JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 4, No. 10, 2005

On Access Restriction with Java Wildcards

Mirko Viroli, DEIS, Alma Mater Studiorum - Università di Bologna, Italy
Giovanni Rimassa, Whitestein Technologies AG, Zurich, Switzerland

Java *wildcards* is a new programming mechanism shipped with the Java 5.0 release, introduced to provide a flexible subtyping mechanism for generic types. Safety is retained by providing rather peculiar and non-trivial mechanisms to restrict access to a class functionalities (methods and fields), which are currently not deeply described in the Java Language Specification. In this paper we develop on the theory of *variant parametric types* from which wildcards originated, and study a framework to describe these access restriction issues in detail, promoting the understanding and fruitful exploitation of this new programming concept.
Our work is both technical and conceptual. On the one hand, we provide an abstract characterisation of formal rules to access restriction, then instantiated to the particular implementation of wildcards in current Java. On the other hand, we show that such a characterisation induces a natural description and understanding of access restriction in terms of the ability of (instances of) a generic class to produce/consume elements of the abstracted type.

1 INTRODUCTION

The characterisation of the different kinds of type polymorphism that can be expressed within object-oriented programming languages is well known since almost twenty years, and features inclusion polymorphism and parametric polymorphism as the two possible universal forms of polymorphism [2]. Despite this peer relationship in the classification, inclusion and parametric polymorphism have never been equal in the history of object orientation.

Inclusion polymorphism — typically achieved through the combination of inheritance and subtyping — is considered one of the fundamental pillars of the object-oriented paradigm, whereas parametric polymorphism ended up being absent from many object-oriented languages, and was introduced in major ones only in later versions. This predominance of inclusion polymorphism actually stems from valid language level arguments [9], and is supported by a clear and well-known *domain interpretation* [3] through the 'is-a' specialisation relation — describing which concepts of the system to be modelled can be represented by the language construct.

Even Java, though introduced relatively recently, did completely without parametric polymorphism until its version J2SE 5.0 (October 2004) [13] — also internally known as JDK 1.5. [1] In fact, differently from e.g. C++, Java initially strived to

[1] Similar argument applies to C#, which, even though younger than Java, included generics

Cite this article as follows: Mirko Viroli, Giovanni Rimassa: *On Access Restriction with Java Wildcards*, in Journal of Object Technology, vol. 4, no. 10, 2005, pages 117–139, http://www.jot.fm/issues/issues_2005_12/article6

be a strictly single paradigm language: it was object-oriented from the start and has purposefully avoided to break new grounds, choosing instead to keep only the simplest, most time-tested features. However, with this new version of Java the language makes a set of new constructs available to the programmer. Mainly, parametric polymorphism has been added to Java through support for *generic classes* in the style of GJ language [12]; differently from *C++ templates*, Java generics rely on F-bounded polymorphism [1], which is more heavily based on static typing and better fits Java compilation schema. Such a proposal is actually known and studied since 1998 (see also [5]) and its main applicability can be considered as being relatively well understood.

However, the release of Java generics comes equipped with a fairly new mechanism called *wildcards* [14], which supports so-called *use-site variance*: it is the result of applying to the Java programming language the construct known as *variant parametric types* (VPTs), evaluated for inclusion in Java as it appeared in [6][2]. Wildcards parameterised types (WPTs) are types of the kind `List<? extends T>`, `List<? super T>`, `List<?>` — where `T` is any reference type. They are used to factor over a number of different instantiations of the same generic class: e.g. any `List<T>` where `T` is subtype of `Number` can be passed to where a `List<? extends Number>` is expected. This construct provides a means by which subtyping (inclusive polymorphism) can better integrate with generics (parametric polymorphism), and finds many applications, e.g. in the Java Collections Framework.

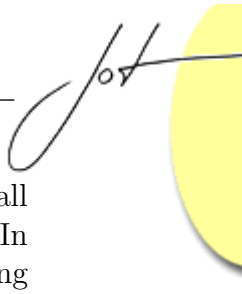
In exchange of the flexible subtyping, a WPT limits the way in which fields and methods of a generic class can be accessed — for instance a `Number` element cannot be put into a list with type `List<? extends Number>` through method `add()`, only the `null` element can. The details of such access restrictions are actually very peculiar and subtle: they are actually grounded on the formal theory of VPTs as described in [6], though some modifications are to be applied to fit the wildcards context. However, the details of this issue are mostly neglected in the Java Language Specification (JLS) [7], making the usage of wildcards in the design of new libraries quite obscure in several situations.

The goal of this paper is to provide a conceptual description and interpretation of such restriction rules, which could help programmers to exploit the wildcards construct so as to meaningfully represent concepts and entities in their program. This is achieved by developing a framework where semantic aspects of wildcards can be understood in terms of access restrictions to a class operations (methods and fields), leading to a natural interpretation in terms of a generic class ability to produce/consume elements of the abstracted types.

The remainder of this article is as follows. In Section 2, we deepen the motivation of this work, showing how the current J2SE and JLS versions provide little support to the understanding of how WPTs can be usefully leveraged to build new libraries. In Section 3, we address the problem in an abstract way, describing the rationale

only in its recent 2.0 version [10].

[2]More on that can be found in the third edition of the Java Language Specification [7].



of access restriction rules and a general implementation for them in what we call the "use-site variance" framework — a generalisation to both VPTs and WPTs. In Section 4 we describe how such rules are specialised to the case of WPTs, providing thorough insights in the semantics of this construct and comparison with respect to VPTs. In Section 5, the production/consumption interpretation of access restriction is provided and put to test in some existing applications of the Java Collections Framework. Section 6 concludes providing final remarks.

2 ISSUES WITH JAVA WILDCARDS

Consider the following generic class `Vector<X>` as allowed in Java 5.0, representing a vector where the element type has been abstracted in a type variable `X` with *(upper)bound* `Object`:

```
class Vector<X extends Object>{
  private X[] xs;
  Vector(X[] xs){ this.xs=xs; }
  void setElement(X x, int pos){ xs[pos]=x; }
  X getElement(int pos){ return xs[pos]; }
  int size(){ return xs.length; }
}
```

From this definition, standard generic types of the kind `Vector<T>` are available to programmers, obtained by instantiating the type parameter `X` with an actual type `T`: such types include e.g. `Vector<Integer>`, `Vector<String>` and `Vector<Vector<String>>`. Other types can then be used which are called *wildcard parameterized types* (WPTs) [7]. They are of the kind `Vector<?>`, `Vector<? extends T>` and `Vector<? super T>`: e.g. types `Vector<? extends Integer>`, `Vector<? super Integer>` and `Vector<? extends Vector<? super Integer>>`. These types are not used to create objects in `new` expressions, but rather as sort of interfaces over standard generic types: when accessing a method or a field of an object whose type is a WPT, a *capture conversion* operation is first applied [7], turning the wildcard "`?`", "`? extends T`" or "`? super T`" into a fresh type variable with certain bounds — and hence a WPT into a standard generic type. Let symbol `<:` express the subtype relation, and `NullType` the type for the `null` expression, we have:

- `Vector<?>` is turned into `Vector<X>` with `NullType<:X<:Object` (any `X`);
- `Vector<? extends T>` is turned into `Vector<X>` with `NullType<:X<:T` (any `X` smaller than `T`);
- `Vector<? super T>` is turned into `Vector<X>` with `T<:X<:Object` (any `X` greater than `T`).

Roughly speaking, a WPT can be understood as a generalisation of standard generic types, where the type parameter is not an actual reference type, but rather a set of such types — an "interval" expressed by the bounds. Subtyping between generic types is e.g. simply expressed in terms of inclusion of such intervals: the WPT `T` is a subtype of `R`, if the interval induced by `T` is included in `R`'s. Notable examples are the following:

Covariance — if `R<:T` then `Vector<R><:Vector<? extends T>`, because of the inclusion: `[R,R]`$\subseteq$`[NullType,T]`

Contravariance — if `T<:R` then `Vector<R><:Vector<? super T>`, because of the inclusion: `[R,R]`$\subseteq$`[T,Object]`

Bivariance — `Vector<R><:Vector<?>` for any `R`, because of the inclusion: `[R,R]`$\subseteq$`[NullType,Object]`

This interval intuition is pretty much what the JLS describes about the semantics of WPTs. This description is actually quite compact and simple, and is sufficient per se to describe how programmers can invoke libraries using wildcards. As an example, consider interface `List<E>` in the Java Collections Framework (implementing the root interface `Collection<E>`):

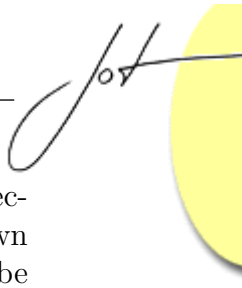
```
class List<E> implements Collection<E>, Iterable<E>{
    ...
    void add(E o);
    E get(int index);
    Iterator<E> iterator();
    boolean addAll(Collection<? extends E> c);
}
```

We can have:

```
List<Number> ln=...;
List<Integer> li=...;
ln.addAll(li); // OK because of WPTs subtyping
```

Method `addAll()` takes a collection `c` and adds all its elements into the receiver list. The wildcard used in the argument type states that any `Collection<T>` can be passed provided that `T` is a smaller type than the receiver's element type `E`: e.g. `Collection<Integer>` can be passed since a `Collection<? extends Number>` is expected — `Integer` is contained in the interval `[NullType,Number]` induced by the wildcard "`? extends Number`".

In general, wildcards are shown to be useful to enlarge the applicability of methods of generic classes, which can now accept a wider range of arguments. However,



the impact of wildcards goes beyond the mere coding of clients for the Java Collections Framework: it is reasonable to expect Java programmers to write their own libraries leveraging the advantages of wildcards — at least for this construct to be considered fully successful.

Subtleties arise when considering that the flexible subtyping of WPTs necessarily comes in exchange of a limited access to their methods and fields. Going back to method `addAll()` in class `List<E>`, any implementor of it should actually exploit the formal argument `c` in a limited way, only in those cases that are safe for all the possible actual arguments passed. This is a direct consequence of Liskov's substitutability principle [8]: the more objects can be passed, the less operations can be applied. To have a first look at how the Java compiler allows/constrains method access through WPTs, consider the following code:

```
Number n=...;
List<? extends Number> len=...;
List<? super Number> lsn=...;

len.add(n);              // Not allowed!
len.add(null);           // Allowed!
len.addAll(len);         // Not allowed!
len.addAll(null);        // Allowed!
lsn.add(n);              // Allowed!
lsn.addAll(lsn);         // Not Allowed!
Number n2=lsn.get(0);    // Not Allowed!
Object o2=lsn.get(0);    // Allowed!
```

Understanding the reasons for these constraints is currently completely left to the programmer's intuition. In simple situations this is possible: in the first case above, one can recognise that `n` cannot be passed to `len.add()` since `len` could have as type parameter any type `E` smaller than `Number`, hence it is not guaranteed that the type of `n` — `Number` — is of a smaller type; other cases such as `lsn.addAll(lsn)` are increasingly complex to guess.

Clearly identifying the access restriction applied to WPTs is a first step towards a true understanding of their semantics, meaning, and applicability to model real-domain concepts. In the end, this is necessary for the designer of a class signature to find a good balance between flexibility in client invocations (widening a method applicability) and expressiveness in code implementation (restricting access to arguments). By widening a method's argument type from `Collection<E>` to `Collection<? extends E>` one actually limits the accessibility to its methods: what functionalities will the programmer be actually allowed to exploit? Are we sure they are expressive enough to implement the intended meaning of the method? These are the sort of issues our work here is meant to start addressing.

3 ACCESS RESTRICTION IN THE USE-SITE VARIANCE FRAMEWORK

In this section we present some design and typing issues related to the use-site variance framework, under which the access restriction features of WPTs can be understood and expressed. We basically rely on ideas of the type system described in [6], but our semi-formal presentation here is novel: only the peculiar aspects needed to deal with access restriction are considered, which are formulated in terms of meta-rules for restrictions to be later specialised. We show that access restriction can be understood in terms of widening method types — where handling of argument and return types are fully dual.

Main Definitions

Our syntax and terminology here resembles that of VPTs reported in [6] for simplicity, in particular we shorten the notation for (type parameter) annotations using symbols `*` for "`?`", `+` for "`? extends`" and `-` for "`? super`". We let meta-variable `C` range over classes, `X` over type variables, and define the syntax:

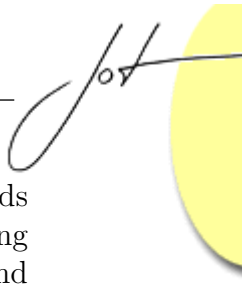
`N`	::=	`C<`$\mathtt{v}_1\mathtt{T}_1$`,..,`$\mathtt{v}_k\mathtt{T}_k$`>`	WPT
`T,R,S`	::=	`X` | `N`	Type
`M`	::=	`(`$\mathtt{T}_1$`,..,`$\mathtt{T}_k$`)->`$\mathtt{T}_0$	Method Type
`v,w,z`	::=	`o` | `+` | `-` | `*`	Type Parameter Annotation

Notation `R[T/X]` is used for the type obtained from `R` by substituting all the occurrences of variable `X` with `T`.

Any class `C` generates four kinds of types: *(i) invariant* types (standard generic types, also called instance types) of the kind `C<T>` (written `C<oT>` in the formal syntax for uniformity), *(ii) covariant* types of the kind `C<+T>`, *(iii) contravariant* types of the kind `C<-T>`, and *(iv) bivariant* types of the kind `C<*>` (written `C<*T>` for any `T` in the formal syntax for uniformity — these two forms being equivalent [6]). Given the relationship between annotations and corresponding subtyping, we introduce an order relation $\leq$ on annotations, defined as the reflexive and transitive closure of relation $\{\mathtt{o} \leq \mathtt{+}, \mathtt{o} \leq \mathtt{-}, \mathtt{+} \leq \mathtt{*}, \mathtt{-} \leq \mathtt{*}\}$: its main property is that $\mathtt{v} \leq \mathtt{w}$ implies `C<vT><:C<wT>`. Moreover, symbol $\vee$ is used for the upperbound of two annotations, and we denote by $\overline{\mathtt{v}}$ the operation of complementing annotation `v`, defined as $\{\overline{\mathtt{o}} = \mathtt{o}, \overline{\mathtt{+}} = \mathtt{-}, \overline{\mathtt{-}} = \mathtt{+}, \overline{\mathtt{*}} = \mathtt{*}\}$.

Technically, in this paper we are concerned with the problem of typing the invocation of a method `m()` or the access to a field `f` on a receiver that is given one of the above types `N`. More briefly, we shall refer to the problem of accessing a method `m()` or field `f` through type `N`.

For standard generic types such as `C<T>`, the problem is already solved by the usual typing rules of GJ [12]. In particular, if class `C<X>` defines method



`To m(T1 x1,..,Tk xk){...}` then accessing method `m()` through type `C<T>` yields a method type of the kind `(T1[T/X],..,Tk[T/X])->To[T/X]`, that is, returning the type obtained from `To` substituting occurrences of `X` with actual type `T`, and expecting as i^{th} argument type `Ti` after substituting `X` with `T`. For instance, invoking method `add()` on a receiver with type `List<Number>` yields method type `(E[Number/E])->void[Number/E]`, that is `(Number)->void` — a `Number` element is expected as argument, and a `void` element is returned, namely, no element.

This discussion naturally extends to the case of more type arguments as in class `D<X,Y>`: the invariant type provides an instantiation to both type arguments `X` and `Y`, which get substituted to the actual formal arguments when accessing a method. Also, reading and writing a field have the same typing treatment as invoking a getter or a setter method for that field. Moreover, we do not handle generic methods as they are mostly orthogonal to our aim — e.g. the *wildcard capture* mechanism described in [14, 7] being not related to access restriction properties. Thus, in this paper, for simplicity we only deal with classes with one type argument, and focus on accessing methods with only one argument — the other cases being a slight generalisation. Our aim is then to give a generalisation to the above access schema, finding rules for accessing a method `m()` through any WPT.

Schema of Access Restriction

This problem can be defined in abstract terms by an operator $[.] \dot{\rightarrow} [.]$: suppose class `C<X>` defines a method `m()` with type `(Ti)->To`, then the notation

$$[\texttt{(Ti)->To}] \xrightarrow{\texttt{C:X}\mapsto\texttt{vT}} [\texttt{(Ti')->To'}]$$

is used to state that accessing `m()` through type `C<vT>` yields method type `(Ti')->To'`. Symbol Δ ranges over elements of the kind `C:X`$\mapsto$`vT`, carrying all the information on a receiver instantiation — hence we generally write `[M]` $\xrightarrow{\Delta}$ `[M']`[3].

In particular, we are concerned with identifying sufficient conditions for a specific implementation of this operator to yield safe solutions, namely, solutions not breaking type soundness of the language.

Following the idea of Liskov's substitutability property, in order to retain safety WPTs should exchange flexibility in subtyping with a limited access to methods. For instance, since `C<+T>` is a supertype of all `C<R>` where `R<:T`, then all objects of a type `C<R>` can be passed to where a `C<+T>` is expected, thus `C<+T>` should provide only those functionalities common to all `C<R>` — otherwise, there would be the risk of accessing a functionality on an object that does not provide it, breaking soundness. In particular, `C<+T>` should provide a general functionality which is a "restricted" version with respect to `C<T>`'s. A simple example is as follows:

[3]Note that, differently from [6], we make the Δ environment carry the original class `C`, since we generally need to recover the bound to the type variable `X` as reported in its class definition.

```
List<+Number> l=new List<Integer>();
l.add(new Float(1.2));   // Should be prevented at compile-time
```

We see that the argument type to `add()` when accessed through `List<+Number>` cannot be `Number`. So, while the "`void add(Number n)`" functionality is provided by type `List<Number>`, only a restricted version of it can be provided by `List<+Number>`.

This notion of restriction is naturally captured by the concept of subtyping between method types: accessing a restricted version of a method amounts to retrieving a greater method type, which in fact expresses a more general (less specific) method type. Recall the contravariance subtyping rule for method types (function types): a method type `(Ti)->To` is a subtype of `(Ti′)->To′` if and only if `Ti` is a supertype of `Ti′` and `To` is a subtype of `To′` — this guarantees to safely pass a `(Ti)->To` where a `(Ti′)->To′` is expected.

Accordingly, if e.g. $\texttt{M}^{\texttt{T}}=(\texttt{Ti}^{\texttt{T}})\texttt{->To}^{\texttt{T}}$ is the type of method `m()` when accessed through type `C<T>`, and $\texttt{M}^{\texttt{+T}}=(\texttt{Ti}^{\texttt{+T}})\texttt{->To}^{\texttt{+T}}$ the one obtained through `C<+T>`, then $\texttt{M}^{\texttt{+T}}$ should necessarily be greater than $\texttt{M}^{\texttt{T}}$. This is because $\texttt{M}^{\texttt{+T}}$ should actually be greater than any $\texttt{M}^{\texttt{R}}$ such that `R<:T`. Considering the above case, the type of `add()` accessed through `List<+Number>` should be greater than the ones obtained from `List<Number>`, `List<Integer>`, `List<Float>`. Similar discussion applies when accessing a method through a type `C<-T>` or `C<*>`. In other words, the variability enabled by a WPT requires a generalisation when accessing a method type.

These relations actually provide the two main ingredients for obtaining conditions for safe access restriction. First of all, we observe that because of subtyping for method types, the rules for access restriction are actually separated in the way they handle argument and return type; thus we can write

$$[\texttt{(Ti)->To}] \xrightarrow{\Delta} [\texttt{(Ti′)->To′}] \quad \Leftrightarrow \quad (\texttt{Ti} \Downarrow^{\Delta}_{-} \texttt{Ti′} \quad \text{and} \quad \texttt{To} \Downarrow^{\Delta}_{+} \texttt{To′})$$

where, given a certain Δ, operator $\Downarrow^{\Delta}_{-}$ is used to obtain the argument type, whereas $\Downarrow^{\Delta}_{+}$ to obtain the return type. These operators are inspired by the close operator introduced in [6]; our work here adds the novel contribution of applying it in an uniform and dual way to both argument and return types — whereas in [6] it is applied to return types only.

Note that an element $\Delta = \texttt{C:X}\mapsto\texttt{vT}$ can be seen as introducing constraints in the variability of `X`: `X=T` if `v` is `o`, `X<:T` if `v` is `+`, `X:>T` if `v` is `-`, and no constraint on `X` if `v` is `*`.

Secondly, the conditions for these operators to be safe are obtained from the subtyping relation as follows. Given a Δ constraining variable `X` to range within given bounds, operators $\Downarrow^{\Delta}_{+}$ and $\Downarrow^{\Delta}_{-}$ determine a safe restriction relation $\xrightarrow{\Delta}$ if: *(i)* when $\texttt{To} \Downarrow^{\Delta}_{+} \texttt{To′}$ then `To′` is a supertype of `To` independently of the variability of `X` in Δ, and similarly, *(ii)* when $\texttt{Ti} \Downarrow^{\Delta}_{-} \texttt{Ti′}$ then `Ti′` is a subtype of `Ti` independently of the variability of `X` in Δ. When $\texttt{To} \Downarrow^{\Delta}_{+} \texttt{To′}$ holds we say that `To` "upwardly-closes" to `To′`

$$\frac{\mathtt{w} \leq \mathtt{v}}{\mathtt{X} \Downarrow_{\mathtt{v}}^{\mathtt{C}:\mathtt{X}\mapsto\mathtt{w}\mathtt{T}} \mathtt{T}} \quad [\text{VAR}]$$

$$\frac{\mathtt{R} \Downarrow_{\mathtt{v}}^{\Delta} \mathtt{S}}{\mathtt{C<wR>} \Downarrow_{+}^{\Delta} \mathtt{C<(v\vee w)S>}} \quad [\text{REC+}]$$

$$\frac{\mathtt{R} \Downarrow_{\mathtt{v}}^{\Delta} \mathtt{S} \qquad \mathtt{w} \geq \overline{\mathtt{v}}}{\mathtt{C<wR>} \Downarrow_{-}^{\Delta} \mathtt{C<wS>}} \quad [\text{REC-}]$$

Figure 1: Access Restriction Rules

through Δ: return type $\mathtt{To}'$ will be greater then any allowed substitution of $\mathtt{X}$ to $\mathtt{To}$. Dually, when $\mathtt{Ti} \Downarrow_{-}^{\Delta} \mathtt{Ti}'$ holds we say that $\mathtt{Ti}$ "downwardly-closes" to $\mathtt{Ti}'$ through Δ: the argument type $\mathtt{Ti}'$ should be smaller then any allowed substitution of $\mathtt{X}$ applied to $\mathtt{Ti}$. In fact, these close operators yield a method type which is compatible with the original one — it is more general — and that takes into account the possible variability of $\mathtt{X}$ due to the receiver's WPT[4].

General Rules for Use-Site Variance

A possible implementation for the close operators is as defined by the inference rules shown in Figure 1 — as usual, the top-side of such rules express a necessary condition for the bottom-side to hold. In this section we mostly focus on presenting examples of computation results of closing operators, leaving the discussion on their interpretation to the next section.

Operator $\Downarrow_{\mathtt{v}}^{\Delta}$ defines a binary relation between types, where the left-side is the type to be closed and the right-side the result of closing (an upperbound with $\Downarrow_{+}$ and a lowerbound with $\Downarrow_{-}$). In the left-side, such rules are intended to be applied to types including the type variable of Δ — if this is not the case a type is to be trivially considered closed to itself. Also note that being a relational operator, zero, one or many results can be obtained in general from a type to be closed. We write $R \not\Downarrow_{\mathtt{v}}^{\Delta}$ when for no type S we have $R \Downarrow_{\mathtt{v}}^{\Delta} S$.

Rule [VAR] corresponds to the basis of the recursive algorithm: a type variable can be closed if the annotation in $\mathtt{X}$'s instantiation $\mathtt{w}$ is smaller than the direction of the close operator $\mathtt{v}$. Basically, this rule is used to close a type variable to its bound as declared in Δ, provided it is compatible with the direction of closing. Examples

[4] Following the discussion in [6], this restriction approach can be seen in an abstract way as: *(i)* considering the receiver in the domain of existential types, *(ii)* accessing the method type on it, which yields an existential method type, *(iii)* obtaining a supertype of such method type in the domain of WPTs. Thanks to this connection with existential types, this general technique — which is a key design contribution of [6] — is shown to retain safety.

of applications for this rule are as follows:

$$\begin{array}{lll} \texttt{X} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{+} & \texttt{T} \\ \texttt{X} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto-\texttt{T}}_{-} & \texttt{T} \\ \texttt{X} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto\texttt{o}\texttt{T}}_{+} & \texttt{T} \end{array}$$

Rule [REC+] handles recursion through generic types of upward closing, finding upperbounds. First, the argument `R` is closed itself using the direction `v`, then the annotation to be used is obtained by "mixing" (through $\vee$ operator) `v` and the original annotation `w` in `R`. This rule exploits the fact that *(i)* if S is an upperbound to R then `C<+S>` is an upperbound to `C<+R>` (`v` $=$ `w` $=$ `+`), *(ii)* if S is a lowerbound to R then `C<-S>` is an upperbound to `C<-R>` (`v` $=$ `w` $=$ `-`), and *(iii)* in the other cases only the upperbound `C<*>` can be used. For instance, we have:

$$\begin{array}{lll} \texttt{C<X>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{+} & \texttt{C<+T>} \\ \texttt{C<-X>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{+} & \texttt{C<*T>} \\ \texttt{C<-X>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto-\texttt{T}}_{+} & \texttt{C<-T>} \\ \texttt{C<+C<X>>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{+} & \texttt{C<+C<+T>>} \\ \texttt{C<-C<X>>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{+} & \texttt{C<*C<+T>>} \end{array}$$

Finally, rule [REC-] dually handles recursion through generic types of downward closing, finding lowerbounds. Argument `R` is closed to `S` using direction `v`: if `w` is greater than the opposite of `v`, then, `S` is kept with annotation `w`. This rule exploits the fact that *(i)* if S is an upperbound to R then `C<-S>` is a lowerbound to `C<-R>` (`v` $=$ `+` and `w` $=$ `-`), *(ii)* if S is a lowerbound to R then `C<+S>` is a lowerbound to `C<+R>` (`v` $=$ `-` and `w` $=$ `+`); the cases *(iii)* where `v` $=$ `o` or `w` $=$ `*` trivially close. For instance, we have:

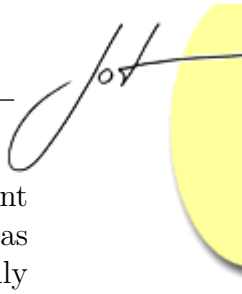
$$\begin{array}{lll} \texttt{C<-X>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{-} & \texttt{C<-T>} \\ \texttt{C<+X>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto-\texttt{T}}_{-} & \texttt{C<+T>} \\ \texttt{C<+C<-X>>} & \Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{-} & \texttt{C<+C<-T>>} \end{array}$$

These rules are obtained as a generalisation of those in [6], translated in our upward/downward dual schema.

4 ACCESS RESTRICTIONS OF JAVA WILDCARDS

We first note that the rules provided in previous section may cause close operators to yield zero results. This would mean that a method becomes completely unaccessible when invoked through a certain receiver. For instance we have:

$$\begin{array}{ll} \texttt{X} & \not\Downarrow^{\texttt{C}:\texttt{X}\mapsto-\texttt{T}}_{+} \\ \texttt{X} & \not\Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{-} \\ \texttt{C<X>} & \not\Downarrow^{\texttt{C}:\texttt{X}\mapsto-\texttt{T}}_{-} \\ \texttt{C<X>} & \not\Downarrow^{\texttt{C}:\texttt{X}\mapsto+\texttt{T}}_{-} \end{array}$$



Considering the first case, it would mean that a method `get()` returning an element of the type variable `X` cannot be invoked through a receiver `C<-T>`. In fact, whereas `T` is a lowerbound for `X`, an upperbound is in fact not known: `X` cannot be upwardly closed. Similarly, considering the second case, it would mean that a method `set()` accepting an element of type variable `X` cannot be invoked through a receiver `C<+T>`. The implementation of WPTs in Java 5.0 applies two basic variations to the above rules so as to make them less restrictive, and in the end, ensuring that it is never the case that a method cannot be invoked through a given receiver.

The first idea is that in the `get()` case, the type variable `X` has at least the upperbound defined in the `extend` clause where `X` is declared, or `Object` if none is specified — e.g. `Object`s can always be retrieved from any vector. Hence, the following new rule is to be added:

$$\texttt{X} \Downarrow_{+}^{\Delta} \mathit{bound}(\Delta) \quad \text{[BOUND]}$$

In conjunction with the other rules, we now have e.g. that:

$$\begin{array}{lll} \texttt{X} & \Downarrow_{+}^{\texttt{C:X}\mapsto\texttt{-T}} & \texttt{T} \\ \texttt{C<-X>} & \Downarrow_{-}^{\Delta} & \texttt{C<-}\mathit{bound}(\Delta)\texttt{>} \\ \texttt{C<+C<X>>} & \Downarrow_{-}^{\Delta} & \texttt{C<+C<-}\mathit{bound}(\Delta)\texttt{>>} \end{array}$$

In particular, the first case makes upward closing always return at least one result.

A dual consideration is that method `set()` could be invoked through type `C<+T>` by passing the `null` value — since in fact `NullType` can always be considered a lowerbound to each type variable. Seen in the context of access restriction, this means that the two following top level rules defining the close operator ${}^{w}\Downarrow_{\texttt{v}}^{\Delta}$ for WPTs are to be used:

$$\frac{\texttt{R} \Downarrow_{\texttt{v}}^{\Delta} \texttt{S}}{\texttt{R} \; {}^{w}\Downarrow_{\texttt{v}}^{\Delta} \texttt{S}} \quad \text{[OK]}$$

$$\frac{\texttt{R} \not\Downarrow_{\texttt{v}}^{\Delta}}{\texttt{R} \; {}^{w}\Downarrow_{\texttt{v}}^{\Delta} \texttt{NullType}} \quad \text{[NULL]}$$

They state that if the standard close operators ($\Downarrow_{\texttt{v}}^{\Delta}$) yields some result, than this is a result for WPTs as well [OK], otherwise the result of downward closing yields `NullType` [NULL]. The combination of these two adaptations is basically a pointwise departure from original rules, ensuring methods to be accessible through any receiver.

Another interesting issue with access restriction rules is that more results can be provided when closing a type, e.g.:

$$\begin{array}{lll} \texttt{C<C<-X>>} & \Downarrow_{+}^{\texttt{C:X}\mapsto\texttt{+T}} & \texttt{C<+C<*T>>} \\ \texttt{C<C<-X>>} & \Downarrow_{+}^{\texttt{C:X}\mapsto\texttt{+T}} & \texttt{C<-C<-T>>} \end{array}$$

To check this situation suffices it to note that the following code is correctly compiled by the Java 5.0 compiler (both results of closing can be seen as return type of the method):

```
class C<X> extends Object{
    C<C<? super X>> m(){
        return null;
    }
    void meth(C<? extends Number> t){
        C<? extends C<?>> t1=t.m();
        C<? super C<? super Number>> t2=t.m();
    }
}
```

The type of expression `t.m()` is precisely obtained by closing type `C<C<? super X>>` (that is, `C<C<-X>>`) under the environment `C:X↦+Number`: as the code shows this type is interpreted as being a subtype of both `C<? extends C<?>>` and `C<? super C<? super Number>>` (that is, `C<+C<*Number>>` and `C<-C<-Number>>`). But what is the type of `t.m()` anyway? As discussed in [6], this type is not a VPTs (it is neither a wildcard) and cannot be directly expressed by a programmer: Java 5.0 represents it as the type "`C<C<? super capture of ? extends Number>>`"[5], as revealed by the error message obtained trying to assign `t.m()` to a wrong variable:

```
Integer t1=t.m();
-->
incompatible types
found   : C<C<? super capture of ? extends Number>>
required: Integer
```

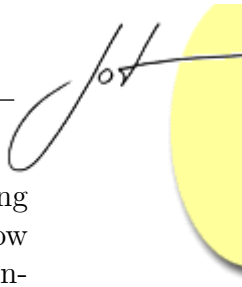
This type is implemented inside the compiler by a "capture" as described in Section 2, that is, as the type `C<C<? super Y>>` where `Y` is a fresh type variable with upperbound `Number` (and lowerbound `NullType`).

VPTs and Uniqueness

It is interesting here to show the differences between WPTs and VPTs. VPTs basically adhere to the general restriction rules of Section 3. The only departure is due to the need of simplifying the programmers' understanding (and also the implementation of static analysis in compilers): the original work of VPTs in [6] proposed to avoid closure operators to yield more results. To this end, the following more refined rule is used instead of [REC+]:

$$\frac{\mathtt{R} \Downarrow^{\Delta}_{+} \mathtt{S}}{\mathtt{C\langle wR\rangle} \Downarrow^{\Delta}_{+} \mathtt{C\langle(+\vee w)S\rangle}} \quad [\text{REC+U}]$$

[5] In type theory, this type can be understood as the existential type `∃X<:Number.C<(∃Y:>X.C<Y>)>` [11].



This rule guarantees the result of (both updward and downward) closing being unique: it promotes "covariance propagation" for upward closing, since e.g. now type `C<C<-X>>` only closes to `C<+C<*T>>` in environment `C:X↦+T` — it can be understood as simply "adding" the covariance symbol `+` at each level of nesting. Apart from this issue, which only affects types with a complex parameterisation structure, and from the idea of exploiting the trivial type variable bounds `NullType` and the declared upperbound, VPTs and WPTs provide the same access restriction rules.

5 UNDERSTANDING RESTRICTIONS

Whereas the closure rules discussed in previous sections can actually be used to compute method types when accessed through WPTs — and hence be used e.g. to guide the implementation of a compiler —, it is clear that a more conceptual description of their behaviour would be usuel, to be more fruitfully leveraged by designers and programmers. A first step towards the understanding of restrictions in use-site variance is made in [6], where VPTs are given an interpretation applicable to collection classes only, referring to read-only and write-only views to a collection induced by covariant and contravariant parametric types. For instance, given the generic class `List<X>` for lists of elements, `List<+Integer>` is understood as the type of those lists of integers which can only be read — since in VPTs, invoking method `set()` is not allowed through `List<+Integer>`. Similarly, `List<-Integer>` is the type of those lists that can only be written, and `List<*>` which can neither be read nor written.

This interpretation has however some problems. First, it does not work well with wildcards, as we showed that e.g. `null` elements can be written in a list `List<? extends Integer>` anyway. Then, it applies to collection (and container) classes only, as only for them the concepts of reading and writing are defined. Moreover, this interpretation also falls short when collection classes have methods that go beyond simple setters or getters of elements `X`. For instance, a method "`boolean contains(X x)`" in class `List<X>` cannot be invoked on a type `List<+Integer>` even though it is not about writing the content of the list. Since WPTs happen to be useful in a wider range of cases, including not only collection classes with the above kinds of methods, but also others such as e.g. streams, references, event producers and listeners classes, the need for a more broadly applicable interpretation clearly arises.

Accordingly, we introduce a wider interpretation for WPTs and their features, which generalises over the one in [6], and enables the understanding of a number of possible application cases — this interpretation can be used with VPTs as well, the few differences will be emphasised throughout. By exploiting the ideas explained in the previous section, this is mainly achieved by seeing methods of generic classes as functionalities to produce and consume elements (of the abstracted type), and describing WPTs as views over that classes that prevent their ability to either produce or consume such elements. Since here we move to more practical aspects, we

```
interface ReadCollection<X>{
    X get(int pos);
}
interface WriteCollection<X>{
    void put(X elm, int pos);
}
interface Collection<X>{
    X get(int pos);
    void put(X elm, int pos);
    boolean contains(X elm);
}
interface Stream<X>{
    X readNext();
    void writeNext(X elm);
}
```

```
interface Reference<X>{
    void initElement(X elm);
    X resolve();
}
interface Generator<X>{
    X produce();
}
interface Listener<X>{
    void listen(X event);
}
```

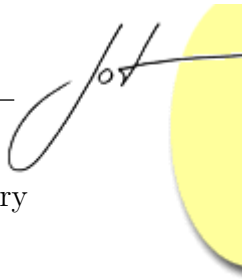
Figure 2: Examples of Interfaces for generic managers of different sorts

get back to WPT syntax, writing types `C<? extends T>`, `C<? super T>` and `C<?>` for `C<+T>`, `C<-T>` and `C<*>`.

Basic Ontology

As far as access restriction in WPTs is concerned, a generic class `C<X>` is interpreted as a (generic) class of *managers* handling *elements* — that is, objects of the abstracted type `X`. Methods of a generic class, with their arguments and result, are correspondingly seen as functionalities provided by one such manager: some of them may let these elements enter and escape the scope of the manager. This is achieved by consuming and producing these elements (or managers of these elements), respectively as actual arguments and return objects. A standard generic type such as `C<Integer>` is used to create and handle a particular case of managers: those of integer elements.

The concept of a "manager" should up to this point be considered in its broader acceptation, as a generalisation of the concepts of collection, stream, reference, generator, listener, and the like — see Figure 2. In general, a write-only collection, an output stream, and a listener can be seen as managers with functionalities that only consume elements, while a read-only collection, an input stream, and a generator as managers with functionalities that only produce elements. Certain classes, such as `Collection<X>`, `Stream<X>`, and `Reference<X>` have both a production and consumption character, and this is in fact the most frequent situation — as argued e.g. in [4] and observed in the Java Collections Framework. WPTs are precisely introduced to deal with these cases, to separately focus on the production or con-



sumption ability of these "hybrid" classes, and distinguish them when necessary and/or useful.

The WPTs `C<? extends Integer>`, `C<? super Integer>`, and `C<?>` — being all supertypes of `C<Integer>` that impose some access restriction to its functionalities — can be seen as interfaces over the class of managers `C<Integer>`. They are used to reduce the original ability of a manager to produce and consume elements through `C`'s functionalities, which is exchanged by a more flexible subtyping. As a result of the definition of the close operators, which look for minimal/maximal supertypes/subtypes independently of the type variable variability, we have that: `C<? extends Integer>` prevents any ability to consume elements of the abstracted type, and is hence called the *production version* of `C<Integer>`; `C<? super Integer>` prevents any ability to produce elements of the abstracted type, and is hence called the *consumption version* of `C<Integer>`; finally `C<?>` prevents any production and consumption ability, and is hence called the *closed version* of `C<Integer>`.

This production/consumption characterisation is provided to a class by propagating it to all its methods. For a given type of elements `T`, we say that the signature to `C<? extends T>` is obtained from `C<T>`'s by restricting all its functionalities to their production version, and similarly `C<? super T>` to the consumption version, and `C<?>` to the closed version. According to the general substitutability principle [8], each such restriction amounts to widening the production of the functionality (method return type) and narrowing the consumption of the functionality (method argument types) with respect to `C<T>`'s original version. This limits the contexts where the functionality can be applied, but consequently widens the set of managers on which it can be safely invoked.

Basic Interpretation of Restrictions

We first analyse the case where the type variable of a generic class appears as method argument and return type as it is — that is, not inside parameterisations. This is actually a very common case e.g. in simple collection classes such as class `Vector<X>` reported in Section 2. By applying the access restriction rules described in previous section, and for a given type `T`, we can identify for each of the types `Vector<T>`, `Vector<? extends T>`, `Vector<? super T>`, and `Vector<?>` a different signature composed by the restricted versions of `Vector`'s methods. These are reported in Figure 3. The case of method `getElement()` in the signature for `Vector<? extends T>` is read e.g. as follows: to obtain the return type we should (upwardly-)close the declared return type `X` under the environment Δ associating `X` to `+T`, which yields `T`. Applying closure to an argument type needs instead downward closure: here `int` naturally yields `int` again, for this is not a generic type containing the abstracted type `X`. The cases of method `setElement()` in `Vector<? extends T>` and `getElement()` in `Vector<? super T>` involve rules [NULL] and [BOUND] (`Object` is the declared bound of `X` in `Vector`). Finally note that as one may expect from substitutability,

```
signature to Vector<T>{ // Prod & Cons
  void setElement(T x, int pos);          // X ⇓_−^{C:X↦oT} T
  T getElement(int pos);                  // X ⇓_+^{C:X↦oT} T
  int size();
}
signature to Vector<? extends T>{ // Prod
  void setElement(NullType x, int pos);   // X ⇓_−^{C:X↦+T} NullType
  T getElement(int pos);                  // X ⇓_+^{C:X↦+T} T
  int size();
}
signature to Vector<? super T>{ // Cons
  void setElement(T x, int pos);          // X ⇓_−^{C:X↦-T} T
  Object getElement(int pos);             // X ⇓_+^{C:X↦-T} bound(C,X)
  int size();
}
signature to Vector<?>{ // Closed
  void setElement(NullType x, int pos);   // X ⇓_−^{C:X↦*T} NullType
  Object getElement(int pos);             // X ⇓_+^{C:X↦*T} bound(C,X)
  int size();
}
```

Figure 3: Viewpoints over a class `Vector<X>`

the signature to `List<T>` is that of a type smaller than both `List<? extends T>` and `List<? super T>`'s, which are themselves smaller than `List<?>`'s.

In these very simple cases, the production/consumption metaphor works as follows: *(i)* applying the production abstraction to method `setElement()` makes it no longer consuming elements of the abstracted type `T` — only the very specific `null` value is accepted — while leaving `getElement()` and `size()` unchanged; dually, *(ii)* applying the consumption abstraction to method `getElement()` makes it no longer producing elements of the abstracted type `T` — only very general `Object` elements are returned — while leaving `setElement()` and `size()` unchanged; finally, *(iii)* applying production and consumption works in both ways, leaving unchanged only the `size()` method. In practice, by simply looking at `X`'s position in a method's signature, one can deduce that `getElement()` is intrinsically a production functionality, method `setElement()` is a consumption one, and method `size()` is neither a production nor a consumption one. In the end, signature `List<? extends T>` forbids consumptions of elements of the abstracted type, `List<? super T>` forbids production of elements of the abstracted type, and `List<?>` forbids both.

More generally, Figure 4 shows the restrictions applied to return types (top) and argument types (bottom) in these cases, where columns range over possible receiver types, rows over the original type (return/argument type), and the table cell reports

ret	C<T>	C<? extends T>	C<? super T>	C<?>
X	T	T	*bound*	*bound*

arg	C<T>	C<? extends T>	C<? super T>	C<?>
X	T	NullType	T	NullType

Figure 4: Restriction rules for variables

the closed type. This figure shows that in the original signature, the restrictions simply allow `X` to appear as return type in the production receiver `C<? extends T>` and as argument in the consumption receiver `C<? super T>`. These restrictions are a consequence of rules [VAR,BOUND,NULL].

Note that a method "`boolean contains(X x);`" in class `Vector<X>` would be restricted similarly to method `setElement()` above — most notably only `null` can be passed when invoked through a production type `Vector<? extends T>`. This perfectly fits our interpretation, since both `contains()` and `setElement()` are consumption functionalities.

The same interpretation can be applied to VPTs as well, which actually provides a more restrictive access to methods. When the return type closes to the bound and the argument type to `NullType`, VPTs actually make the method be completely inaccessible: in Figure 3, the signature to `Vector<? extends T>` would lack `setElementAt()`, `Vector<? super T>` would lack `getElementAt()`, and `Vector<?>` both `setElementAt()` and `getElementAt()`.

First-Level Nesting of Type Variables

As a further step towards describing access restriction rules we consider the case where the formal argument `X` appears inside a parameterisation, that is in a type `C<X>`, `C<? extends X>`, `C<? super X>`, or implicitly in types `C<?>`.

Figure 5 shows the result of applying access restriction in this case. Following rule [REC+], return types are widened by "joining" the variance annotation of the receiver and that of the original return type as defined in the class. For instance, getting a consumer `D<? super X>` of `X` elements from a production manager `C<? extends T>` yields a manager `D<?>`, where neither production nor consumption can be applied. This situation can be understood observing that both the restriction of the manager providing the functionality `C<? extends T>` and the restriction originally considered for the return type `D<? super X>` have to be applied, which clearly results in a closed manager. This restriction is obtained by applying [REC+] where inner types are closed directly exploiting rule [VAR].

The case of argument types (bottom of Figure 5) needs instead a more refined view. It turns out that only four cases of restrictions over the argument actually lead to a functionality consuming elements which are not `null`: *(i)* accepting a closed abstraction is allowed independently of the receiver, *(ii)* a consumer `C<? super T>` is

allowed to accept a producer `D<? extends T>`, and *(iii)* a producer `C<? extends T>` is allowed to accept a consumer `D<? super T>`. The first case is justified considering that any manager of `T` should be allowed to consume a closed abstraction of `T` elements, for it never affects its original consumption/abstraction ability with respect to `T` elements. The second case and third case are justified in terms of hiding/encapsulation: a manager consuming `T` elements can do so by taking another manager in its scope which produces such `T` elements, and viceversa, a manager producing `T` elements can do so by taking another manager in its scope which consumes such `T` elements. Other cases are instead not allowed: for instance a producer `C<? extends T>` is not allowed to consume a producer `D<? extends T>` itself, as this would allow `C<? extends T>` to consume `T` elements through `D`. The restriction rule [REC-] is responsible for this behaviour.

Note that these parameterisation schemata actually do apply in the Java Collections Framework. Consider for instance the following class `List<X>`:

```
class List<X>{
    ...
    List<X> getTail();  // Tail getter
    void setTail(List<X> l); // Tail setter
    void addAll(List<? extends X> l); // From l to this
}
```

Signatures of its WPTs are then as follows:

```
signature to List<? extends T>{
   ...
   List<? extends T> getTail(); // Produces a producer version of tail.
   void setTail(NullType l);  // No consumption of Ts
   void addAll(NullType l);  //  No consumption of Ts
}
signature to List<? super T>{
   ...
   List<? super T> getTail(); // Produces a consumer version of tail
   void setTail(NullType l);  // No consumption of Ts
   void addAll(List<? extends T>); // Can consume a producer
 }
```

In particular, notice that access restrictions properly deal with the read/write interpretation over collections described in [6]. For instance, on a write-only `List` — a consumer `List<? super T>` — one can access the tail (`getTail()`) retrieving a write-only list, which can then be recursively updated. Moreover, it is safe to consume a `List<? extends T>`, as it can be exploited to retrieve `T` elements, to be possibly consumed by the receiver.

In most of these cases VPTs behave similarly: the only difference is that where argument types close to `NullType` or to the particular case `D<? super` *bound*`>`, VPTs

ret	C<T>	C<? extends T>	C<? super T>	C<?>
D<X>	D<T>	D<? extends T>	D<? super T>	D<?>
D<? extends X>	D<? extends T>	D<? extends T>	D<?>	D<?>
D<? super X>	D<? super T>	D<?>	D<? super T>	D<?>
D<?>	D<?>	D<?>	D<?>	D<?>

arg	C<T>	C<? extends T>	C<? super T>	C<?>
D<X>	D<T>	NullType	NullType	NullType
D<? extends X>	D<? extends T>	NullType	D<? extends T>	NullType
D<? super X>	D<? super T>	D<? super T>	D<? super *bound*>	NullType
D<?>	D<?>	D<?>	D<?>	D<?>

Figure 5: Restriction rules for nested types

make such methods completely inaccessible. Note that in all such cases, however, consumption of the abstracted elements is forbidden both in WPTs and VPTs.

Deeper Nestings

The case where type variable `X` occurs inside a more complex parameterisation such as `C<C<X>>` are generally handled by rules [REC+] and [REC-] (when inner types are treated with [REC+] and [REC-] recursively). As described in previous section, however, such behaviours are characterised by loss of uniqueness, and become then much more complicated to describe: in general, both return type and argument type can be closed to more than one type. We can expect the occurrence of these cases in programs to be less frequent, and in situations where any description would likely tend to be less intuitively applicable — in our case it would e.g. be about a manager producing/consuming a manager of managers and so on.

Still, we find here useful to provide some example application, at least to show the connection with the description provided so far.

```
class Reference<X>{
  ...
  X get();
  void set(X x);
}
class Vector<X>{
  ...
  void addAllRefs(Vector<? extends Reference<? extends X>> v);
  Vector<Reference<X>> getAllRefs();
}
```

Method `addAllRefs()` takes a vector of references `v`, and stores their content — `X` elements — into the receiver; dually, `getAllRefs()` produces a vector of refer-

ences. Note that cases with deeper nesting likely occur when handling complex data structures. When a method accepts a generic type, it is always useful to declare the greater argument type possible according to the intended semantics of the method: since `v` is going to be used as a producer of X elements — a producer of producers, in particular —, its right type is `Vector<? extends Reference<? extends X>>` instead of `Vector<Reference<X>>`. On the other hand, the return type to `getAllRefs()` can be `Vector<Reference<X>>`, for no hypothesis can be made on its production/consumption character. To understand how accessing these methods is restricted by WPTs, we apply close rules obtaining:

$$\texttt{Vector<? extends Vector<? extends X>>} \Downarrow_{-}^{\texttt{Vector:X}\mapsto\texttt{+T}} \texttt{NullType}$$

$$\texttt{Vector<Vector<X>>} \Downarrow_{+}^{\texttt{Vector:X}\mapsto\texttt{+T}} \texttt{Vector<? extends Vector<? extends T>>}$$

$$\texttt{Vector<? extends Vector<? extends X>>} \Downarrow_{-}^{\texttt{Vector:X}\mapsto\texttt{-T}} \texttt{Vector<? extends Vector<? extends T>>}$$

$$\texttt{Vector<Vector<X>>} \Downarrow_{+}^{\texttt{Vector:X}\mapsto\texttt{-T}} \texttt{Vector<? extends Vector<? super T>>}$$

That is, we have the interfaces:

```
signature to Vector<? extends T>{
   ...
   void addAllRefs(NullType v);
   Vector<? extends Reference<? extends T>> getAllRefs()
}
signature to Vector<? super T>{
   ...
   void addAllRefs(Vector<? extends Reference<? extends T>>)...
   Vector<? extends Reference<? super T>> getAllRefs()...
}
```

Through a production vector `Vector<? extends T>`, elements of the abstracted type cannot be consumed by `addAllRef()` — this method has a purely consumption character, hence only `null` can be passed. On the other hand, invoking `getAllRefs()` yields a purely production vector, that is a production vector including production references — in the end, this ensures that elements `X` cannot be consumed through the manager produced by `Vector<? extends T>`.

Through the consumption vector `Vector<? super T>`, the argument to `addAllRefs()` remains unchanged, analogously to the case of method `setProd()` in Figure 5. Instead, invoking `getAllRefs()` yields a production vector of consumption references — that is, from the returned manager we can just obtain consumption of `X` elements.

In spite of the intrinsic complexity in understanding these cases, we observe that

semantic-driven design choices could lead to patterns of parameterisation where the production/consumption interpretation still work reasonably.

As discussed in previous sections, VPTs here are more restrictive in that they always yield one result for closure — still preserving the intended interpretation.

6 CONCLUSIONS

The ontological description provided in this paper improves the traditional interpretation of variance based on read/write restrictions to collections. We generalise this idea relying on the more general concept of manager of elements, and on restrictions to its production/consumption functionalities. On the one hand, this allows us to deal with other classes than collections, such as streams, references, event producers and listeners, and so on. On the other hand, it also tackles those cases where methods are not necessarily used to "store" or "retrieve" elements through the class.

Being a general framework for use-site variance, the framework developed here is not applicable to WPTs only, but to the apprach of VPTs in [6] as well. The main difference is that VPTs are a bit more restrictive than WPTs: first, rules [BOUND] and [NULL] are not defined in VPTs, hence some methods can even become completely inaccessible; second, closure in VPTs always yield one solution, picking the one characterised by the so-called covariance propagation. In spite of these differences, the production/consumption interpretation described here works for VPTs as well, refining the more classical read/write one.

The main future work of this research is likely to be rooted on devising a full domain interpretation for Java generics, focussing on GJ-like F-bounded generics. This is meant to pave the way towards a better understanding of the features of Java generics, highlighting rooms for future extensions, developments, and applications.

ACKNOWLEDGMENT

We thank anonymous reviewers for useful comments, and Atsushi Igarashi for some clarification on the relationship between variant parametric types and existential types. This work was partially supported by the Italian PRIN 2004 Project "Extensible Object Systems" (Viroli).

REFERENCES

[1] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. F-bounded quantification for object-oriented programming. In *Proc. of ACM FPCA*, pages 273–280, September 1989.

[2] L. Cardelli and P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[3] J. O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, New York, 1999.

[4] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'95)*, pages 156–168, Austin, TX, October 1995.

[5] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23:396–450, 2001.

[6] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2347 of *LNCS*, pages 441–469. Springer-Verlag, 2002.

[7] B. Joy, J. Gosling, G. Steele, and G. Bracha. *The Java Language Specification (Third Edition)*. Addison-Wesley, New York, 2005.

[8] B. Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5), 1988.

[9] B. Meyer. Genericity versus inheritance. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'86)*, pages 391–405, 1986.

[10] Microsoft. C# language specification 2.0. Technical report, Microsoft, 2003. Web site: http://msdn.microsoft.com/vcsharp.

[11] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Trans. Prog. Lang. Syst.*, 10(3):470–502, 1988.

[12] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Making the future safe for the past: Adding Genericity to the Java programming language. In *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 183–200. ACM, New York, 1998.

[13] Sun Microsystem. J2SE 5.0. http://java.sun.com, 2004.

[14] M. Torgersen, C. Plesner Hansen, P. von der Ahé, E. Ernst, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. *Journal of Object Technology*, 11(3):1–20, 2004.

ABOUT THE AUTHORS

Mirko Viroli is a researcher associated with DEIS department, Alma Mater Studiorum - Università di Bologna, Italy. His interests cover advanced features of object-oriented programming languages, and engineering of concurrent and agent-based systems. He can be reached at mirko.viroli@unibo.it. See also http://www.ingce.unibo.it/~mviroli.

Giovanni Rimassa is employed by Whitestein Technologies AG, Switzerland, in a senior researcher position. He leads the development of infrastructures and APIs for agent-based systems. He can be reached at gri@whitestein.com.