborts. This error occurrence can be stated using $\langle \Pi, \varpi \rangle [e] \hookrightarrow \mathbf{E}$. The small-step dynamic call-by-name semantics is formalised in Fig 11, together with some auxiliary functions in Fig 12.

B. Soundness of Type System

Before formulating the soundness, we extend the static semantics of the language to include those intermediate expressions given in Sec A. In the process, we require introduction of a *store typing* to describe the type of each location. This ensures that objects created in the store during run-time are type-wise consistent with that captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics. In our case, it is denoted by:

$$\Sigma \in \mathit{StoreType} = \mathit{Location} \rightarrow_{\mathit{fin}} \mathit{Type}$$

Judgements in the static semantics will be extended with store typing, as follows:

$$\Gamma; \Sigma; Q \vdash e :: \tau, \psi$$

$$\frac{[\textbf{ELF}_\textbf{m}]\quad v^* \subseteq \mathit{dom}(\Gamma) \quad Q' \subseteq Q \quad \Gamma; \Sigma; Q \vdash e :: \tau, \psi \quad \vdash \tau <: \tau_1 \Rightarrow \psi_1}{\Gamma; \Sigma; Q \vdash \texttt{ret}_\texttt{m}(Q', v^*, \tau, e) :: \tau_1, \psi \wedge \psi_1}$$

$$\frac{[\textbf{ELF}_\textbf{d}]\quad \Gamma; \Sigma; Q \vdash e :: \tau, \psi}{\Gamma; \Sigma; Q \vdash \texttt{ret}_\texttt{d}(v^*, e) :: \tau, \psi} \qquad \frac{[\textbf{LOC}]\quad \tau = \Sigma(\iota) \quad \vdash \tau <: \tau_1 \Rightarrow \psi}{\Gamma; \Sigma; Q \vdash \iota :: \tau_1, \psi}$$

$$\frac{[\textbf{OBJ}]\quad (t, \xi) = \eta \quad \vdash \odot\, t <: \tau \Rightarrow \psi}{\Gamma; \Sigma; Q \vdash \eta :: \tau, \psi} \qquad \frac{[\textbf{VALUE}]\quad \vdash \odot\, t <: \tau \Rightarrow \psi}{\Gamma; \Sigma; Q \vdash (t, \delta) :: \tau, \psi}$$

Figure 13. Type Rules for Intermediates

The static semantics for these intermediate expressions is shown in Figure 13.

The soundness of our static semantics relies on the following consistency relationship between the static and dynamic semantics, defined as follows:

$$\frac{\begin{array}{c}\mathit{dom}(\Pi) = \mathit{dom}(\Gamma) \quad \mathit{dom}(\varpi) = \mathit{dom}(\Sigma) \quad V_L = \mathit{vars}(\psi) - Q \\ \forall v \in \mathit{dom}(\Pi) \cdot \forall \rho_1 \in \mathit{Subs} \cdot \exists \rho_L \in \mathit{Subs} \cdot \\ (\mathit{dom}(\rho_L) = V_L \wedge \rho = \rho_1 \circ \rho_L \wedge\ (\rho(\psi) \Rightarrow \\ (\Pi(v) \in \mathit{prim} \Rightarrow \mathit{type}(\Pi(v)) <: \rho(\Gamma(v))) \wedge \\ (\Pi(v) \in \mathit{Location} \Rightarrow \\ \mathit{type}(\varpi(\Pi(v))) <: \rho(\Gamma(v))))))\end{array}}{\Gamma; \Sigma; Q; \psi \models \Pi, \varpi}$$

In the above relation, ρ_L is a ground substitution of local type variables occurring in the constraint ψ, and the composition of substitutions is recursively defined as: $(\rho_1 \circ \rho_2)(v) = \mathit{if}\ (v \in \mathit{dom})\ \mathit{then}\ \rho_2(v)\ \mathit{else}\ \rho_1(v)$.

The following theorem states the progress of well-typed expressions.

THEOREM 1 (PROGRESS) *Let Γ be an environment mapping program variables to ground types. If $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$ and $\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$, then either*

- *e is a value, or*
- *$\langle \Pi, \varpi \rangle [e] \hookrightarrow$ **Error-Null** | **Error-Cast**, or*
- *there exist Π', ϖ', e' such that $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.*

A proof of Theorem 1 can be found in Appendix C.1.

The next theorem states that each well-typed expression preserves its type under reduction with a runtime environment and a store which are consistent with the compile-time counterparts.

THEOREM 2 (PRESERVATION) *Let Γ be an environment mapping program variables to ground types. If*

$$\Gamma; \Sigma; Q \vdash e :: \tau, \psi$$
$$\Gamma; \Sigma; Q; \psi \models \Pi, \varpi$$
$$\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \hat{\Pi}, \hat{\varpi} \rangle [\hat{e}]$$

then there exists $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ such that

$$\Gamma - \mathit{diff}(e, \hat{e}) = \hat{\Gamma} - \mathit{diff}(\hat{e}, e)$$
$$\hat{\Sigma} \supseteq \Sigma$$
$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash \hat{e} :: \tau, \hat{\psi}$$
$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q}; \hat{\psi} \wedge \psi \models \hat{\Pi}, \hat{\varpi}.$$

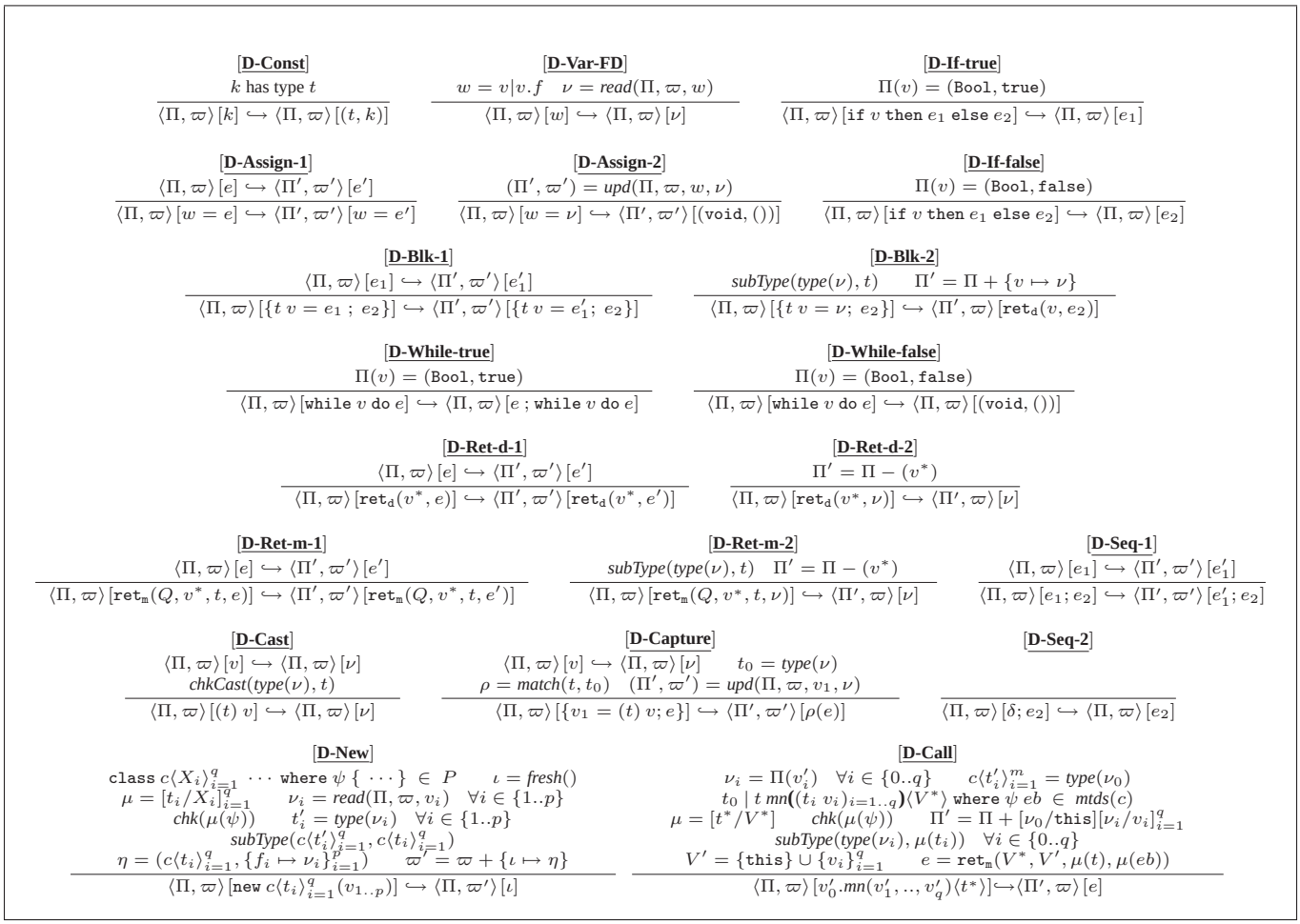

Figure 11. Dynamic Semantics

```
read(Π, ϖ, v) = Π(v);
read(Π, ϖ, v.f) =
    ι = Π(v);
    if ϖ(ι) = null throw Error-Null;
    ϖ(ι).f;

chk(φ) =
    if ¬(⊢ φ) throw Error-Type;
    true;

chkCast(t1, t2) =
    if ¬(⊢ t1<:t2) throw Error-Cast;
    true;

upd(Π, ϖ, v, νs) =
    ν = Π(v);
    if ¬(⊢ type(νs) <: type(ν))
        throw Error-Type;
    (Π[νs/v], ϖ);
upd(Π, ϖ, v.f, νs) =
    ι = Π(v);
    if ϖ(ι) = null throw Error-Null;
    νf = ϖ(ι).f;
    if ¬(⊢ type(νs) <: type(νf)) throw Error-Type;
    (Π, ϖ[νs/ϖ(ι).f]);

subType(t1, t2) =
    if ¬(⊢ t1<:t2) throw Error-Type;
    true;

match(tv, t) = [t/tv];
match(c⟨t*v⟩, c⟨t*⟩) = [t*/t*v];
match(t′, t) = throw Error-Type;
```

Figure 12. Dynamic Semantics:Auxiliary Definitions

Function *diff*(e, e') returns a list of local variables that appears in e but not e' :

$$
\begin{array}{lllll}
\mathit{diff}(e,e') & =_{df} & \mathit{let} & \mathit{lst} & = & \mathit{local}(e) \\
 & & & \mathit{lst1} & = & \mathit{local}(e') \\
 & & & n & = & \mathit{length}(\mathit{lst}) - \mathit{length}(\mathit{lst1}) \\
 & & \mathit{in} & (\mathit{take}(n,\mathit{lst}) \lhd n \geq 0 \rhd [\,]) & & \\
\mathit{take}(n,\mathit{lst}) & =_{df} & ([\,] \lhd n \leq 0 \rhd [\mathit{head}(\mathit{lst})] +\!\!+ \mathit{take}(n-1, \mathit{tail}(\mathit{lst}))) & & & \\
x \lhd b \rhd y & =_{df} & \mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y & & &
\end{array}
$$

Function *local*(e) returns a list of sets of local variables. It is defined as follows:

$$
\begin{array}{lll}
\mathit{local}(e) =_{df} \mathit{case}\ e\ \mathit{of} & & \\
\quad \mathtt{ret_m}(Q, v^*, \tau, e) & \rightarrow & \mathit{local}(e) +\!\!+ [\{v^*\}] \\
\quad \mathtt{ret_d}(v^*, e) & \rightarrow & \mathit{local}(e) +\!\!+ [\{v^*\}] \\
\quad w = e & \rightarrow & \mathit{local}(e) \\
\quad (t\ v = e_1; e_2) & \rightarrow & \mathit{local}(e_1) \\
\quad \mathit{otherwise} & \rightarrow & \emptyset
\end{array}
$$

Note that $\Gamma - [\,] =_{df} \Gamma$, $\Gamma - ([s] +\!\!+ S) =_{df} (\Gamma - s) - S$. The proof is simple: by induction over the depth of type derivation of expression e. A proof of Theorem 2 can be found in Appendix C.2.

C. Proofs of Theorems

C.1 Proof of Theorem 1 (Progress)

By induction over the depth of type derivation for expression e.

Cases [NULL, VOID, VALUE, LOC, OBJ]. Trivial.

Case [VAR–FIELD]. We deal with expression w. As $w = v \mid v.f$ is well-typed, the evaluation rule [**D-Var-FD**] is followed, the evaluation either reports an **Error-Null** or advances one step yielding a value.

Case [ASSIGN]. We deal with expression $w = e$. From type rule, we have $\Gamma; \Sigma; Q \vdash e :: \oplus\, t, \psi$. By induction hypothesis, either (i) e is a value ν, or (ii) $\langle \Pi, \varpi \rangle [e] \hookrightarrow \mathit{Error}$, or (iii) $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.

Subcase (i): Let the runtime type of ν be $\hat{t}$, and that of w be t_1. Then, we have $\odot\, \hat{t} <: \oplus\ t$ and $\odot\, t_1 <: \ominus\ t$, which implies $\hat{t} <: t <: t_1$. Hence, the *upd* function at [**D-Assign-2**] will not throw **Error-Type** exception, and proceed to update the runtime environment Π or the runtime store, as described in [**D-Assign-2**].

Subcase (ii): This will result in the execution of $\langle \Pi, \varpi \rangle [w = e]$ aborted with *Error*.

Subcase (iii): This will result in the execution of the assignment to reach $\langle \Pi, \varpi \rangle [w = e']$, via [**D-Assign-1**].

Case [SEQ]. We have $\Gamma; \Sigma; Q \vdash e_1 :: \circledast\, t, \psi$. By induction hypothesis, either (i) e_1 is a value ν, or (ii) $\langle \Pi, \varpi \rangle [e_1] \hookrightarrow \mathit{Error}$, or (iii) $\langle \Pi, \varpi \rangle [e_1] \hookrightarrow \langle \Pi', \varpi' \rangle [e_1']$.

Subcase (i): The execution proceeds to reach $\langle \Pi, \varpi \rangle [e_2]$ unconditionally, according to [**D-Seq-2**].

Subcase (ii): The execution will be aborted with *Error* exception.

Subcase (iii): The execution proceeds to reach $\langle \Pi', \varpi' \rangle [e_1'; e_2]$, according to [**D-Seq-1**].

Case [LOCAL]. Given that $\Gamma; \Sigma; Q \vdash \{t\ v\ =\ e_1\ ;\ e_2\} :: \tau, \psi_1 \wedge \psi_2$. We have $\Gamma; \Sigma; Q \vdash e_1 :: \oplus\, t, \psi_1$. By induction hypothesis, either (i) e_1 is a value ν, or (ii) $\langle \Pi, \varpi \rangle [e_1] \hookrightarrow \mathit{Error}$, or (iii) $\langle \Pi, \varpi \rangle [e_1] \hookrightarrow \langle \Pi', \varpi' \rangle [e_1']$.

Subcase (i): Let the runtime type of ν be $\hat{t}_0$ and the runtime type of v be $\hat{t}$. As the consistency relation holds between the static and the dynamic semantics, we have for all ground substitution ρ, $\vdash \rho(\psi_1) \Rightarrow \hat{t} = \rho(t)$. Since $\vdash \rho(\psi_1) \Rightarrow \odot \hat{t}_0 <: \oplus\, t$, *subType*(*type*$(\nu), \hat{t}$) = *subType*$(\hat{t}_0, \hat{t})$ = `true`. Hence, the execution will proceed to the state $\langle \Pi', \varpi \rangle [\mathtt{ret_d}(v, e_2)]$ according to [**D-Blk-2**].

Subcase (ii). The execution will throw the corresponding *Error* exception.

Subcase (iii). The execution will proceed to $\langle \Pi', \varpi' \rangle [\{t\ v\ = e_1'; e_2\}]$ according to [**D-Blk-1**].

Case [NEW]. Given $\Gamma; \Sigma; Q \vdash \mathtt{new}\ c\langle t_i \rangle_{i=1}^{q}(v_1, .., v_p) :: \tau, \psi$, let $\hat{t}_i$ (for all $i = 1..q$) and $\hat{t}_{v_i}$ (for all $i = 1..p$) be the runtime types of type arguments and value arguments to `new`. Then we have, for all ground substitution ρ, $\vdash \rho(\psi) \Rightarrow \wedge_{i=1}^{q}(\hat{t}_i = \rho(t_i))$ and $\vdash \rho(\psi) \Rightarrow \wedge_{i=1}^{p}(\hat{t}_{v_i} <: \rho(\Gamma(v_i)))$. Furthermore, $\vdash \rho(\psi) \Rightarrow \rho(\Gamma(v_i)) <: t_i'$, for all i. Hence, both calls to *chk* and *subType* at runtime do not fail, and the execution proceeds to the state $\langle \Pi, \varpi' \rangle [\iota]$, where ι is the location referencing the new object.

Case [COND]. Given $\Gamma; \Sigma; Q \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :: \tau, \psi$ and $\Gamma(v) <: \oplus\ \mathtt{Bool}$, the runtime value of v will either be `true`, `false`, or null $()$. In the first two subcases, the execution proceeds to next state according to the rules [**D-If-true**] and [**D-If-false**] respectively. In the last subcase, there is no corresponding dynamic rule, and exception **Error-Null** will be thrown.

Case [WHILE]. Given $\Gamma; \Sigma; Q \vdash \mathtt{while}\ v\ \mathtt{do}\ e\ :: \tau, \psi$ and $\Gamma(v) <: \oplus\ \mathtt{Bool}$, the runtime value of v will either be `true`, `false`, or null $()$. In the first two subcases, the execution proceeds to next state according to the rules [**D-While-true**] and [**D-While-false**] respectively. In the last subcase, there is no corresponding dynamic rule, and exception **Error-Null** will be thrown.

Case [$\mathbf{ELF_d}$, $\mathbf{ELF_m}$]. We have $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$ as the premise of the static semantics. By induction hypothesis, either (i) e is a value ν, or (ii) $\langle \Pi, \varpi \rangle [e]$ produces *Error*, or (iii) $\langle \Pi, \varpi \rangle [e] \hookrightarrow \langle \Pi', \varpi' \rangle [e']$.

Subcase (i): Let the runtime type of ν be $\hat{t}_\nu$ and that of return value be $\hat{t}$ then for all ground substitution ρ we have $\vdash \rho(\psi) \Rightarrow \rho(\tau) = \odot \hat{t}$. Also, we have $\vdash \rho(\psi) \Rightarrow \hat{t}_\nu <: \rho(\tau)$. Hence, the call to *subType* in the rule [**D-Ret-2**] returns `true`, and the execution proceeds to $\langle \Pi', \varpi \rangle [\nu]$ accordingly.

Subcase (ii): The execution will throw the corresponding *Error* exception, as no rule applies.

Subcase (iii): The execution step to the new state following rule [**D-Ret-1**].

Case [CAST]. Any type mismatch during cast will be captured by *chkCast* and **Error-Cast** exception will be thrown. Otherwise, casting will succeeds and the execution proceeds to the next state $\langle \Pi, \varpi \rangle [(t, \iota)]$.

Case [CAPTURE]. We have $\Gamma; \Sigma; Q \vdash \{v_1 = (t)v; e\} :: \tau, \psi_1 \wedge \psi_2$. From its premise, we have $t = c\langle \odot\, V_i \rangle_{i=1}^{n}$. Executing the expression v either yields an *Error* exception or returns a value ν. We consider the case where ν is returned. Let t_0 be the type of ν as declared in the runtime environment. The use of flow symbol $\odot$ in t implies that *match*(t, t_0) succeeds and produces ρ only when $\rho(t) = t_0$. Hence, by rule [**D-Capture**], the execution proceeds to the state $\langle \Pi', \varpi' \rangle [\rho e]$. Updating of v_1 does not fail, similar with [ASSIGN].

Case [CALL]. Given $\Gamma; \Sigma; Q \vdash v_0'.\mathit{mn}(v_1', .., v_q')\langle t^* \rangle : \tau, \psi$. Let the runtime type arguments be $\langle \hat{t}^* \rangle$ and the value arguments have type $\hat{t}_{v_i'}$ for $i = 0..q$. Also, the ground substitution μ in [**D-Call**] is an instance of ρ in [CALL], which makes ψ true. Thus, we have, $\vdash \mu(\psi) \Rightarrow \hat{t}_{v_i'} <: \mu(\tau_i')$, $i = 0..q$, and $\vdash \hat{t}_0 <: \mu(t_0)$. Hence, the call to *subType* in [**D-Call**] yields true,

and the execution proceeds to the state $\langle \Pi, \varpi \rangle [e]$ according to [**D-Call**]. □

C.2 Proof of Theorem 2 (Preservation)

The proof for Theorem 2 requires several lemmas.

LEMMA 3 (Type Substitution). *If* $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$*, then for all substitution* ρ *such that* $\vdash \rho(\psi)$*, we have* $\rho(\Gamma); \rho(\Sigma); Q \vdash \rho(e) :: \rho(\tau), \rho(\psi)$.

The proof is by induction on a derivation of $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$.

The next lemma, called *assumption weakening lemma*, states that the static judgment remains valid despite a variation of its assumption. This assumes the store type Σ to have unbounded mapping of locations to types. However, the type environment Γ takes the form of stackable mapping between variables and types, and is allowed to grow (by pushing in new mappings) and shrink (by popping out mappings from stack). The lemma states that such change to type environment preserves the type judgment, if the change are properly constrained.

LEMMA 4 (Assumption Weakening). *Given that the following judgment holds:*

$$\Gamma; \Sigma; Q \vdash e :: \tau, \psi$$

Let $\hat{\Gamma}$, $\hat{\Sigma}$ *and* $\hat{Q}$ *be such that:*

$$\begin{gathered} \mathit{vars}(e) \subseteq \mathit{dom}(\Gamma) \cap \mathit{dom}(\hat{\Gamma}) \\ Q \subseteq \hat{Q} \vee \hat{Q} \subseteq Q \\ \mathit{vars}(\psi) - Q = \mathit{vars}(\psi) - \hat{Q} \\ \exists v^* \cdot (\Gamma - \{v^*\} = \hat{\Gamma}) \vee (\hat{\Gamma} - \{v^*\} = \Gamma) \\ \hat{\Sigma} \supseteq \Sigma \end{gathered}$$

Then, there exists $\hat{\psi}$ *such that* $\vdash \hat{\psi} \Leftrightarrow \psi$ *and*

$$\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash e :: \tau, \hat{\psi}$$

The call $\mathit{vars}(e)$ *returns all program variables occurring in* e*, whereas* $\mathit{vars}(\psi)$ *returns all (type) variables occurring in* ψ.

Proof of Lemma 4: By structural induction on the static semantics of the form $\Gamma; \Sigma; Q \vdash e :: \tau, \psi$. For any $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$, we say that they satisfy the premises of the Lemma if the following holds:

$$\begin{gathered} \mathit{vars}(e) \subseteq \mathit{dom}(\Gamma) \cap \mathit{dom}(\hat{\Gamma}) \\ Q \subseteq \hat{Q} \vee \hat{Q} \subseteq Q \\ \mathit{vars}(\psi) - Q = \mathit{vars}(\psi) - \hat{Q} \\ \exists v^* \cdot (\Gamma - \{v^*\} = \hat{\Gamma}) \vee (\hat{\Gamma} - \{v^*\} = \Gamma) \\ \hat{\Sigma} \supseteq \Sigma \end{gathered}$$

Cases [NULL, VOID, LOC, OBJ, VALUE]. Trivial.

Case [VAR–FIELD]. We deal with expression w, where $w = v \mid v.f$. For any $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ satisfying the premise of the lemma, we see that $\Gamma(v) = \hat{\Gamma}(v)$. Hence, $\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash w : \tau, \psi$.

Case [ASSIGN]. We deal with expression $w = e$. We have $\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash e :: \oplus t, \hat{\psi}$ for $\alpha t{=}\mathit{GetType}(\hat{\Gamma}, w){=}\mathit{GetType}(\Gamma, w)$. The desired result is then derived by induction hypothesis.

Cases [LOCAL, SEQ, COND, WHILE, CAST, CAPTURE, $\mathbf{ELF_d}$, $\mathbf{ELF_m}$]. By induction hypothesis.

Case [NEW]. The result holds because $\Gamma(v_i) = \hat{\Gamma}(v_i)$, for all $i = 1..p$.

Cases [CALL]. The result holds because $\Gamma(v_i') = \hat{\Gamma}(v_i')$ for all $i = 1..q$. □

Proof of Theorem 2 This can be proven by induction over the depth of type derivation of expression e.

Cases [NULL, VOID, LOC, OBJ, VALUE]. Vacuously true.

Case [VAR–FD]. This can be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively.

Case [ASSIGN]. There are two rules by which one-step derivation can be applied:

Subcase [**D-Assign-1**]: By induction hypothesis, there exists Γ', Σ' and Q' such that $\Gamma'; \Sigma'; Q' \vdash e' :: \oplus t', \psi'$ and which satisfies the premise of the theorem. Since $\oplus t' <: \oplus t$, we thus have $\Gamma'; \Sigma'; Q' \vdash e' :: \oplus t, \psi' \wedge \psi''$, where $\vdash \oplus t' <: \oplus t \Rightarrow \psi''$. The desired result can then be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ', Σ' and Q' respectively.

Subcase [**D-Assign-2**]: Consider the assignment to a variable v. Given that $\mathit{upd}(\Pi, \varpi, w, \nu)$ returns successfully (Π', ϖ'), it must be the case that $\mathit{type}(\nu) <: \mathit{type}(\Pi'(v))$. The desired result can then be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively. Similar argument applies to the assignment to a field.

Case [SEQ]. There are two rules by which one-step derivation can be applied:

Subcase [**D-Seq-1**]: By induction hypothesis, there exists Γ', Σ' and Q' that establishes the consistency relation at the hypothesis. We elect $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to be Γ', Σ' and Q' respectively to obtain the desired result.

Subcase [**D-Seq-2**]: By setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to be Γ, Σ and Q respectively.

Case [COND]. There are two rules by which one-step derivation can be applied: [**D-If-True**], [**D-If-False**]. Both can be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively.

Case [WHILE]. Similar as the argument for case [COND].

Case [LOCAL]. There are two rules to consider:

Subcase [**D-Blk-1**]: By induction hypothesis.

Subcase [**D-Blk-2**]: Since $\mathit{subType}(\mathit{type}(\nu), t)$, Γ' and Σ used in [LOCAL] remain consistent with Π' and ϖ in this subcase. We set $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ', Σ and Q respectively.

Case [CAST]. This can be proven by setting $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively.

Case [CAPTURE]. The argument for one-step derivation [**D-Capture**] is similar to that for case [**D-Assign-2**], except for the assignment of runtime type information of ν to the type variables occurring in t. This assignment proceeds successfully because of the premise of [CAPTURE]. We set $\hat{\Gamma}$, $\hat{\Sigma}$ and $\hat{Q}$ to Γ, Σ and Q respectively. It suffices to show that $\hat{\Gamma}; \hat{\Sigma}; \hat{Q} \vdash \rho(e) :: \tau, \hat{\psi}$. This is true by applying Type Substitution Lemma to the following premise of [CAPTURE]: $\Gamma; \Sigma; Q \vdash e :: \tau, \psi_2$.

Case [NEW]. We set $\hat{\Gamma} = \Gamma$, $\hat{\Sigma} = \Sigma + \{\iota \mapsto \odot\ c\langle \odot\ t_i \rangle_{i=1}^{q}\}$ and $\hat{Q} = Q$.

Case [CALL]. The fact that $\hat{\tau}$, as obtained from [$\mathbf{ELF_m}$], is a subtype of τ obtained from [CALL], is established from the result of [$\mathbf{ELF_m}$] and the constraint $\rho(\oplus\ t) <: \tau$ occurred in ψ in the premise of [CALL]. Finally, by assumption weakening rule, we set $\hat{\Gamma} = \Gamma + \{v_i :: \oplus \hat{t}_i\}_{i=1}^{q} + \{\mathtt{this} :: \oplus \hat{t}_0\}$, $\hat{\Sigma} = \Sigma$, $\hat{Q} = Q \cup \{V^*\}$.

Case [$\mathbf{ELF_d}$, $\mathbf{ELF_m}$]. There are two subcases for consideration:

Subcase [**D-Ret-d-1**, **D-Ret-m-1**]: By induction hypothesis.

Subcase [**D-Ret-d-2**, **D-Ret-m-2**]: By induction hypothesis and the Assumption Weakening Lemma. □