Wild FJ

Mads Torgersen
University of Aarhus, Denmark
madst@daimi.au.dk

Erik Ernst
University of Aarhus, Denmark
eernst@daimi.au.dk

Christian Plesner Hansen
OOVM, Aarhus, Denmark
plesner@quenta.org

Abstract

This paper presents a formalization of wildcards, which is one of the new features of the Java programming language in version JDK5.0. Wildcards help alleviating the impedance mismatch between generics, or parametric polymorphism, and traditional object-oriented subtype polymorphism. They do this by quantifying over parameterized types with different type arguments. Wildcards take inspiration from several sources including use-site variance, and they could be considered as a way to introduce a syntactically light-weight kind of existential types into a main-stream language. This formalization describes the mechanism, in particular the wildcard capture process where the existential nature of wildcards becomes evident.

1. Introduction

This paper presents a formalization of the *wildcards* feature of the forthcoming release of the Java 2 Standard Edition Development Kit version 5.0 (J2SE 5), along with a description and motivation of the choices in the design process that gave rise to this particular formalization, and some formal safety properties.

The core idea of wildcards is quite simple. Plain generics in the Java programming language allows classes like the Java platform API class `List` to be parameterized with different element types, e.g., `List`⟨`Integer`⟩ and `List`⟨`String`⟩. Plain generics does not provide a general way to abstract over such different kinds of lists to exploit their common properties as lists, although polymorphic methods may provide an approximation of such an abstraction in specific situations. A wildcard is a special type argument '`?`' ranging over all possible specific type arguments, so that `List`⟨`?`⟩ is the type of all lists, regardless of their element type. Moreover, wildcard type arguments may be equipped with bounds, e.g., `List`⟨`? extends Number`⟩ is the type of all lists whose type argument is a subtype of `Number`. Wildcards make subtype polymorphism and parametric polymorphism play together more smoothly.

Subtype polymorphism originated in the object-oriented world, in SIMULA [11]. Parametric polymorphism—also known as genericity or generics—originated in the world of functional programming [21]. Both are powerful abstraction mechanisms, but there is a certain impedance mismatch between them, giving rise to the need for some kind of mediator.

In the mean time, various forms of parametric polymorphism have been added to a number of object-oriented languages over the past two decades [20, 28, 14]. Similarly, a process has been ongoing to extend the Java programming language with parametric polymorphism in the form of parameterized classes and polymorphic methods, i.e., classes and methods with type parameters. Moreover, a similar mechanism is specified in the upcoming future standard of C# [12, 13].

The process towards parametric polymorphism in the Java programming language involved many contributions from several research groups. A number of proposals were presented, including GJ and others [26, 2, 23, 4, 8]. The proposals explored the design space and advanced the field of programming language research. However, it became increasingly clear that parametric polymorphism on its own lacked some of the flexibility that we enjoy with object-oriented subtype polymorphism, thus calling for a better integration of the two.

These integration problems can be reduced in various ways, as demonstrated by several proposed designs [10, 9, 3, 5, 6]. An approach by Thorup and Torgersen [29], known as *use-site variance*, seems particularly successful in integrating the two types of polymorphism without unwanted effects on other parts of the language. The approach was later developed, formalized, and proven type sound by Igarashi and Viroli [17] in the context of the Featherweight GJ calculus [16]. This work addressed typing issues, but was never implemented full-scale. The present formalization exposes the differences between the outcome of the actual language design process and the formal model in [16], including a

FOOL 2005 15 January 2005, Long Beach, California

number of cases where the expressive power and flexibility have been enhanced.

The wildcards feature was developed in a joint project between a group of researchers (which includes the authors of this paper) from the University of Aarhus, Denmark, and Sun Microsystems, Inc., CA, in which we set out to investigate if the theoretical proposals could be adapted and matured to fit naturally into the Java language, and to initiate a robust and efficient implementation.

The project was successful in both regards. The core language mechanism has been reworked syntactically and semantically into wildcards. The wildcards construct has been fully integrated with other language features—in particular with polymorphic methods and type inference—and it has been applied extensively in the Java platform APIs, leading to enhanced expressiveness, simpler interfaces, and more flexible typing.

The implementation within the Java compiler is an extension of the generics implementation based on GJ. GJ enhances the type checker to handle type arguments of classes and methods, and it erases parametric information to produce type safe non-generic bytecode. Our implementation of wildcards further extends the type checker, and the associated modifications are part of the J2SE 5 release.

Although the implementation is specific to the Java programming language, wildcards should also be well suited for other object-oriented languages having or planning an implementation of parametric polymorphism, including e.g. C# and Scala [24].

The development process has raised a wealth of issues whose resolution have interesting theoretical consequences, which we describe by means of the formal calculus presented in this paper. The contributions described in this paper include:

- The formalization of the wildcard mechanism by means of a formal calculus in the style of Featherweight Java [16]
- A new usage of existential types, formalizing the so-called capture conversion process of J2SE 5
- Some safety properties, as well as an argument that safety properties of this calculus support similar properties of the actual implementation

Section 2 informally reviews wildcards by example. A more thorough presentation at the language level can be found in [30]. In Sect. 3 we describe the design process that lead to the calculus and its inference rules. Section 4 presents the calculus by its syntax, typing rules, dynamic semantics, and safety properties. Related work is covered in Sect. 5, and Sect. 6 concludes.

2. Some Examples

Without type parameters, a collection type such as `List` may hold any kind of objects. However, often all elements inserted into such a list have a non-trivial common supertype, and elements extracted from the list will be used under that type via a dynamic cast. The addition of type parameters allows for types on the form `List`$\langle T \rangle$ where T is the element type, thus making this convention explicit and ensuring type safety. Objects inserted must then have that type, and in return extracted objects are known to have that type, avoiding the unsafe cast. However, the improved precision in the typing makes it harder to treat a list as "just a list", even when the specific element type is irrelevant or need not be known exactly. For instance, a method could take a `List` as an argument and only be interested in clearing it or reading properties like the length where the type argument is of no importance. As long as the element type is known at the call site, this may be expressed with plain generics using a polymorphic method with a dummy type variable:

```
⟨X⟩ void m1(List⟨X⟩ list) { ... }
```

The solution is to give a name to the actual element type of the list and then ignore it in the body of the method. This works as long as inference can provide the actual type argument at each call site, but it is not a clean solution.

Moreover, plain generics cannot be used to declare a field or local variable with a type which denotes some kind of `List`, with no knowledge or only partial knowledge about the element type. This is obviously a problem in cases where the generic class provides many features which are independent of the actual type parameters, such as the now generic classes `java.lang.Class` and `java.lang.Enum` in the Java platform APIs; but it is also a problem in general because of the lack flexibility and abstraction in the plain generic type. This problem can be solved by using a *wildcard*, '`?`', in place of the type parameter:

```
private List⟨?⟩ list1;
```

This expresses that `list` is some type of list whose element type is irrelevant; with a bound, e.g., `? extends Number`, it would express that the element type is allowed to vary within a set of subtypes or supertypes (here: subtypes of `Number`). Similarly, a method can be declared to take an argument which is a `List` of anything:

```
void m2(List⟨?⟩ list) { ... }
```

The type `List`$\langle ? \rangle$ is a supertype of `List`$\langle T \rangle$ for any T, which means that any type of list can be assigned into the `list` field or given as an argument to the method. Moreover, we cannot put objects into the list since we do not know the actual element type. However, we are allowed to extract objects from it typed as `Object`: even though we do not know the exact type of the elements, we do know that they will be `Objects`.

In general, assume that the parameterized class `C` is declared as follows:

```
class C⟨X extends B⟩ { ... }
```

When calling a method on an object of type C⟨?⟩, methods that return X will have a return type taken from the declared bound of X, namely B, whereas a method that expects an argument of type X can not be called[1].

A wildcard '?' should not be considered to be the name of a specific type. For instance, the two occurrences of '?' in Pair⟨?,?⟩ are not assumed to stand for the same type, and even for the list shown above, the '?' in its type may stand for two different types before and after an assignment, as in list = new List⟨String⟩() followed by list = new List⟨Integer⟩().

Unbounded wildcards solve a number of problems with abstraction over generic types, but one capability known from polymorphic methods is missing, namely the ability to specify bounds on the type arguments. For example, plain generics can be used to define the following polymorphic method which takes an argument which is a list whose element type is a subtype of Number:

```
⟨X extends Number⟩ void m3(List⟨X⟩ list) {..}
```

As before, this only works for methods and not for fields, because it depends on a specific choice of X at each call site rather than expressing the *type* of those lists whose type argument is a subtype of Number. On the other hand, wildcards with *bounds* do express that type directly and may be used with fields, too:

```
List⟨? extends Number⟩ list2 = ...;
void m4(List⟨? extends Number⟩ list) { ... }
```

This expresses that list2 is some list whose element type is a subtype of Number, and the method m4 can be called with any list as long as its element type is a subtype of Number, including list2 even though the exact type argument of list2 is not known. As before, code using list2 and code in the body of m4 using the argument list cannot write (anything but null) to the lists since the actual element types are unknown, but we are now allowed to read objects under the type Number:

```
list2.set(0,new Float(0.0)); // Compile error
Number num = list2.get(0); // OK
```

Parameterized types with extends-bounded wildcards are *covariant* in the bounds: List⟨? extends Integer⟩ is a subtype of List⟨? extends Number⟩, which is only natural because all objects having the former type will also have the latter type. In our calculus, this is expressed by means of a subtyping relationship between existential types.

An extends-bounded wildcard sets an upper bound on the corresponding unknown type argument, and a *lower* bound can be expressed using super. For example, the type List⟨? super String⟩ is a supertype of List⟨T⟩ if T is a supertype of String. References of this type may refer to a List⟨String⟩, a List⟨Comparable⟨String⟩⟩ or a List⟨Object⟩[2]. In contrast to extends-bounds, super-bounds give rise to contravariant subtyping. For instance, the type Comparator⟨? super Number⟩ is a subtype of Comparator⟨? super Integer⟩.

A super-bound is useful in many places, for instance with consumers of objects such as Comparators. The Java platform class TreeSet represents a set by an ordered tree. One way to define the ordering is to construct the TreeSet with a specific Comparator object implementing the following interface:

```
interface Comparator⟨X⟩ {
  int compare(X fst, X snd);
}
```

To construct a TreeSet⟨String⟩ we provide a Comparator capable of comparing Strings. A Comparator⟨String⟩ could do this, but so could e.g. a Comparator⟨Object⟩, since Strings are Objects. Hence, an appropriate type is Comparator⟨? super String⟩, because it contains exactly the comparators which are able to compare strings.

The task of choosing appropriate types can be moved from the programmer to the compiler, in connection with type inference at polymorphic method invocations. Assuming the definitions below, type inference for the invocation choose(intSet, stringList) has to select a type for X that is a supertype of both Set⟨Integer⟩ and List⟨String⟩:

```
⟨X⟩ X choose(X a, X b) { ... }
Set⟨Integer⟩ intSet = ...
List⟨String⟩ stringList = ...
```

With plain generics, different parameterizations of the same class are incomparable, so the only such type is Object. This is so even though Set⟨X⟩ and List⟨X⟩ share the super interface Collection⟨X⟩. With wildcards it is possible to express this commonality through the type Collection⟨?⟩, and hence a more specific type than Object can be inferred. Moreover, since X is also the return type, the improved inference establishes that a Collection is returned, which allows the call site to use it as such without a cast.

In general, given two parameterized classes with different type arguments at the same parameter position, plain generics cannot unify the two to infer a type involving that parameter. With wildcards it becomes possible: ? can always be used and a more precise bound may be available. The result is more accurate type inference, and better preservation of information about returned results.

The abovementioned implementation of javac contains the improved type inference. In this paper we do not model the improved inference directly. Rather, we assume that in-

[1] Except with the argument null, which has all class types. This is of little practical value, and [17] actually disallows it, but it is a natural consequence of our subtyping rules and we did not want to complicate these rules in order to disallow corner cases.

[2] In fact it may even refer to a List⟨Comparable⟨?⟩⟩, since Comparable⟨?⟩ is also a supertype of String.

ference has taken place in such a way that type arguments to polymorphic method invocations have been chosen and expressed explicitly in the syntax; however, we also assume that this type inference inserts a special marker '$\star$' at the positions where the so-called wildcard capturing invocations of polymorphic methods occur. This makes it possible for us to model wildcard capture explicitly, cleanly separated from type inference in general.

3. Background

The technique of use-site variance was first proposed by Thorup and Torgersen in [29] as a kind of covariant "mode" of type parameter passing to generic classes, denoted by a leading '+' on type arguments. In this proposal, the type `List<+Number>` would be a common supertype for all `Lists` whose element types are subtypes of `Number`, e.g. `List<Integer>` and `List<Float>`. Since the exact element type of a `List<+Number>` is unknown at any specific time, the type system must statically forbid contravariant access, i.e. method calls with the element type in argument position, such as e.g. `add()`. This proposal is referred to as *use-site covariance* because it applies a covariance annotation at the site where a generic class is *used* rather than where it is *declared*. Covariant types also enjoy a covariant suvbtype relation, e.g., `List<+Integer> <: List<+Number>`.

The proposal was informal, but in [17] Igarashi and Viroli not only formalized the mechanism, but also introduced a dual contravariant mode, denoted by '`-`' and a combined so called "bivariant" mode denoted by '`*`'. Moreover they produced some safety results including soundness. This proposal we refer to as *variant parametric types*, following the terminology of Igarashi and Viroli.

The wildcards project started out as an attempt to integrate the typing benefits offered by variant parametric types in the full-blown Java language, to see if a construct suitable for inclusion in a subsequent Java release could be developed. Indeed the first versions used a syntax similar to the proposal of Igarashi and Viroli.

However, we quickly felt that the type system of Igarashi and Viroli was too constraining in practice. Especially the "generification" of the core libraries (such as the Collection API) led to this conclusion.

A number of seemingly different limitations—discussed in the following subsections—turned out to share a common solution: a pervasive redesign which became reflected in both the syntax, the implementation and the name of the mechanism. "Wildcards", as it came to be called, unifies the three variant modes of parameterization into a single one, focusing with the wildcard parameter "`?`" on the fact that the particular type argument is unknown, and reducing the "variances" to optional bounds delimiting the range of the wildcard. In our calculus, the somewhat arbitrary restriction that bounds should be *either* upper or lower is also lifted.

In this section we investigate how the so-called *capture conversion* which is described in the Java Language Specification (JLS for short) [15] addresses all these issues by introducing symbolic representatives of the unknown types "hidden" behind wildcards, in the form of synthetic type variables generated "under the hood" by the compiler. This is a rather unconventional approach, and naturally raises the question: "Is it safe?" It is therefore interesting to investigate it formally, and we proceed by describing how this approach in our calculus is transformed to an essentially equivalent one based on a variant of existentially quantified types.

3.1 Capture Conversion

Inspired by existential types, the type system of Igarashi and Viroli uses an approach of "opening" variant parameters to type variables with appropriate bounds. Consider the following declarations:

```
class Box<X extends Object> {
  void put(X x) { ... }
  X get() { ... }
}
Box<+T> box;
```

To type the call `box.get()` the type signature of the method `get()` is looked up in a class `Box<Y>` where the fresh type variable `Y` is designated by the type environment of the lookup to have the upper bound `T`. Thus, the found type of `get()` is `()`$\rightarrow$`Y`. However, to get the type of `box.get()` we must "close" the return type so as not to contain any of the fresh type variables (i.e. `Y` in this case). This is done by promoting the return type `Y` to its (`Y`-free) upper bound `T`.

If we tried the same with a `Box<-T>` we would find that no promotion was defined for the close operation, so calling the `get()` method would not be well typed.

The type system of the Java programming language (as described in the JLS), takes a more radical approach, called "capture conversion". The "open" process above is applied to every expression. The fresh type variables are made globally accessible, and the opened type is never closed again. Given the above `Box` class and a corresponding declaration

```
Box<? extends T> box;
```

The type of the expression `box` is then `Box<Y>` where `Y <: T` is a globally fresh type variable. By standard rules the type of `box.get()` is thus `Y`.

The justification for this approach is as follows. If an expression has type `Box<? B>` where `B` is a possible `extends`- or `super`-bound, then at runtime this expression must evaluate to a `Box` of *some* specific type `S` satisfying the bound `B`. The freshly introduced type variable `Y` is simply a symbolic representation of that type `S`, summarizing what we know about it. Compared to Igarashi and Viroli, we maintain names for the unknown types such as `S`, whereas the typing rules in [17] apply both open and close immediately

in the same rule, thus forgetting about some relations between types.

This approach means that the type of an expression never contains wildcards at the top level. Therefore, typing of expressions may in a lot of situations be performed exactly as with plain generics. Some consequences:

- Subtyping is defined so that top-level wildcards only occur on the right-hand site, whereas the left hand side is always capture converted. This simplifies the subtype relation a good deal compared to variant parametric types.
- Type inference does not need to be complicated by wildcards in expression types, although it must of course still be improved to be able to *infer* types with wildcards.
- The type system must be extended a little bit to be able to handle type variables with lower bounds (incurred by super-bounded wildcards).

On top of this, however, capture conversion leads to a number of "free" generalizations of the type system which address a number of practical problems with variant parametric types. These will be discussed one-by-one in the following.

3.1.1 Read-only and write-only

As we saw above, the approach of Igarashi and Viroli in effect stipulates the removal of certain methods from the interface of classes with variant parameters. For instance, `Box<-T>` is considered write-only, and therefore does not give access to the `get()` method. We found the read-only/write-only interpretation of co- and contravariance to be somewhat misleading, since you can still e.g. modify a "read-only" list by e.g. calling its `clear()` method.

But furthermore it is also unnecessarily restrictive. There is no reason why we shouldn't be able to read from a `Box<? super T>`, even if we know only that we are reading some kind of `Objects`. We therefore extend capture conversion so that the fresh type variable acquires not only the bounds of the wildcard, but also the declared bounds of the corresponding type argument in the generic class. An example:

```
Box<? super T> contrabox;
```

The type of the expression `contrabox` is `Box<Z>` where `Z` is globally fresh *and* `Z :> T` (the bound from the wildcard) *and* `Z <: Object` (the bound from the declaration of `Box`). Therefore the assignment

```
Object o = contrabox.get();
```

is allowed, because the type of `contrabox.get()` is `Z`, and `Z <: Object` by declaration.

3.1.2 F-bounds

At first glance, one might envision the above expressiveness obtained in the Igarashi/Viroli system by allowing the close operation to promote a type variable such as `Z` above to its upper bound. However, consider the situation where the declared bound of the type parameter is an F-bound; i.e. refers to the parameter itself, as in:

```
class PandoraBox<X extends Box<X>>
  extends Box<X> { ... }
PandoraBox<?> pbox;
```

We may open `pbox` to the type `PandoraBox<Z>` where `Z <: Box<Z>`, but how do we close the type `Z` of the expression `pbox.get()`? `Z` occurs in its own upper bound, so there is no obvious `Z`-free supertype (save the uninteresting `Object`). Keeping `Z` around means that the type of `pbox.get()` remains precisely described, and allows assignments such as:

```
Box<?> box = pbox.get();
```

3.1.3 Type promotion

Even when the close operation does succeed, it sometimes needs to discard some type information in order to provide an expressible type. More precisely, whenever a close operation on a type seeks to eliminate a type variable that occurs as a type argument of that type, the type variable must be re-promoted to a variant parameter. If however it occurs more than once, the information is lost that all occurrences denoted the same type. Even worse, if it occurs in nested parameterizations, the promotion must be applied also to enclosing types. Let us give `Box` a few more methods:

```
class Box<X extends Object> {
  Pair<X,X> asPair() { ... }
  Box<Box<X>> nest() { ... }
}
```

Given a covariant `Box<+T> box`, the type rules of variant parametric types would promote the type of `box.asPair()` to `Pair<+T,+T>`, forgetting that the two elements of the pair are known to have the same type. In the expression `box.nest()` the naive promotion to `Box<Box<+T>>` does not work, since this is not necessarily a supertype of `Box<Box<Z>>` for any `Z <: T`. Hence, the result type must be further weakened to `Box<+Box<+T>>`, forgetting the invariance of the outer `Box`. Igarashi and Viroli describe more contrived situations in which no most specific supertype exists, so that an arbitrary choice must be made.

Wildcards in Java sidestep all these issues by avoiding the promotion step altogether. With a covariant `Box<? extends T> box` the expression `box` has the type `Box<Z>` for a globally fresh `Z` where `Z <: T`. Thus, `box.asPair()` simply has the type `Pair<Z,Z>`, maintaining the information that both components of the pair have the same type, whereas `box.nest()` retains its invariance with the type `Box<Box<Z>>`.

3.1.4 Wildcard capture

A place where capture conversion has a large impact is in the type inference applied to polymorphic method calls in Java. A function like

```
<X> List<X> immutableList(List<X> l)
```

from the `java.util.Collections` class seems to be callable only with an invariant (non-wildcard) list. Indeed that is the case with variant parametric types. In order to provide for variant lists one would have to give the weaker signature

```
List<*> immutableList(List<*> l)
```

which when called with invariant `Lists` would lose the information that the returned list has the same element type as the argument. Alternatively one would have to provide both signatures (plus one each for co- and contravariant types!) to cater for all possibilities with code duplication as a result. According to personal communication [25] this issue was part of the reason for abandoning the adoption of use-site variance in the Scala language.

However, with wildcards a `list` declared as `List<?> list` will have the type `List<W>` for fresh type variable `W`, so in a call `immutableList(list)`, the method will simply be inferred to have the type parameter `W`, wherefore the return type of the method call will be `List<W>`. This means that polymorphic methods can be used as a means to provide names for the types hidden behind wildcards within the scope of their body, much in the same way as the `open` operation on existential types.

As an example consider a stack implementation and an external function to swap the top elements:

```
class Stack<X> {
  void push(X x) { ... }
  X pop() { ... }
}
private <Y> void doSwap(Stack<Y> s) {
  Y y1 = s.pop();
  Y y2 = s.pop();
  s.push(y1);
  s.push(y2);
}
void swap(Stack<?> s) { doSwap(s); }
```

Here the polymorphic `doSwap()` function uses its type argument `Y` to type the temporary variables of its body. However, the applicability of the method to all stacks is advertised by the much simpler signature of the `swap()` function, which uses wildcard capture to delegate to the private `doSwap()`.

Real examples of this approach can be found in the `java.util.Collections` utility methods `shuffle()` and `reverse()`.

Capture may also involve multiple type variables with non-trivial, mutually dependent bounds. Consider the following classes:

```
class Node<N extends Node<N,E>,
    E extends Edge<N,E>> { ... }
class Edge<N extends Node<N,E>,
    E extends Edge<N,E>> { ... }
```

```
class Graph<N extends Node<N,E>,
    E extends Edge<N,E>> {
  N n; E e;
  Graph(N n, E e) { this.n=n; this.e=e; }
}
```

Using these classes we can write some polymorphic methods and use them:

```
<N extends Node<N,E>, E extends Edge<N,E>>
void copy(Graph<N,E> g) {
    create(g.n,g.e);
}
<N extends Node<N,E>, E extends Edge<N,E>>
Graph<N,E> create(N n, E e) {
    return new Graph<N,E>(n,e);
}
Graph<?,?> builder() {
    Graph<?,?> g = ...;
    return copy(g);
}
```

The point is that it is possible[3] to package multiple mutually dependent types as type arguments to a wrapper object, `g`; to use types with wildcards, `Graph`$\langle$`?,?`$\rangle$, to hide the precise type arguments; and to use wildcard capture, in the call to `copy`, to regain named use of these type arguments. Thus, consistency is ensured without burdening client code with dependencies upon the exact values of those mutually dependent type arguments.

3.2 Wild FJ

In order to explain and reason about the properties of wildcards we have developed a descendant of Featherweight GJ (FGJ) formalizing the type rules described above in a small and manageable theoretical setting. Here we discuss our reasons for shaping the calculus the way we do, whereas Section 4 presents the details of the calculus, which we call *Wild FJ* (WFJ).

We take FGJ as a starting point rather than the core calculus developed by Igarashi and Viroli for variant parametric types, because our modeling of wildcards is fundamentally different from their treatment of variance. As compared to FGJ we have omitted constructors, which are only there in order to be a subset of Java, and we have skipped the treatment of casts, which in FGJ play a central role in proving correctness of the erasure approach to generics implementation, but shed no light on wildcards.

3.2.1 Bounds

Compared to Java (with wildcards) as well as FGJ we have generalized the treatment of bounds. In general, everywhere bounds occur we allow (but do not require) both upper and

[3] In fact, the current J2SE 5 compiler contains a bug which makes it reject this example, but the formalization in this paper does support this kind of wildcard capture, and a bug fix for the compiler is upcoming.

lower bounds simultaneously; i.e. in the declarations of type variables for classes and methods, in wildcards and in type environments. We need combined upper and lower bounds to model synthetic type variables anyway, and this scheme has proven to extend nicely (i.e. with negligible overhead) to the rest of the calculus.

The elements of a type environment are therefore of the form $X \in \mathtt{B}_{\lhd}\mathtt{B}_{\rhd}$ where $\mathtt{B}_{\lhd}$ and $\mathtt{B}_{\rhd}$ are optional upper and lower bounds of the form $\lhd\mathtt{S}$ and $\rhd\mathtt{T}$ (read `extends S` and `super T`) for types `S` and `T`.

3.2.2 Modeling capture conversion

Capture conversion is described above as introducing fresh type variables into the global scope as a result of local typing. While this provides a tremendously lightweight implementation strategy for a compiler written in an object-oriented language, it is not in good accordance with the traditional compositional nature of formal typing rules.

In order to approach compositionality, we may observe that, being globally fresh, type variables introduced by capture conversion can only occur in the type of an expression if they are introduced by a subexpression (or the expression itself). It is therefore reasonable to let the typing rules return these new type variables and their bounds along with the type of a given expression. The type variables can then be propagated upwards as necessary.

A set of unknown, bounded type variables (i.e. a type environment) wrapped with a type in which they occur free: this calls for an existential notation, as in:

$$\exists(\mathtt{Z} \lhd \mathtt{T}).\mathtt{Box<Z>}$$

Note that this does not mean that we add existential types to the syntax of Wild FJ. Instead we keep a separate level of semantic types which are syntactic types wrapped up with type environments in this existential fashion. Thus, existential types are always at the top-level and may not occur nested within other types.

The equivalent of capture conversion is a helper function *snap*, which given a syntactic type converts all its top-level wildcards into type variables and produces a semantic (existential) type instead. This process of conversion into a more general type scheme models the fact, odd as it may seem, that expressions in Java can have types that are not expressible within the language itself. While it is at odds with traditional notions of linguistic purity, it does not create problems as such in our calculus. It does however suggest that the transformation could be performed as a preprocessing step of Wild FJ source code into an alternative syntax ("$\exists$J"?) based entirely on existential types. In order not to depart too much from the specification of Java wildcards we choose however to perform the translation "on demand" within the type system itself.

3.2.3 Type inference

FGJ does not model type inference as part of the calculus. Rather, type parameters for polymorphic methods are explicit, thought to have been inferred by a preprocessing step. Indeed, building type inference into the type system itself would break its soundness proof. This is because soundness is based on a subject-reduction theorem which takes the type of the program at every evaluation step and shows that it is a subtype of the previous one. Type inference guesses type parameters based on the best available type information, but as type information improves during program evaluation, inference might guess differently (better) for each step, making subsequent types of the evaluating program incompatible with previous ones.

Unfortunately, wildcard capture as described above hinges on the ability of type inference to pass on synthetic type variables that by design cannot be expressed explicitly in syntax. This seems to call for type inference to be present as part of the type system itself, which conflicts with the above line of argument.

Our solution is a compromise: we generally expect type inference to be performed as a batch process, but allow it to insert a special marker '$\star$' for method type arguments that are to be determined by wildcard capture. The type system then contains a severely watered-down version of inference, represented by the *capture* function, that serves to replace these stars with types derived from the value arguments to the method. Since this involves no guessing, and hence no opportunity to improve over time (it is already precise) it is not in conflict with the above observations.

3.2.4 Subtyping

In the Java compiler, the type system is of course only employed before the execution of the program. This means especially that subtyping is only applied between the types of expressions (expressed as capture converted types without top-level wildcards) as subtypes and the types of variables and method arguments, to which the expressions are assigned, as supertypes.

In a calculus, however, in order to prove subject reduction we must show subtyping between the types of two different expressions, namely the program before and after a given evaluation step. These existential types may be very different, having chosen different fresh type variables during the typing process etc. We might have chosen to deal with this by stating the subject reduction property in much more convoluted terms, prescribing how the two types were to be matched up. However, convoluted properties involve convoluted proofs, and it would then remain to be shown that this was in fact still a subject reduction property despite its alternative formulation.

We have chosen instead to describe subtyping itself in terms of existential types. Rather than having capture converted types on the left hand side as in Java, we have existen-

tial types on both sides. Note that this means that, in contrast to Igarashi and Viroli [17], we do not need subtyping rules concerned with types with wildcard arguments.

The subtyping rules are rather standard for existential types, which means that they deal naturally with differently-named or unused type variables. Thus, subject reduction can be stated in straightforward terms. This does mark a departure from subtyping as described by the JLS, albeit one that actually simplifies the rules, and it remains an interesting application of the calculus to show that its subtyping relation does in fact generalize the JLS semantics.

4. The Calculus

In the following we present the syntax, semantics, and typing rules for the WFJ calculus.

4.1 Syntax and Auxiliary Functions

In this section we present the syntax of the WFJ calculus and the auxiliary functions used by the type system and the dynamic semantics. The syntax of the WFJ calculus is shown in Fig. 1. We generally follow well-known notational conventions, especially from [16]. In particular, we indicate the syntactic structure of an expression by means of metavariables, which are the non-terminals in the grammar along with the lexical metavariables shown at the bottom of Fig. 1. We use overbars, as in $\overline{\mathtt{T}}$, to indicate a sequence of elements.

Expressions are variables, field lookup, method invocations ($\mathtt{e.<\overline{P}>m(\overline{e})}$), or object creation ($\mathtt{new\ N(\overline{e})}$). Method invocations are always annotated with explicit type arguments, which may be types or the special marker $\star$ that indicates a request for wildcard capture. A type is either a type variable ($\mathtt{X}$) or an application of a parameterized class to type arguments ($\mathtt{C<\overline{A}>}$), which are again types or wildcards. We sometimes write $\mathtt{C}$ for $\mathtt{C<>}$. A subset of the types must be separately recognizable, namely class types $\mathtt{N}$ applied solely to types, not wildcards, and the union $\mathtt{K}$ of these types and type variables. Bounds ($\mathtt{B}$) consist of upper and lower bounds, and they may be present or absent ($\bullet$). Class declarations ($\mathtt{Q}$) contain type variables with bounds, a superclass, field and method declarations. A constructor is implicit, taking one argument for each field. Method declarations ($\mathtt{M}$) contain type variables with bounds, return type, method name, arguments, and a body that returns the result of evaluating an expression.

Type environments Δ map type variables to bounds. An entry is written $\mathtt{X} \in \mathtt{B}$ and type variables are looked up by application $\Delta(\mathtt{X})$. $dom(\Delta)$ signifies the domain of Δ, and type environments are concatenated using juxtaposition $\Delta\Delta'$. Since type environments are mappings, we require that concatenated environments have disjoint domains.

Variable environments Γ map formal method arguments to their syntactic types. An entry is written $\mathtt{x}$: $\mathtt{T}$, and variables are looked up by application $\Gamma(\mathtt{x})$.

Finally, existential types are of the form $(\exists\Delta.\mathtt{K})$, where the type variables in $dom(\Delta)$ may occur freely in $\mathtt{K}$. Throughout the calculus we assume an implicit α-conversion rule for existential types, allowing for a consistent renaming of the type variables in Δ, as well as the adding and removal of type variables to Δ which are unused (do not occur free) in $\mathtt{K}$. In the following, sometimes we concatenate type environments originating from several existential types. Here the α-conversion ensures that type variables are transparently renamed when necessary to make the concatenation possible, i.e., to ascertain that the domains of the concatenated environments are disjoint.

The auxiliary functions are shown in Fig. 2. The function *fields* is used to obtain the fields of a given class along with their types. It yields the empty result for the predefined class `Object`, and for a class type it inserts the actual type arguments $\overline{\mathtt{T}}$ to get the fields of the class itself, adding them to the inherited fields from the recursive application of *fields*.

The function *mtype* provides the type of a method in context of a class type, represented by its formal type arguments with bounds followed by the types of its value arguments and its return type. The two cases handle methods defined directly in the class and methods inherited from a superclass, respectively.

The function *mbody* is used to obtain the list of arguments ($\overline{\mathtt{x}}$) along with the method body ($\mathtt{e}_0$), again in context of a class type and with two cases corresponding to the ones for *mtype*. Note that *mbody* is invoked with types ($\overline{\mathtt{V}}$) rather than the mixture of types and $\star$ ($\overline{\mathtt{P}}$), because wildcard capture has already occurred when this function is applied.

The function $bound^1$ extracts the declared upper bound of a type variable $\mathtt{X}$ from an environment Δ. If $\mathtt{X}$ exists in Δ, but the upper bound is absent, the result is `Object`. The function $lbound^1$ is similar, but returns the declared lower bound and is undefined when the lower bound is absent.

The function *bound* recursively applies $bound^1$ to find the least class type that is an upper bound of the given class type or variable, based on the environment Δ. A short hand for applying *bound* to existential types is also provided.

The calculus is based on existential types, and they are created from syntactic types by means of the *snap* function. The effect of *snap* on class types is to convert all top-level wildcard arguments to type variables, which are then inserted in the type environment of the resulting existential type. The actual conversion is done by the helper function *fix*, which scans through a list of type arguments producing new type variables and bounds for every wildcard it encounters. The bounds are obtained by calling *merge* with the wildcard bound (if present) and the declared bound (if present) of the given type parameter. Given both, *merge* will select the wildcard bound, which is ensured by well-formedness to be a tighter bound than the declared bound. The type variables and bounds produced by *fix* and *merge* are reassembled by

Syntax:

$\mathtt{d},\mathtt{e}$	::=	$\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)}$	*expressions*
$\mathtt{S},\mathtt{T},\mathtt{U},\mathtt{V}$	::=	$\mathtt{C<}\overline{\mathtt{A}}\mathtt{>} \mid \mathtt{X}$	*types*
$\mathtt{A}$	::=	$\mathtt{T} \mid \mathtt{?\ B}$	*type arguments*
$\mathtt{P}$	::=	$\mathtt{T} \mid \star$	*method type parameters*
$\mathtt{N}$	::=	$\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$	*class types*
$\mathtt{K},\mathtt{L}$	::=	$\mathtt{N} \mid \mathtt{X}$	*class types and var.s*
$\mathtt{B}$	::=	$\mathtt{B}_{\triangleleft}\ \mathtt{B}_{\triangleright}$	*bounds*
$\mathtt{B}_{\triangleleft}$	::=	$\triangleleft\mathtt{T} \mid \bullet$	*upper bounds*
$\mathtt{B}_{\triangleright}$	::=	$\triangleright\mathtt{T} \mid \bullet$	*lower bounds*
$\mathtt{Q}$	::=	$\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}\triangleleft\mathtt{N\ \{}\ \overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;}\ \overline{\mathtt{M}}\ \mathtt{\}}$	*class declarations*
$\mathtt{M}$	::=	$\mathtt{<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>\ T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{ return\ e;\ \}}$	*method declarations*
$\mathtt{E},\mathtt{F}$	::=	$\exists\Delta.\mathtt{K}$	*existential types*
Δ	::=	$\emptyset \mid \Delta,\mathtt{X}\in\mathtt{B}$	*type environments*

Metavariables:

$\mathtt{f}$	*field names*
$\mathtt{C},\mathtt{D}$	*class names*
$\mathtt{X},\mathtt{Y},\mathtt{Z}$	*type variables*
$\mathtt{x},\mathtt{y},\mathtt{z}$	*variables*
$\mathtt{m}$	*method names*

Figure 1. Syntax of the WFJ calculus

snap into the type environment used in the resulting existential type.

Note that the work of *snap* and its helper function is purely syntactic—there is no analysis of the semantics of the involved types. This supports the suggestion of Sect. 3.2.2 that the translation from syntactic to existential types would be well suited for a preprocessing step instead of being embedded in the calculus.

Finally, the *capture* function formalizes the wildcard capture process. Its role is to perform the simple type inference of method type arguments which is needed when a provided argument is '$\star$', as explained in Sect. 3.2.3. This function is used in a context where there is an invocation of a method, which in this calculus implies that there is a list of actual type arguments as well as a receiver, a method name, and some actual value arguments. The intuition behind the five arguments of $\mathit{capture}_{\Delta}(\mathtt{P},\mathtt{X},\overline{\mathtt{T}},\overline{\mathtt{K}})$ are as follows:

- The type environment Δ provides bounds for the free type variables in the remaining four arguments.
- $\mathtt{P}$ is an actual type argument which is being translated. It is a type $\mathtt{U}$ or a capture marker $\star$; the former is passed through unchanged, and the latter is resolved using the rest of the arguments.
- $\mathtt{X}$ is the formal type argument taken from the method declaration. It is used to find a location in the value argument types of the method where the 'captured' type can be found.
- $\overline{\mathtt{T}}$ is the list of formal value argument types, and it is searched in order to find the position of the formal type argument $\mathtt{X}$ as a top-level type argument to a class $\mathtt{C<}\ldots\mathtt{X}\ldots\mathtt{>}$.
- $\overline{\mathtt{K}}$ is the list of actual value argument types. Using the above information, this is where the result is extracted.

In summary, the wildcard capture process proceeds as follows: Choose a formal value argument type $\mathtt{T}_i$ which is a class type $\mathtt{C<}\overline{\mathtt{A}}\mathtt{>}$ using $\mathtt{X}$ as one of its top-level type arguments ($\mathtt{A}_j$). Find the class type at the same position in the list of actual argument types $\mathtt{K}_j$. Find a supertype $\mathtt{C<}\overline{\mathtt{A}'}\mathtt{>}$ of $\mathtt{K}_j$ which is an instance of the class type $\mathtt{C}$ where we found $\mathtt{X}$ above. Finally, select the type argument in $\overline{\mathtt{A}'}$ from the position j where $\mathtt{X}$ was found in $\overline{\mathtt{A}}$. If $\mathtt{A}'_j$ is a type then it is the result of the capture operation.

Note that if $\mathtt{X}$ occurs in more than one position, the type rule for method invocation ensures that the types found at these positions are equal.

4.2 Subtyping rules

The WFJ subtype rules are shown in Fig. 3. Subtyping generally takes place in the realm of existential types ($\exists\Delta.\mathtt{K}$), and subtyping for syntactic types ($\mathtt{T}$) is lifted to existential types by means of *snap*, as shown in rule (WS-SYNTACTIC).

The existential subtype rules are as follows. (WS-TRANS) gives transitivity. (WS-VAR) and (WS-LVAR) perform environment lookup. (WS-SUBCLASS) uses the declared subclassing relation to construct a corresponding subtyping

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet \quad \text{(F-OBJECT)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \lhd \texttt{N}\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\ \} \qquad \mathit{fields}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}}\ \overline{\texttt{g}},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}} \quad \text{(F-CLASS)}$$

Method type lookup:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \lhd \texttt{N}\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\ \} \qquad \texttt{<}\overline{\texttt{Y}}\ \overline{\texttt{B}}'\texttt{>}\ \texttt{U m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\ \{\ldots\} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\ \overline{\texttt{B}}'\texttt{>}\overline{\texttt{U}} \to \texttt{U})} \quad \text{(MT-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \lhd \texttt{N}\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\ \} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \quad \text{(MT-SUPER)}$$

Method body lookup:

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \lhd \texttt{N}\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\ \} \qquad \texttt{<}\overline{\texttt{Y}}\ \overline{\texttt{B}}'\texttt{>}\ \texttt{U m(}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\ \{\ \texttt{return e}_0\texttt{;}\ \} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{<}\overline{\texttt{V}}\texttt{>m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{x}}.[\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e}_0} \quad \text{(MB-CLASS)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{B}}\texttt{>} \lhd \texttt{N}\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\ \} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mbody}(\texttt{<}\overline{\texttt{V}}\texttt{>m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mbody}(\texttt{<}\overline{\texttt{V}}\texttt{>m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})} \quad \text{(MB-SUPER)}$$

Upper bounds:

$$\frac{\Delta(\texttt{X}) = \bullet\ \texttt{B}_\triangleright}{\mathit{bound}^1_\Delta(\texttt{X}) = \texttt{Object}} \qquad \frac{\Delta(\texttt{X}) = \lhd\texttt{T}\ \texttt{B}_\triangleright}{\mathit{bound}^1_\Delta(\texttt{X}) = \texttt{T}} \qquad \frac{\Delta(\texttt{X}) = \texttt{B}_\lhd \triangleright \texttt{T}}{\mathit{lbound}^1_\Delta(\texttt{X}) = \texttt{T}}$$

$$\frac{\mathit{bound}^1_\Delta(\texttt{X}) = \texttt{C<}\overline{\texttt{A}}\texttt{>}}{\mathit{bound}_\Delta(\texttt{X}) = \texttt{C<}\overline{\texttt{A}}\texttt{>}} \qquad \frac{\mathit{bound}^1_\Delta(\texttt{X}) = \texttt{Y} \quad \mathit{bound}_\Delta(\texttt{Y}) = \texttt{C<}\overline{\texttt{A}}\texttt{>}}{\mathit{bound}_\Delta(\texttt{X}) = \texttt{C<}\overline{\texttt{A}}\texttt{>}} \qquad \mathit{bound}_\Delta(\texttt{C<}\overline{\texttt{A}}\texttt{>}) = \texttt{C<}\overline{\texttt{A}}\texttt{>}$$

$$\frac{\mathit{bound}_{\Delta\Delta'}(\texttt{K}') = \texttt{C<}\overline{\texttt{A}}\texttt{>} \qquad \mathit{snap}(\texttt{C<}\overline{\texttt{A}}\texttt{>}) = \exists\Delta''.\texttt{N}}{\mathit{bound}_\Delta(\exists\Delta'.\texttt{K}') = \exists\Delta'\Delta''.\texttt{N}}$$

Creating existential types:

$$\mathit{snap}(\texttt{X}) = \exists.\texttt{X} \qquad \frac{\texttt{class C<}\overline{\texttt{Y}}\ \overline{\texttt{B}}_0\texttt{>} \lhd \texttt{N}\ \{\ldots\} \quad \mathit{fix}(\overline{\texttt{A}}, \overline{\texttt{B}}_0) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}}) \quad \Delta = \overline{\texttt{X}} \in [\overline{\texttt{T}}/\overline{\texttt{Y}}]\overline{\texttt{B}}}{\mathit{snap}(\texttt{C<}\overline{\texttt{A}}\texttt{>}) = \exists\Delta.\texttt{C<}\overline{\texttt{T}}\texttt{>}}$$

$$\frac{\mathit{fix}(\overline{\texttt{A}}, \overline{\texttt{B}}_0) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}})}{\mathit{fix}(\texttt{T} :: \overline{\texttt{A}}, \texttt{B}_0 :: \overline{\texttt{B}}_0) = (\texttt{T} :: \overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}})} \qquad \frac{\mathit{fix}(\overline{\texttt{A}}, \overline{\texttt{B}}_0) = (\overline{\texttt{T}}, \overline{\texttt{X}}, \overline{\texttt{B}}) \qquad \texttt{X fresh}}{\mathit{fix}(\texttt{?}\ \texttt{B} :: \overline{\texttt{A}}, \texttt{B}_0 :: \overline{\texttt{B}}_0) = (\texttt{X} :: \overline{\texttt{T}}, \texttt{X} :: \overline{\texttt{X}}, \mathit{merge}(\texttt{B}, \texttt{B}_0) :: \overline{\texttt{B}})}$$

$$\mathit{fix}(\bullet, \bullet) = (\bullet, \bullet, \bullet)$$

$$\mathit{merge}(\bullet\ \bullet, \texttt{B}_\lhd\ \texttt{B}_\triangleright) = \texttt{B}_\lhd\ \texttt{B}_\triangleright \qquad \mathit{merge}(\lhd\texttt{T}\ \bullet, \texttt{B}_\lhd\ \texttt{B}_\triangleright) = \lhd\texttt{T}\ \texttt{B}_\triangleright \qquad \mathit{merge}(\bullet \triangleright \texttt{S}, \texttt{B}_\lhd\ \texttt{B}_\triangleright) = \texttt{B}_\lhd \triangleright \texttt{S}$$
$$\mathit{merge}(\lhd\texttt{T} \triangleright \texttt{S}, \texttt{B}_\lhd\ \texttt{B}_\triangleright) = \lhd\texttt{T} \triangleright \texttt{S}$$

Capture:

$$\mathit{capture}_\Delta(\texttt{U}, \texttt{X}, \overline{\texttt{T}}, \overline{\texttt{K}}) = \texttt{U} \qquad \frac{\texttt{T}_i = \texttt{C<}\overline{\texttt{A}}\texttt{>} \qquad \texttt{A}_j = \texttt{X} \qquad \Delta \vdash \texttt{K}_i\ \texttt{<:}\ \texttt{C<}\overline{\texttt{A}}'\texttt{>} \qquad \texttt{A}'_j = \texttt{V}}{\mathit{capture}_\Delta(\star, \texttt{X}, \overline{\texttt{T}}, \overline{\texttt{K}}) = \texttt{V}}$$

Figure 2. Auxiliary functions

$$\Delta \vdash \exists\Delta'.\mathtt{X} <: \exists\Delta'.\mathit{bound}^1_{\Delta\Delta'}(\mathtt{X}) \quad \text{(WS-VAR)} \qquad \Delta \vdash \exists\Delta'.\mathit{lbound}^1_{\Delta\Delta'}(\mathtt{X}) <: \exists\Delta'.\mathtt{X} \quad \text{(WS-LVAR)}$$

$$\frac{\Delta \vdash \mathtt{E} <: \mathtt{E}'' \qquad \Delta \vdash \mathtt{E}'' <: \mathtt{E}'}{\Delta \vdash \mathtt{E} <: \mathtt{E}'} \quad \text{(WS-TRANS)} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\}}{\Delta \vdash \exists\Delta'.\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <: \exists\Delta'.[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \quad \text{(WS-SUBCLASS)}$$

$$\frac{\Delta\Delta' \vdash \overline{\mathtt{U}} \in [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{B}}}{\Delta \vdash \exists\Delta'.[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{K} <: \exists\overline{\mathtt{X}}\in\overline{\mathtt{B}}.\mathtt{K}} \quad \text{(WS-ENV)} \qquad \frac{\Delta \vdash \mathit{snap}(\mathtt{S}) <: \mathit{snap}(\mathtt{T})}{\Delta \vdash \mathtt{S} <: \mathtt{T}} \quad \text{(WS-SYNTACTIC)}$$

$$\Delta \vdash \mathtt{T} \in \bullet\,\bullet \quad \text{(WB-NONE)} \qquad \frac{\Delta \vdash \mathtt{S} <: \mathtt{T}}{\Delta \vdash \mathtt{T} \in \bullet \rhd \mathtt{S}} \quad \text{(WB-LOWER)} \qquad \frac{\Delta \vdash \mathtt{T} <: \mathtt{S} \qquad \Delta \vdash \mathtt{T} \in \bullet\,\mathtt{B}_\rhd}{\Delta \vdash \mathtt{T} \in \lhd \mathtt{S}\,\mathtt{B}_\rhd} \quad \text{(WB-UPPER)}$$

Figure 3. WFJ subtyping and bounds checking rules

relation. (WS-ENV) essentially expresses that any correct instantiation (i.e., choice of type arguments satisfying the bounds) creates a subtype. Note that there is no reflexivity rule—it emerges as a special case of (WS-ENV), using $\overline{\mathtt{X}}$ for $\overline{\mathtt{U}}$ and $\overline{\mathtt{X}} \in \overline{\mathtt{B}}$ for Δ'. Moreover, by (WS-ENV), α-equivalent existential types are also mutual subtypes, as we would expect.

The bounds check rules simply state that any type satisfies the missing bounds, that a type `T` satisfies a lower bound if that lower bound is a subtype of `T`, analogously for upper bounds, and for two bounds additionally that the lower bound must be a subtype of the upper bound.

4.3 Well-formedness

Well-formedness is required in the premises of typing rules, and the meaning of these requirements are specified in Fig. 4. In particular, type variables are well-formed if defined in the environment, the class `Object` is well-formed, and classes applied to general type arguments are well-formed if the type arguments are well-formed and the *snap* of them satisfies the bounds. Wildcard arguments are well-formed if the types mentioned in their bounds are well-formed. Well-formedness thus ensures that wildcard bounds are either absent or are tight enough to satisfy the declared bounds of the class they occur in.

4.4 Typing rules

The WFJ typing rules are shown in Fig. 5. A variable is typable with the *snap* of the syntactic type it has in the variable environment Γ.

A field lookup is typed by finding the type of the receiver ($\mathtt{e}_0$), taking the upper bound of that (to find the least upper bound which is an existential type over a class type), looking up the fields of that class, and taking *snap* of the type of the field.

Method invocation is typed by finding the class of the upper bound of the receiver type as before, using that to find a method type, capturing any $\star$ markers in the type arguments $\overline{\mathtt{P}}$, checking that the resulting actual type arguments $\overline{\mathtt{V}}$ satisfy the declared bounds for the method type arguments, that the actual arguments $\overline{\mathtt{e}}$ have types that are subtypes of the value argument types $\overline{\mathtt{U}}$, and applying *snap* to the return type. The resulting existential type must include all the type environments that `K` may depend on, i.e., Δ_0 from the receiver bound, $\overline{\Delta}$ from the actual value arguments, and Δ' from the *snap* of the method return type. The implicit α-conversion can take care of eliminating unused type variables from the resulting type environment.

4.5 Reduction rules

The WFJ reduction rules are shown in Fig. 6. We have made the evaluation order more explicit than in other papers including [16], because this is necessary in order to make wildcard capture work. In particular, all value arguments to a method invocation must be fully evaluated before (WR-INVK) can be applied, i.e., before the method can be called. Otherwise, the reduction rules are rather straightforward. For field lookup the receiver is a value and its class is available, so we can find the fields of that class and select the corresponding constructor argument. For method invocation the receiver class `N` is known, so it is easy to find the method type. The actual value arguments, $\overline{\mathtt{v}}$, are values, so their classes are available, too, in empty type environments. Wildcard capture also works in the empty environment, because $\overline{\mathtt{N}}$ does not contain type variables, and similarly for the method body. Finally, the method invocation works like β-reduction, by substituting actuals for formals (including the receiver) in the body of the method. The concruence rules are standard.

4.6 Properties

We state the subject reduction theorem. A proof is in progress, and we expect to publish it in a later version of this work. This of course is an entirely vacuous statement as far as validity of the theorem goes, but we are nevertheless optimistic.

$$\Delta \vdash \bullet\bullet \text{ ok} \qquad \frac{\Delta \vdash \mathtt{T} \text{ ok}}{\Delta \vdash \bullet \triangleright \mathtt{T} \text{ ok}} \qquad \frac{\Delta \vdash \mathtt{T} \text{ ok}}{\Delta \vdash \triangleleft \mathtt{T} \bullet \text{ ok}} \qquad \frac{\Delta \vdash \mathtt{T} \text{ ok} \quad \Delta \vdash \mathtt{S} \text{ ok} \quad \Delta \vdash \mathtt{S} <: \mathtt{T}}{\Delta \vdash \triangleleft \mathtt{T} \triangleright \mathtt{S} \text{ ok}}$$

$$\frac{\Delta \vdash \mathtt{B} \text{ ok}}{\Delta \vdash \mathtt{?B} \text{ ok}} \text{ (WF-?)} \qquad \Delta \vdash \mathtt{Object} \text{ ok} \text{ (WF-OBJECT)} \qquad \Delta \vdash \star \text{ ok} \text{ (WF-STAR)} \qquad \frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X} \text{ ok}} \text{ (WF-VAR)}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\} \qquad \Delta \vdash \overline{\mathtt{A}} \text{ ok} \qquad \mathit{snap}(\mathtt{C<}\overline{\mathtt{A}}\mathtt{>}) = \exists \Delta'.\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta\Delta' \vdash \overline{\mathtt{T}} \in [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{B}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{A}}\mathtt{>} \text{ ok}} \text{ (WF-CLASS)}$$

$$\frac{\textbf{if } \mathit{mtype}(\mathtt{m},\mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}}'\ \overline{\mathtt{B}}'\mathtt{>}\overline{\mathtt{T}}' \to \mathtt{T}' \textbf{ then } \overline{\mathtt{B}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}']\overline{\mathtt{B}}' \quad \wedge \quad \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}']\overline{\mathtt{T}}' \quad \wedge \quad \overline{\mathtt{Y}} \in \overline{\mathtt{B}} \vdash \mathtt{T} <: [\overline{\mathtt{Y}}/\overline{\mathtt{Y}}']\mathtt{T}'}{\mathit{override}(\mathtt{m},\mathtt{N},\mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T})} \text{ (WF-METHOD)}$$

Figure 4. WFJ well-formedness rules

Expression Typing:

$$\Delta;\Gamma \vdash \mathtt{x} : \mathit{snap}(\Gamma(\mathtt{x})) \text{ (WT-VAR)} \qquad \frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \qquad \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\mathtt{N}_0 \qquad \mathit{fields}(\mathtt{N}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \mathit{snap}(\mathtt{T}_i) = \exists \Delta'.\mathtt{K}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i : \exists \Delta_0\Delta'.\mathtt{K}} \text{ (WT-FIELD)}$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{P}} \text{ ok} \quad \Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{E}_0 \quad \mathit{bound}_\Delta(\mathtt{E}_0) = \exists \Delta_0.\mathtt{N}_0 \quad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \exists \overline{\Delta}.\overline{\mathtt{K}} \\ \Delta_1 = \Delta\Delta_0\overline{\Delta} \quad \mathit{mtype}(\mathtt{m},\mathtt{N}_0) = \mathtt{<}\overline{\mathtt{Y}}\overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U} \quad \overline{\mathtt{V}} = \mathit{capture}_{\Delta_1}(\overline{\mathtt{P}},\overline{\mathtt{Y}},\overline{\mathtt{U}},\overline{\mathtt{K}}) \\ \Delta_1 \vdash \overline{\mathtt{V}} \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{B}} \quad \Delta_1 \vdash \overline{\mathtt{K}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \quad \mathit{snap}([\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}) = \exists \Delta'.\mathtt{K}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{<}\overline{\mathtt{P}}\mathtt{>}\mathtt{m}(\overline{\mathtt{e}}) : \exists \Delta_0\overline{\Delta}\Delta'.\mathtt{K}} \text{ (WT-INVK)}$$

$$\frac{\Delta \vdash \mathtt{N} \text{ ok} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{E}} \qquad \Delta \vdash \overline{\mathtt{E}} <: \mathit{snap}(\overline{\mathtt{T}})}{\Delta;\Gamma \vdash \mathtt{new\ N}(\overline{\mathtt{e}}) : \exists.\mathtt{N}} \text{ (WT-NEW)}$$

Method Typing:

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{B}}',\mathtt{T},\overline{\mathtt{T}} \text{ ok} \qquad \Delta = \overline{\mathtt{Y}} \in \overline{\mathtt{B}}',\overline{\mathtt{X}} \in \overline{\mathtt{B}} \\ \Delta;\ \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 : \mathtt{E} \qquad \Delta \vdash \mathtt{E} <: \mathit{snap}(\mathtt{T}) \\ \mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\} \qquad \mathit{override}(\mathtt{m},\mathtt{N},\mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}'\mathtt{>}\overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}'\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}\ \mathtt{e}_0;\ \}\ \text{OK in } \mathtt{C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}} \text{ (WT-METHOD)}$$

Class Typing:

$$\frac{\overline{\mathtt{X}} \in \overline{\mathtt{B}} \vdash \overline{\mathtt{B}},\mathtt{N},\overline{\mathtt{T}} \text{ ok} \qquad \overline{\mathtt{M}} \text{ OK in } \mathtt{C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>}}{\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}\ \text{OK}} \text{ (WT-CLASS)}$$

Figure 5. WFJ typing rules

Computation:

$$\frac{\emptyset;\emptyset \vdash \mathtt{v} : \exists.\mathtt{N} \quad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{v}.\mathtt{f}_i \rightarrow \mathtt{v}_i}\ \text{(WR-FIELD)} \qquad \frac{\emptyset;\emptyset \vdash \mathtt{v} : \exists.\mathtt{N} \quad \emptyset;\emptyset \vdash \overline{\mathtt{v}} : \exists.\overline{\mathtt{N}} \quad \mathit{mtype}(\mathtt{m},\mathtt{N}) = \mathtt{<}\overline{\mathtt{Y}}\ \overline{\mathtt{B}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U} \quad \overline{\mathtt{V}} = \mathit{capture}_\emptyset(\overline{\mathtt{P}},\overline{\mathtt{Y}},\overline{\mathtt{U}},\overline{\mathtt{N}}) \quad \mathit{mbody}(\mathtt{<}\overline{\mathtt{V}}\mathtt{>m},\mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0}{\mathtt{v}.\mathtt{<}\overline{\mathtt{P}}\mathtt{>m}(\overline{\mathtt{v}}) \rightarrow [\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{v}/\mathtt{this}]\mathtt{e}_0}\ \text{(WR-INVK)}$$

Congruence:

$$\frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e.f} \rightarrow \mathtt{e}'.\mathtt{f}}\ \text{(WRC-FIELD)} \qquad \frac{\mathtt{e} \rightarrow \mathtt{e}'}{\mathtt{e.<P>m}(\overline{\mathtt{e}}) \rightarrow \mathtt{e}'.\mathtt{<P>m}(\overline{\mathtt{e}})}\ \text{(WRC-INV-RECV)}$$

$$\frac{\mathtt{e}_i \rightarrow \mathtt{e}_i'}{\mathtt{e.<P>m}(\ldots\mathtt{e}_i\ldots) \rightarrow \mathtt{e.<P>m}(\ldots\mathtt{e}_i'\ldots)}\ \text{(WRC-INV-ARG)} \qquad \frac{\mathtt{e}_i \rightarrow \mathtt{e}_i'}{\mathtt{new\ C}(\ldots\mathtt{e}_i\ldots) \rightarrow \mathtt{new\ C}(\ldots\mathtt{e}_i'\ldots)}\ \text{(WRC-NEW-ARG)}$$

Figure 6. WFJ reduction rules

THEOREM 1 (Subject Reduction). *If* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{E}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$ *then* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{E}'$ *for some* $\mathtt{E}'$ *such that* $\emptyset \vdash \mathtt{E}' <: \mathtt{E}$

4.6.1 Transferring results to Java

The subtyping rules in the JLS [15] have been formalized in Fig. 7. The rules for reflexivity, transitivity, variable lookup, and subclassing are straightforward. The rule (JS-SNAP) expresses that the JLS approach to performing *snap* is to do it on the left hand side of a subtyping judgement. This corresponds to doing it whenever the type of an expression evaluation must be computed, whereas the right hand side is never subject to this treatment.

The following theorem says that the JLS notion of subtyping includes WFJ subtyping. In other words, programs which are correct WFJ programs will also be correct programs in a calculus that only differs from WFJ by using JLS subtyping. A proof of this property is under construction, but is not yet complete.

THEOREM 2 (WFJ subtype implies JLS subtype). *Assume that* $\Delta \vdash \mathtt{S} <: \mathtt{T}$, *then* $\Delta \vdash \mathtt{S} <:_{JLS} \mathtt{T}$.

5. Related and Future Work

Our WFJ calculus builds upon earlier work about featherweight languages. It follows the approach and style of [16], and [17] was inspirational even though our formalization differs considerably because of the prominent role played by existential types.

The language design effort behind wildcards has been influenced by the BETA programming language and the virtual class members of that language [18, 19], via traditional declaration-site variance [3] and use-site variance [29, 17]. Scala [24] supports types as members of objects, thus enabling approaches similar to BETA, but on a more functionally oriented ground. To our knowledge, the wildcards language feature presented in [31] is novel in that it allows for a non-trivial subset of the possibilities of full-fledged existential types, but in a much more light-weight syntactic form; we find it likely that full-fledged existentials would not have been accepted into a main-stream language.

Existential types were introduced by Cardelli and Wegner in [7] and are often associated with [22] by Mitchell and Plotkin, establishing an interpretation of them as a tool for information hiding and abstraction. Such types are introduced by a '`pack`' or '`close`' operation and eliminated by an '`unpack`' or '`open`' operation. Several years later, Abadi and Cardelli existential types were used to describe the type of self in an object calculus [1, Ch.13], using syntactic variants of `pack` and `unpack`.

To our knowledge, it is a novel contribution of our calculus that it is based on a version of existential types where the quantification variables are implicitly propagated into the environment rather than being controlled by explicit `pack` and `unpack` constructs. Traditionally, this is considered unsound, but we believe that this problem is solved by ensuring that such propagated variables are distinct, and by ensuring that every expression evaluation produces its own fresh variables.

Note that the wrapping technique which is illustrated at the end of Sect. 3.1.4 makes the connection to traditional existential types even closer: Assignment of a wrapper object to a reference with a wildcarded type corresponds to `pack`, and a wildcard captured method call corresponds to `unpack`. This supports the claim that no expressive power is lost by using wildcards, compared to traditional existential types.

We still need to finish the proof of soundness, and in this sense the present paper is work in progress. However, we also believe that the formalization already without complete proofs represents a worthwhile contribution because of this novelty in the treatment of existential types, and more generally its provision of a formal basis for a now widely dis-

$$\Delta \vdash \mathtt{X} <:_{\mathrm{JLS}} \mathit{bound}^1_\Delta(\mathtt{X}) \quad \text{(JS-VAR)} \qquad \Delta \vdash \mathit{lbound}^1_\Delta(\mathtt{X}) <:_{\mathrm{JLS}} \mathtt{X} \quad \text{(JS-LVAR)} \qquad \Delta \vdash \mathtt{T} <:_{\mathrm{JLS}} \mathtt{T} \quad \text{(JS-REFL)}$$

$$\frac{\Delta \vdash \mathtt{S} <:_{\mathrm{JLS}} \mathtt{T} \quad \Delta \vdash \mathtt{T} <:_{\mathrm{JLS}} \mathtt{U}}{\Delta \vdash \mathtt{S} <:_{\mathrm{JLS}} \mathtt{U}} \quad \text{(JS-TRANS)} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X}}\ \overline{\mathtt{B}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <:_{\mathrm{JLS}} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \quad \text{(JS-SUBCLASS)}$$

$$\frac{\mathit{snap}(\mathtt{C<}\overline{\mathtt{A}}\mathtt{>}) = \exists \Delta'.\mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta\Delta' \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <:_{\mathrm{JLS}} \mathtt{T}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{A}}\mathtt{>} <:_{\mathrm{JLS}} \mathtt{T}} \quad \text{(JS-SNAP)} \qquad \frac{\Delta \vdash \overline{\mathtt{T}} \subset: \overline{\mathtt{A}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} <:_{\mathrm{JLS}} \mathtt{C<}\overline{\mathtt{A}}\mathtt{>}} \quad \text{(JS-WILD)}$$

$$\frac{}{\Delta \vdash \mathtt{T} \subset: \mathtt{T}} \quad \text{(JC-REFL)} \qquad \frac{\Delta \vdash \mathtt{T} \in \mathtt{B}}{\Delta \vdash \mathtt{T} \subset: \mathtt{?B}} \quad \text{(JC-BOUND)}$$

Figure 7. JLS subtyping rules

seminated typing feature. While we do not expect the proofs to reveal unsoundness of the mechanism itself (who does?) they may reveal errors in the formalization which will lead to subtle changes in the future.

6. Conclusion

We have presented a formalization of wildcards, including the syntax, dynamic semantics, and type system, and stated some safety properties. The paper describes how the distinguishing features of wildcards have been preserved in the calculus, and the formalization itself serves to document the nature of these properties. In particular, the usage of existential types in the calculus reflects the existential nature of wildcard capture. The proofs of safety properties have not yet been finished, but we believe that the formulation of the formalization itself is a useful result because it reveals the underlying structure of wildcards and allows for a comparison with other work.

Acknowledgements

Gilad Bracha, Neal Gafter, and Peter von der Ahé also participated in the wildcards implementation project, and they have always responded in a helpful way when we needed it. Moreover, we thank Phil Wadler for several comments on this work, in particular suggesting some significant and interesting changes which we have not yet carried out.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[2] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java programming language. In *Object Oriented Programming: Systems, Languages and Applications*, Atlanta, Georgia, October 1997. OOPSLA97, ACM Press. Toby Bloom, editor.

[3] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Object Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming*, pages 161–168, Ottawa, Canada, October 1990. OOPSLA/ECOOP90, ACM Press. Norman K. Meyrowitz, editor.

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In OOPSLA98 [27].

[5] Kim Bruce. Subtyping is not a good match for object-oriented programming languages. In *European Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 1997. ECOOP97, LNCS 1241, Springer Verlag. Mehmet Akşit and Satoshi Matsuoka, editors.

[6] Kim Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998. ECOOP98, LNCS 1445, Springer Verlag. Eric Jul, editor.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):480–521, December 1985.

[8] Robert Cartwright and Guy L. Steele. Compatible genericity with runtime-types for the Java programming language. In OOPSLA98 [27].

[9] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Principles of Programming Languages*, pages 125–135, San Francisco, California, January 1990. POPL90, ACM Press. Paul Hudak, editor.

[10] William Cook. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming*, pages 57–70, Nottingham, England, July 1989. ECOOP89, Cambridge University Press. Stephen Cook, editor.

[11] Ole Johan Dahl and Kristen Nygaard. SIMULA, an algol-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[12] ECMA. C# language specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, 2002.

[13] ECMA. C# language specification. ECMA TC39-TG2/2004/14: C# language specification, Working Draft 2.7, June 2004.

[14] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification – Third Edition*. The Java Series. Addison-Wesley, third. edition, 2004.

[16] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications*, pages 132–146, Denver, Colorado, October 1999. OOPSLA99, ACM Press. Linda Northrop, editor.

[17] Atsushi Igarashi and Mirko Viroli. On variance-based subtyping for parametric types. pages 441–469. Revised version with proofs at `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/papers/pdf/variance.full.p%df`.

[18] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programming: Systems, Languages and Applications*, New Orleans, Louisiana, October 1989. OOPSLA89, ACM Press. Norman K. Meyrowitz, editor.

[19] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[20] Bertrand Meyer. Genericity versus inheritance. In *Object Oriented Programming: Systems, Languages and Applications*, pages 391–405, Portland, Oregon, November 1986. OOPSLA86, ACM Press. Norman K. Meyrowitz, editor.

[21] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[22] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 1988.

[23] Andrew Myers, Joseph Bank, and Barbara Liskov. Parameterized types for Java. In *Conf. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 1997. POPL97, ACM Press. Neil D. Jones, editor.

[24] Martin Odersky. Report on the programming language Scala. Technical report, EPFL, Lausanne, 2002.

[25] Martin Odersky. Personal communication, Summer 2004.

[26] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 January 1997.

[27] OOPSLA98. *Object Oriented Programming: Systems, Languages and Applications*, Vancouver, BC, October 1998. ACM Press. Craig Chambers, editor.

[28] David Stoutamire and Stephen Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, August 1996.

[29] Kresten Krab Thorup and Mads Torgersen. Unifying genericity. pages 186–204.

[30] Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ah, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, December 2004. `http://www.jot.fm/issues/issue_2004_12/article5`.

[31] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proceedings of the ACM Symposium of Applied Computing*, 2004.