Understanding Access Restriction of Variant Parametric Types and Java wildcards

Abstract

Variant parametric types [10] have been introduced to provide a flexible subtyping mechanism for generic types, and are recently being developed into wildcards [17], shipped worldwide with the upcoming version of JDK 1.5 official Java release. The two approaches, which are strictly related, retain safety by providing rather peculiar and non-trivial mechanisms to restrict access to a class functionalities (methods and fields). In this paper we aim at studying a unified framework to describe this issue in detail, and to facilitate the understanding and exploitation of this new programming concept.

Our work is both technical and conceptual. On the one hand, we provide formal rules to access restriction which can easily be specialised for the two approaches, so as to emphasise similarities and differences between them. On the other hand, we show that such rules promote a natural description and understanding of access restriction in terms of the ability of a generic class to produce/consume elements of the abstracted type.

1 Introduction

The characterisation of the different kinds of type polymorphism that it is possible to express 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 [3]. 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 such as C++ and Eiffel only in later versions. This predominance of inclusion polymorphism actually stems from valid language level arguments [12], and is supported by a clear and well-known domain interpretation [6] 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 being born relatively recently, did completely without parametric polymorphism until its upcoming version 1.5 [16]. In fact, differently from e.g. C++, Java initially strived to 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 the release of the upcoming Java Development Kit version 1.5, the language makes a set of new tools available to the programmer. Mainly, parametric polymorphism has been added to Java through support for generic classes in the style of GJ language [15]; differently from C++ templates, Java generics rely on F-bounded polymorphism [2], 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 [9]) and its main applicability can be considered as being relatively well understood. However, Java generics come equipped with a mechanism called wildcards [17] which is fairly new: it is the result of applying to the Java programming language the construct known as variant parametric types, evaluated for inclusion in Java as it appeared in 2002 [10]. Variant parametric types are types that factor over a number of different instantiations of the same generic class, providing a means by which subtyping (inclusive polymorphism) can better integrate with generics (parametric polymorphism).

In exchange of the flexible subtyping, a variant parametric type limits the way in which its fields and methods can be accessed. The details of such access restrictions, which are actually the most peculiar and subtle aspect of this mechanism, have been designed in [10] and presented through a sound type system — according to the standard programming languages approach [18].

At this stage, a main open issue in this context is to provide a useful domain interpretation of such restriction rules, which could help programmers to exploit the wildcards construct to meaningfully represents concepts and entities to be represented in the program. On the one hand, one such interpretation is depicted in [10] in terms of the read-only/write-only exploitation of collection classes — following the existing work on the similar mechanism of variant classes [12, 5, 1, 4]. This interpretation is however too limited, and it is also shown to fall shortly when full-featured collection classes are concerned (see e.g. [17]), thus calling for a new description approach. On the other hand, wildcards and variant parametric types indeed are slightly different mechanisms, whose relationships is yet to be studied in detail possibly revealing the need for different interpretations. In particular, the semantics of variant parametric types is described in terms of existential types [8, 14], through the combination of an open operator. Instead, at the time of writing wildcards do not provide any clear semantic specification: they are mostly defined as partially abstract generic types with safe access — we used the beta compiler of JDK 1.5 to test and understand typing details.

Accordingly, the goal of this paper is to start addressing these issues, providing a framework where (i) semantic aspects of wildcards and variant parametric types can be uniformly understood in terms of access restriction, (ii) their differences can be remarked, and (iii) a natural interpretation to access restriction can be derived.

The remainder of this paper is as follows. In Section 2 we start presenting variant parametric types. Our description is semi-formal: on the one hand, it relies on formal rules to describe subtle typing details avoiding ambiguities; on the other hand, it still concentrates on a few relevant issues without providing the complete standard framework, based on formal syntax, operational semantics, and typing of a core calculus. Moreover, instead of relying on the connection with existential types — which can actually be harder to grasp by non-informed people — we discuss access restrictions based on the idea of narrowing method types, which we believe actually provides an easier understanding, featuring e.g. full duality between argument types and return types restrictions. Indeed, this presentation choice allows for a quite compact representation of access restriction rules (with respect to [10]), which enables to develop a reasonably simple interpretation for them.

Accordingly, based on this description, Section 3 elaborates on an ontological framework which we associate to variant parametric types, and which can be used as a substantial step for developing a full domain interpretation for this construct. This is rooted on the idea of seeing generic classes as “generic managers” of elements, and their methods as functionalities to consume and produce such elements. Accordingly, variant parametric types are seen as
interfaces applied to such managers to limit their ability to produce/consume elements. Simple examples taken from the Collection library of JDK 1.5 are used to evaluate this interpretation.

Section 4 concludes by discussing perspective of future works.

The paper is based on variant parametric types for simplicity of presentation: still, its application to wildcards is discussed — all the differences will be clearly pointed out throughout the paper.

2 Variant Parametric Types

In this section we present the main design and typing issues of variant parametric types. Even though we rely on the same type system described in [10], our semi-formal presentation here is novel, as it focuses on just the peculiar aspects of variant parametric types dealing with access restriction while abstracting away from other less relevant issues. In particular, we show that access restriction can be understood in terms of narrowing method types, where handling of argument and return types are actually dual.

We let meta-variable C range over classes, X over type variables, and define the syntax:

\[
\begin{align*}
N & ::= C<T_1>...<T_k> & \text{Variant parametric type} \\
T,R,S & ::= X | N & \text{Type} \\
M & ::= (T,...,T)->T & \text{Method type} \\
v,w,z & ::= \circ | \star & \text{Variance annotations}
\end{align*}
\]

Any class C generates four kinds of variant parametric types: \(i\) invariant types (also called instance types) of the kind \(C<T>_\circ\) (abbreviated \(C<T>_\circ\)), \(ii\) covariant types of the kind \(C<T>_+\), \(iii\) contravariant types of the kind \(C<T>_\circ\), and \(iv\) bivariant types of the kind \(C<T>_\circ\).

**Invariant Types**

Invariant types such as \(C<T>_\circ\) are standard generic types, to which the usual typing rules of GJ [15] apply. In particular, if the following method definition occurs in class \(C<X>_\circ\)

```java
class C<X> {
    ...
    T m(T1 x1,...,Tk xk){...}
}
```

then accessing method \(m\) through type \(C<T>_\circ\) yields a method type of the kind \((T1<T/X>,...,Tk<T/X>)\rightarrow T_<T/X>\), that is, returning the type obtained from \(T\) substituting occurrences of \(X\) with actual type \(T\), and as \(i^{th}\) argument type \(T_i\) after substituting \(X\) with \(T\). For instance, given the definition

```java
class List<X> {
    X head;
    List<X> tail;
    List(X head, List<X> tail){ this.head=head; this.tail=tail; }
    X getHead(){ return head; }
    List<X> getTail(){ return tail; }
    void setHead(X head){ this.head=head; }
    void setTail(List<X> tail){ this.tail=tail; }
}
```
accessing method `getHeadFrom()` through type `List<String> -> String`. This means that when invoking `getHeadFrom()` on an object with type `List<String>`, an object with type `List<String>` has to be passed as argument, and an object with type `String` is expected as return.

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. 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. Moreover, we do not handle generic methods, for they play no significant role as far as variant parametric types are concerned [10]. This issue could be reconsidered in future works due to the appearance of the wildcard capture mechanism [17] in the final release of JDK 1.5.

**Variant Types and Subtyping**

For each invariant type `C<T>`, three variant types are automatically defined: `C<+T>`, `C<-T>`, and `C<*T>`. The subtyping relationship between these types is defined by the following rules:

- when `T = R`, `C<T> = C<R>` invariance
- when `T <: R`, `C<+T> <: C<+R>` covariance
- when `R <: T`, `C<-T> <: C<-R>` contravariance
- for each `T,R`, `C<T> <: C<R>` bivariance

which are added to the usual subtyping rules of GJ — handling inheritance subtyping, bound to type variables, transitivity and reflexivity. By transitivity, we obtain the following cases, which underlie the most typical applications of variant types: (i) a type `C<+T>` is used as a common supertype to all types `C<R>` with `R` subtype of `T`, (ii) a type `C<-T>` is used as a common supertype to all types `C<R>` with `T` subtype of `R`, and (iii) `C<*T>` as a common supertype to all types `C<R>`, `C<+R>`, and `C<-R>`, for any `R` — thus `C<*T>` is often abbreviated with `C<>`. Variant parametric types `C<+T>`, `C<-T>`, and `C<*T>` are used to generalise over different instantiations of class `C<X>`, and are not used to create objects.

Given the relationship between variant annotations, we introduce a `≤` order relation on them, defined as the reflexive and transitive closure of relation `{ o ≤ +, o ≤ -, + ≤ *, - ≤ * }`. In particular, we will use symbol `∨` for the upperbound of two annotations, `∧` for the lowerbound. Also, we denote by `v` the operation of inverting annotation `v`, that is, exchanging `+` with `-` and vice-versa, and leaving `◦` and `*` unchanged.
Variant Types and Schema of Access Restriction

Because of Liskov substitutability property [11], in order to retain safety variant parametric types exchange this flexible subtyping with a limited access to methods (and fields) — e.g., since more objects can be used where a $C<+T>$ is expected (compared to $C<T>$), $C<+T>$ should provide less functionality than $C<T>$. A simple example is as follows:

```java
List<+Number> l=new List<Integer>(1,null);
l.setHead(new Float(1.2)); // Should be prevented at compile-time
```

There, we see that the argument type to `setHead()` when accessed through $List<+Number>$ cannot be `Number` — i.e., the “`void setHead(Number head)`” functionality can no longer be provided.

This access restriction basically amounts to retrieving a smaller method type than what the invariant type would have yielded, which in particular cases can degenerate up to making a method completely unaccessible. Suppose a method $m$ can be accessed from $C<T>$, yielding method type $M^T$ of the kind $(T_i^T)\rightarrow T_o^T$ as described above. Then, the same method accessed from $C<+T>$ yields a different method type $M^T$ of the kind $(T_i^T)\rightarrow T_o^T$. $M^T$ is a restricted version of $M^T$ precisely in the sense that it is a subtype of $M^T$ according the contravariance subtyping for functions: namely, $T_i^T$ is a supertype of $T_i^T$ and $T_o^T$ is a subtype of $T_o^T$. Sometimes method $m$ becomes even inaccessible from $C<+T>$, which we can interpret as $T_i^T$ becoming too great, or $T_o^T$ too small. Similar discussion applies to the method type $M^T=(T_i^T)\rightarrow T_o^T$ obtained accessing $m$ from $C<-T>$ and $M^T=(T_i^T)\rightarrow T_o^T$ obtained accessing $m$ from $C<*T>$. $M^T$ can be different from $M^T$ and $M^T$, but still, due to the subtyping relation existing between them, we have that $M^T$ is a subtype of $M^T$ and of $M^T$.

Hence, access restriction rules can be defined in terms of a relation $\rightarrow$ of the kind

$$[(T_i)\rightarrow T_o] \xrightarrow{X:vT} [(T_i')\rightarrow T_o']$$

stating that, given a method with type $(T_i)\rightarrow T_o$ in a class $C<X>$, accessing it through type $C<vT>$ yields method type $(T_i')\rightarrow T_o'$.

Details of Access Restriction

We introduce the following terminology: by “$X$ is instantiated to $vT$”, written $X:vT$, we mean $X=T$ if $v$ is $\circ$, $X:<T$ if $v$ is $+$, $X:>T$ if $v$ is $-$, and no constraint on $X$ if $v$ is $*$. Symbol $\Delta$ is let range over elements $X:vT$.

Then, the rules for access restriction are actually separated in the way they handle argument and return type, in particular we write:

$$
\begin{align*}
\frac{\begin{array}{c}
T_i \xleftarrow{X:vT} T_i' \\
T_o \xrightarrow{X:vT} T_o'
\end{array}}{[(T_i)\rightarrow T_o] \xrightarrow{X:vT} [(T_i')\rightarrow T_o']}
\end{align*}
$$

Notation $T_i \xleftarrow{X:vT} T_i'$ means that under the hypothesis that $X$ is instantiated to $vT$, $T_i'$ is a maximal subtype of $T_i$ without mentioning $X$ — that is, where $X$ does not occur. We say that $T_i$ downwardly-closes to $T_i'$. Similarly, notation $T_o \xrightarrow{X:vT} T_o'$ means that under the hypothesis

\[\text{The mechanism of wildcards is such that } T_i^T \text{ is at least the type of } \text{null and } T_o^T \text{ is at most } \text{Object. This means with wildcards a method is never made unaccessible.}\]
that \( X \) is instantiated to \( vT \), \( T_0' \) is a minimal supertype of \( T_0 \) without mentioning \( X \). We say that \( T_0 \) upwardly-closes to \( T_0' \).

This rule is at the core of the access restriction technique of variant parametric types. In fact, the close operations yield valid variant parametric types which are compatible with the original argument/return type and that take into account the variability of \( X \) imposed by the receiver’s instantiation. Following the discussion in [10], 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 type, (iii) obtaining the maximal subtype of such method type in the domain of variant parametric types. Thanks to this connection with existential types, this technique — which is a key design contribution of [10] — is shown to retain safety.

The recursive algorithms computing the two close operations are reported in Figure 1. To make rules more compact, we syntactically allow the notations \( T \downarrow \Delta R \) and \( T \downarrow \Delta^* R \).

Rule \([\text{VAR}]\) is used for the fixpoint of the algorithm: a type variable can be closed if the two annotations in the close operator and in \( X \)'s instantiation coincide. For instance, for return types we have \( X \downarrow X : +T \downarrow^+T \), and for argument types we have \( X \downarrow X : -T \downarrow^-T \).

Rule \([\text{REC+}]\) handles recursion of upward closing through nested types: the resulting annotation is obtained by “mixing” (through \( \vee \) operator) the original annotation \( w \) and the direction of closing for the inner type \( v \). For instance, for one-level nesting we have \( C<X> \downarrow^*T C<-T> \) and \( C<-X> \downarrow^*T C<+T> \), and in more complex ones, \( C+C<X> \downarrow^*T C+C+C<T> \) and \( C-C<X> \downarrow^*T C*C+C<T> \).

Finally, rule \([\text{REC-}]\) dually handles downward closing through nested types, which is a bit more complicated. First, closing is applicable if \( w \) subsumes (is greater than) the opposite of \( v \). As we shall show in next section, other then trivial cases where \( w = * \) or \( v = o \), this implies that closing is applicable when \( w \) is \( + \) and \( v \) is \(-\), or viceversa. Then, the annotation in the closed type \( w \) is always left unchanged with respect to the original type, which can be seen as equivalent to annotation \( w \vee \top \) — dually to rule \([\text{REC+}]\). For instance, we have \( C<-X> \downarrow^+T C<-T> \) and \( C+C<X> \downarrow^-T C+C<T> \), and \( C+C+C<X> \downarrow^+T C+C+C<T> \).

When either of the two close operations does not yield a result, the method is simply not accessible from that variant parametric type. For instance, we have:

\[
X \downarrow^*T \perp \quad X \downarrow^-T \perp, \quad C<X> \downarrow^+T \perp \quad C<X> \downarrow^-T \perp
\]

Moreover, the reader should notice that we characterised the output of close operations
as minimal supertypes/maximal supertypes instead of least supertypes/greatest subtypes. In fact, close operations are relations rather than functions: they may yield more than one output. For instance, we have:

\[ C\triangleleft C\triangleleft X \triangleleft \square \\xrightarrow{\triangleleft X+T} C\triangleleft C\triangleleft X \triangleleft \square \]

\[ C\triangleleft C\triangleleft X \triangleleft \square \\xrightarrow{\triangleleft X+T} C\triangleleft C\triangleleft X \triangleleft \square \]

This generally means that the restriction applied to a method type when accessed through variant parametric types actually yields more possible method types.

The type system proposed in [10] adopts the following approach to tackle this issue. Concerning return types and upward-close-operation, rule [REC+] is changed to the following version where \( v \) is constrained to +:

\[
\frac{R \xrightarrow{\triangleleft} S}{C\triangleleft wR \xrightarrow{\triangleleft} C\triangleleft ((+\bigvee w)S)} \quad [REC+U]
\]

This rule is known as the covariance propagation rule in [10]: it is used in variant parametric types, whereas wildcards do exploit rule [REC+]. Regarding argument types and downward-close operation, among the allowed method types the type system simply checks whether any of them is compatible with the current arguments passed to the invocation. If none is found the invocation is forbidden. With respect to these issues, the wildcards mechanism followed the design choices adopted by variant parametric types.

### 3 A Domain Interpretation to Understand Access Restriction

In [10] variant parametric types are given a domain interpretation applicable to collection classes only, referring to read-only and write-only views to a collection induced by covariant and contravariant 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, List\(<-Integer>\) which can only be written, and List\(<*Integer>\) which can neither be read nor written. Other than being applicable only to collection classes, this interpretation also falls short when collection classes have methods that do not work as simple setters or getters of elements \( X \). For instance, a method boolean equalsToHead(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. Still, variant parametric types happen to be useful in a wider range of cases, including both collection classes with the above kinds of methods, or even other kind of classes, such as stream classes, reference classes, event producers and listeners.

Accordingly, in this paper we introduce a more general interpretation for variant parametric types and their features, which abstracts over a number of possible application cases. This is meant to identify the main pillars of a domain interpretation for variant parametric types. In particular, we focus on providing a general means to understand the rules behind access restriction enforced by variant parametric types. By enjoying the presentation idea explained in previous section, this is mainly achieved by seeing generic classes as managers providing functionalities to produce and consume elements (of the abstracted type), and describing variant parametric types as views over that managers that enforce their ability to either produce elements or consume elements.
interface Collection<X> {
    X get(int position);
    void put(X element, int position);
}

interface Stream<X> {
    X readNext();
    void writeNext(X element);
}

interface Reference<X> {
    void initElement(X element);
    X resolve();
}

interface Generator<X> {
    X produce();
}

interface Listener<X> {
    void listen(X event);
}

Figure 2: Interfaces for generic managers of different sorts

3.1 Basic Ontology

As far as variant parametric types are concerned, a generic class C<X> is interpreted as a (generic) class of managers of elements of type X. Methods of the class, with their arguments and result, are correspondingly seen as functionalities provided by a manager (of that class), some of which may let such elements enter and escape the scope of the manager, that is, they may consume and produce such elements. An instance 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. Correspondingly, 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 [7]. Variant parametric types are precisely introduced to deal with these cases, to focus on the production and consumption ability of these general managers, and distinguish them when necessary and/or useful.

Variant types C<+Integer>, C<-Integer>, and C<*Integer> — being all supertypes of C<Integer> imposing some access restriction to C’s objects — 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. In particular, C<+Integer> is the production abstraction over C<Integer>, preventing any consumption ability, C<-Integer> is the consumption abstraction over C<Integer>, preventing any production ability, while C<*Integer> is the closed abstraction over C<Integer>, preventing any production and consumption ability.

These abstractions are provided over a class by propagating them to all its functionalities. For a given type of elements T, we say that the interface (or signature) to C<+T> is obtained from C<T>’s by restricting all its functionalities to their production version, and similarly for C<-T> and C<*T>. According to the general substitutability principle [11], 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 \texttt{C<T>’s original version}, thus limiting the contexts where the functionality can be applied—and consequently widening the set of managers on which it can be invoked. In particular, depending on the typing rules adopted, it may even happen that widening and narrowing degenerate until making the functionality actually inaccessible. Understanding in an intuitive way the details on such restrictions, as described in previous section, is the more crucial aspect of our ontological study, and is described in an incremental way in the following.

### 3.2 Basic Interpretation of Restrictions

We first analyse the case where the type variable of a generic class appears as method’s argument and return types as it is—that is, not inside parameterisations. This is actually a very common case e.g. in simple collection classes such as the following:

```java
class Vector<X>{
    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; }
}
```

By applying the access restriction rules described in previous section, and for a given type \(T\), we can identify for each of the types \texttt{Vector<T>}, \texttt{Vector<T>}, \texttt{Vector<T>}, and \texttt{Vector<T>} a different signature composed by the restricted versions of \texttt{Vector’s methods}. These are informally reported in Figure 3. In these very simple cases, the production/consumption metaphor gives a very intuitive description of access restrictions: (i) applying the production abstraction to method \texttt{setElement()} makes it inaccessible, while leaving \texttt{getElement()} and \texttt{size()} unchanged; dually, (ii) applying the consumption abstraction to method \texttt{getElement()} makes it inaccessible, while leaving \texttt{setElement()} and \texttt{size()} unchanged; finally, (iii) applying both allows for just the \texttt{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.

More generally, Figure 4 shows the restrictions applied to return types (left) and argument types (right) in these cases, where columns range over possible receiver types, rows over the original type (return/argument type), and the table cell reports the corresponding restricted type — “deny” refers to the restriction making the method inaccessible. The figure shows that in the original signature, a (non-generic) type `R` unrelated to the formal argument `X` is left unchanged, while the restrictions simply allow `X` to appear as return type in the production receiver `C<T>` and as argument in the consumption receiver `C<-T>`. These restrictions are a consequence of rule [VAR] in Figure 1.

Notice that differently from variant parametric types, wildcards never make a method to be inaccessible due to such restrictions: in the above tables, they simply provide `Object` — or rather the least supertype that does not mention `X` substituted to `T` — instead of “deny” in return types (left) and the type for `null` instead of “deny” in argument types (right). Consequently, in their type system it happens that types `C<+Object>` and `C<*>` — or rather `C<?extends Object>` and `C<?>` in their syntax — impose the same restrictions.

### 3.3 Restriction with 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<+X>`, `C<-X>`, or `C<*X>`. Then, we have four cases for argument types and four for return types, which we exemplify through the 8 methods added to class `Vector<X>` as shown in Figure 5. These methods get and receive managers for `X`, which can be either in the standard form, or in their production/consumption/closed abstraction. We may assume, for instance, that method `setProd()` takes a manager `D<+X>` and uses it to get `X` elements from it, method `setCons()` takes a manager `D<-X>` and uses it to insert `X` elements in it; dually, `getProd()` is used to produce a manager `D<+X>` that is itself able to produce `X` elements, and so on. Figure 5 shows the result of applying access restriction in this case, which is reported in more general form in the tables of Figure 6.

Narrowing of return types is obtained 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<-X>` of `X` elements (through method `getCons()`) from a production manager `C<+T>` yields a manager `D<+T>`, 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<+T>`, and the restriction originally considered for the return type `D<-X>` have to be applied, which clearly result in a closed manager. This restriction is formally described by rule [REC+], where inner types are closed directly exploiting rule [VAR].

The case of argument types needs instead a more refined view. It turns out that only four cases of restrictions over the argument actually lead to an accessible functionality: (i)
signature to Vector<X>{
    ...
    void set(D<X> l);
    void setProd(D<+X> l);
    void setCons(D<-X> l);
    void setBlock(D<*X> l);
    D<X> get();
    D<+X> getProd();
    D<-X> getCons();
    D<*X> getBlock();
}

signature to Vector<*X>{
    ...
    void setBlock(D<*X> l);
    D<*X> getBlock();
}

signature to Vector<+X>{
    ...
    void setCons(D<*T> l);
    D<-T> getCons();
    void setBlock(D<*T> l);
    D<*T> getBlock();
}

signature to Vector<-X>{
    ...
    void setProd(D<*T> l);
    D<-T> getProd();
    void setBlock(D<*T> l);
    D<*T> getBlock();
}

Figure 5: Examples of Restrictions in nested parameterisations

<table>
<thead>
<tr>
<th>ret</th>
<th>C&lt;T&gt;</th>
<th>C&lt;+T&gt;</th>
<th>C&lt;-T&gt;</th>
<th>C&lt;*T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>D&lt;X&gt;</td>
<td>D&lt;T&gt;</td>
<td>D&lt;+T&gt;</td>
<td>D&lt;-T&gt;</td>
<td>D&lt;*T&gt;</td>
</tr>
<tr>
<td>D&lt;+X&gt;</td>
<td>D&lt;+T&gt;</td>
<td>D&lt;+T&gt;</td>
<td>D&lt;+T&gt;</td>
<td>D&lt;+T&gt;</td>
</tr>
<tr>
<td>D&lt;-X&gt;</td>
<td>D&lt;-T&gt;</td>
<td>D&lt;-T&gt;</td>
<td>D&lt;-T&gt;</td>
<td>D&lt;-T&gt;</td>
</tr>
<tr>
<td>D&lt;*X&gt;</td>
<td>D&lt;*T&gt;</td>
<td>D&lt;*T&gt;</td>
<td>D&lt;*T&gt;</td>
<td>D&lt;*T&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arg</th>
<th>C&lt;T&gt;</th>
<th>C&lt;+T&gt;</th>
<th>C&lt;-T&gt;</th>
<th>C&lt;*T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>D&lt;X&gt;</td>
<td>D&lt;T&gt;</td>
<td>deny</td>
<td>deny</td>
<td>deny</td>
</tr>
<tr>
<td>D&lt;+X&gt;</td>
<td>D&lt;+T&gt;</td>
<td>deny</td>
<td>D&lt;+T&gt;</td>
<td>deny</td>
</tr>
<tr>
<td>D&lt;-X&gt;</td>
<td>D&lt;-T&gt;</td>
<td>D&lt;-T&gt;</td>
<td>deny</td>
<td>deny</td>
</tr>
<tr>
<td>D&lt;*X&gt;</td>
<td>D&lt;*T&gt;</td>
<td>D&lt;*T&gt;</td>
<td>D&lt;*T&gt;</td>
<td>D&lt;*T&gt;</td>
</tr>
</tbody>
</table>

Figure 6: Restriction Rules for Nested Types (referred to a generic class C<X>)
accepting a blocked abstraction is allowed independently of the receiver, (ii) a consumer \( C^{-T} \) is allowed to accept a producer \( D^{+T} \), and (iii) a producer \( C^{+T} \) is allowed to accept a consumer \( D^{-T} \). The first case is justified considering that any manager of \( T \) should be allowed to consume a blocked 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 vice versa, 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 \( C^{+T} \) is not allowed to consume a producer \( D^{+T} \) itself, as this would allow \( C^{+T} \) to consume \( T \) elements through \( D \). Restriction rule [REC-] is responsible for this behaviour.

These parameterisation schemata actually do apply in practice. Consider for instance the following class \( \text{List}<X> \):

```java
class List<X>{
    ...
    X getHead(){...} // Standard getter for the head
    void setHead(X x){...} // Standard setter for the head
    List<X> getTail(){...} // Standard getter for the tail
    void setTail(List<X> l){...} // Standard setter for the tail
    void addAll(List<+X> l){...} // Copies from l to this
}
```

Variant versions of its instantiations have the signatures:

```java
interface to List<+T>{
    ...
    T getHead(); // Can produce the head
    List<+T> getTail(); // Can produce the tail, in a producer version
    // NO!! setTail Cannot consume a new tail
    // NO!! addAll Cannot consume from another tail
}
interface to List<-T>{
    ...
    void setHead(T x) // Can consume a new head
    // NO!! setTail Cannot consume a new tail
    List<-T> getTail() // Can produce the tail, in a consumer version
    void addAll(List<+T>) Can consume a producer
}
```

In particular, notice that access restrictions properly deal with the read/write interpretation over collections described in [10]. For instance, on a write-only \( \text{List} \) — a consumer \( \text{List}<-T> \) — other than updating the head (\( \text{setHead}() \)), one can access the tail (\( \text{getTail}() \)) retrieving a write-only list, which can then be recursively updated.
3.4 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. Moreover, we can expect the occurrence of these cases in programs to be less frequent, and in situations where our ontology is less intuitively applicable — e.g. when a manager accepts a manager of managers.

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

```java
class Reference<X>{
    ...
    X get(){...}
    void set(X x){...}
}
class Vector<X>{
    ...
    void addAllRefs(Vector<+Reference<+X>> v){...}
    Vector<Reference<X>> getAllRefs(){...}
}
```

Method `addAllRefs()` takes all the $X$ elements inside references inside the vector and store them into the receiver; dually, `getAllRefs()` produces a vector of references. By applying restrictions rules we obtain:

- $\Rightarrow_{+T}^{X:T}$
- $\Rightarrow_{+T}^{X:T}$
- $\Rightarrow_{-T}^{-T}$
- $\Rightarrow_{+T}^{-T}$

That is, we have the interfaces:

```java
signature to Vector<+T>{
    ...
    // NO!! addAllRefs
    Vector<+Reference<+T>> getAllRefs()
}
signature to Vector<-T>{
    ...
    void addAllRefs(Vector<+Reference<+T>> v){...}
    Vector<+Reference<-T>> getAllRefs(){...}
}
```

Through a production vector $Vector<+T>$, one is not allowed to invoke `addAllRefs()` (which has in fact a consumption character), but can invoke `getAllRefs()` which yields a production vector including production references (due to co-variance propagation). In the end, this
ensures that elements X cannot be consumed through the manager produced by Vector<+T>. On the consumption vector Vector<-T> invoking getAllRefs() is allowed yielding a production vector including consumption references, while setAllRefs() can be invoked by passing an element Vector<+Reference<+T>>. Noting that declaring argument to addAllRefs() to be production-only (at both levels) is necessary in order to make it accessible through a consumption vector.

4 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 case where methods are not necessarily used to “store” or “retrieve” elements through the class.

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 as well as wildcards. This is meant to pave the way towards a better understanding of the features of Java generics, and highlighting rooms for future extensions, developments, and applications.

References


on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’95), pages 156–168, Austin, TX, October 1995.


