A Model for Java with Wildcards

Nicholas Cameron[1], Sophia Drossopoulou[1], and Erik Ernst[2⋆]

[1] Imperial College London
{ncameron, scd}@doc.ic.ac.uk
[2] University of Aarhus
eernst@daimi.au.dk

Abstract. Wildcards are a complex and subtle part of the Java type system, present since version 5.0. Although there have been various formalisations and partial type soundness results concerning wildcards, to the best of our knowledge, no system that includes all the key aspects of Java wildcards has been proven type sound. This paper establishes that Java wildcards are type sound. We describe a new formal model based on explicit existential types whose pack and unpack operations are handled implicitly, and prove it type sound. Moreover, we specify a translation from a subset of Java to our formal model, and discuss how several interesting aspects of the Java type system are handled.

1 Introduction

This paper establishes type soundness for Java wildcards, used in the Java type system to reconcile parametric polymorphism (also known as generics) and inclusion polymorphism [7] (subclassing). A *parametric* (or *generic*) type in Java, e.g., `class List<X> ...`, can be instantiated to a *parameterised type*, e.g., `List<Integer>`. Wildcards extend generics by allowing parameterised types to have actual type arguments which denote unknown or partially known types, such as `List<?>` where `?` stands for an unknown actual type argument. With traditional generics, different actual type arguments to the same parametric type create unrelated parameterised types, but wildcards introduce *variance*, i.e., they allow for subtype relationships among such types; for instance, `List<Integer>` is a subtype of `List<? extends Number>`.

Wildcards have been part of the Java language since 2004, but type soundness for Java with wildcards has been an open question until now. There are several informal, semi-formal, and formal descriptions of Java wildcards [3,11,17,24] and soundness proofs for partial systems [5,14]. However, a soundness proof for a type system exhibiting all of the interesting features of Java wildcards has been elusive. Showing type soundness for Java with wildcards is difficult for a number of reasons: first, the Java wildcard syntax prioritises a concise notation for common cases, but is not expressive enough to denote all the types which may arise during type checking; second, modeling wildcards with traditional existential types is

⋆ Supported by the DRCTPS grant 95093428.

not straightforward; and third, the inference of type parameters during a process known as wildcard capture requires careful treatment.

We show type soundness for Java with wildcards using a new formal model, TameFJ, which extends FGJ [13], which is a formalisation of Java generics. We use explicit existential types (such as ∃X.List<X>) to model Java wildcard types, but packing and unpacking of existential types is handled implicitly, in the rules for typing and subtyping. The implicit unpacking of existential types is used to model wildcard capture in Java. The use of an implicit approach (also found in Wild FJ [17]) contrasts with our recent previous work [5], where explicit packing and unpacking expressions were used. This approach created problems with expressiveness and, a soundness proof for Wild FJ is still missing, due in part to the complexity of creating existential types 'on the fly'. In this paper we provide a soundness proof without compromising expressiveness. In addition, we define a translation to TameFJ from a subset of the Java language that includes wildcards.

The main contribution of this paper is the new formal model and the soundness result, achieved via several technical innovations. Additionally, the translation supports the claim that TameFJ is a faithful model of Java with wildcards.

In Sect. 2 we outline the background for Java wildcards and their formalisation. In Sect. 3 we define and discuss TameFJ and its type soundness proof, and in Sect. 4 we present the translation from Java to TameFJ. Finally, in Sect. 5 we cover related work, and in Sect. 6 we discuss future work and conclude.

2 Background

In this section we give some background for the key concepts of this paper. We deliberately keep this section rather brief; more details may be found in [5]. In the examples here and elsewhere in the paper we make use of a class hierarchy of shapes: `Shape` is a subclass of `Object`, `Polygon` and `Circle` are subclasses of `Shape`, and `Square` is a subclass of `Polygon`.

2.1 Generics, Wildcards and Existential Types

Generics [3,11] add parametric polymorphism to Java. Classes and interfaces may be generic, i.e., they may have formal type parameters. Parameterised types are then constructed by applying generic types to actual type parameters. For example, a list class could be defined as `class List<X>...`, and we may then construct lists of strings and shapes using `List<String>` and `List<Shape>`, or more complex types like `List<List<Y>>`, where `Y` could be a type variable defined in the context. Similarly, methods may have formal type parameters and receive actual type arguments at invocation; e.g., the method `walk` has one formal type parameter, `X`, while `walkSquares` has none:

```
<X> List<X> walk(Tree<X> x) {...}

List<Square> walkSquares(Tree<Square> y) {
    return this.<Square>walk(y);
}
```

Wildcards A *wildcard type* is a parameterised type where `?` is used as an actual type parameter, for example `List<?>`. Such a type can be thought of as a list of *some* type, where the wildcard is *hiding* that type. Where multiple wildcards are used, for example `Pair<?, ?>`, each wildcard hides a potentially different type. Wildcards enjoy *variant subtyping* [14], so `List<Shape>` is a subtype of `List<?>`. This is in contrast to generic types that are *invariant* with respect to subtyping, so `List<Circle>` is not a subtype of `List<Shape>`. Wildcards may be given upper or lower bounds using the `extends` and `super` keywords, respectively. This restricts subtyping to be *co-* or *contravariant*. So `List<Square>` is a subtype of `List<? extends Polygon>` and `List<Shape>` is a subtype of `List<? super Polygon>`, but not vice versa.

Java wildcards have the property of *wildcard capture*, where a wildcard is promoted to a fresh type variable. This occurs most visibly at method calls: `List<?>` is not a subtype of `List<X>`, but the wildcard can be *capture converted* to a fresh type variable which allows otherwise illegal method invocations. Consider the following legal Java code (using the method `walk` declared above):

```
List<?> walkAny(Tree<?> y) {                              (example 1)
    this.walk(y);
}
```

At the method invocation, the wildcard in the type of `y` is capture converted to a fresh type variable, `Z`, and the method invocation can then be thought of as `this.<Z>walk(y)`. In Sect. 3.4 we show how this example is type checked in TameFJ.

Wildcard capture may give rise to types during type checking that can not be denoted using the Java syntax. This is a serious obstacle for a direct formalisation of Java wildcards using the Java syntax, because type soundness requires typability of every step of the computation, and this may require the use of types that cannot be denoted directly.

In the next example we show how the type system treats each wildcard as hiding a potentially different type. The method invocation at `1` is type incorrect because the method `compare` requires a `Pair` parameterised by a single type variable twice. `Pair<?, ?>` can not be capture converted to this type because the two wildcards may hide different types. The invocation of the `make` method at `2` has a type which is *expressible but not denotable*. The type checker knows that the wildcards hide the same type (even though this can not be denoted in the surface syntax) and so capture conversion, and thus type checking, succeeds.

```
<X>Pair<X, X> make(List<X> x) {}                          (example 2)
<X>Boolean compare(Pair<X, X> x) {}

void m()
{
    Pair<?, ?> p;
    List<?> b;

    this.compare(p);             //1, type incorrect
    this.compare(this.make(b));  //2, OK
}
```

Again, we show how this example is type checked in Sect. 3.4. The example can be easily understood (and type checked) by using existential types to denote the types that are expressible but not denotable using Java syntax.

Existential types Existential types are a form of parametric polymorphism whereby a type may be existentially quantified [6,7,10,20,21,22]. For example, a function may be defined with type $\exists$`T.T` $\rightarrow$ `T`, that is, the function has type `T` to `T` for some type, `T`. Existential types are a basis for data hiding and abstraction in type systems; an early practical use was in modelling abstract data types [20]. In our formalisation we use existential types in a Java like setting, and so are concerned with types of the form $\exists$`X.List<X>`. Values of existential types are opaque packages; usually they are created using a *pack* (or *close*) expression and then have to be *unpack*ed (or *open*ed) using another expression before they can be used; in our approach both packing and unpacking occur implicitly.

2.2 Formalising Wildcards

The correspondence between wildcards and existential types goes back to the work on Variant Parametric Types [14]. It has been integral to all formal work with wildcards since [24]. This correspondence is discussed in more depth in Sect. 4.

Wildcards are a strict extension of Java generics, and far more interesting to describe formally. A number of features contribute to this, but foremost among them is wildcard capture. Wildcard capture is roughly equivalent to unpacking an existential type [17,24], but an explicit unpack expression appears to be very hard to use to safely model wildcard capture [5]. Wildcards may have lower bounds, which also introduces problems. Indeed, they had to be omitted from our previous work [5] in order to show type soundness. Lower bounds can cause problems by transitivity of subtyping; a naïve formalism would consider a type variable's lower bound to be a subtype of its upper bound, even if there is no such relationship in the class hierarchy. This issue is addressed in Sect. 3.3. Furthermore, when an existential type is created (which occurs in subtyping in Java and TameFJ) we must somehow keep track of the witness type—the type hidden by the wildcard—in order to recover it when the type is unpacked. In [5]

this is done in the syntax of close expressions. However, in a system without explicit packing of existential types it has proven very difficult to track the witness types. Therefore, we resort to following the Java compiler (and [17]) and infer the hidden type parameters during execution and type checking. This reliance on a simple kind of type inference can cause problems for the proof of subject reduction, as described in [17], and it is one of the contributions of this paper to handle it safely.

Wild FJ [17] is the first, and previously only, formalism that includes all the interesting features of Java wildcards. Our formal model is, in many ways, a development of Wild FJ. The syntax of Wild FJ is a strict subset of Java with wildcards, requiring explicit type arguments to polymorphic method calls as in our approach. However, Java types are converted to existential types 'on the fly', and this conversion of types makes the typing, subtyping, well-formedness, and auxiliary rules more complicated in Wild FJ. As a rough metric there are 10 auxiliary functions with 23 cases, nine subtyping, and 10 well-formedness rules in Wild FJ, compared with seven auxiliary functions with 15 cases, eight subtyping (11, counting subclassing), and eight well-formedness rules in our system. Type soundness has never been proven for Wild FJ.

3 Type Soundness for Java Wildcards

We show type soundness for Java by developing a core calculus, TameFJ, which models all the significant elements of type checking found in Java with wildcards. TameFJ is not a strict subset of the Java language. However, a Java program written in a subset of Java (corresponding to the syntax of Wild FJ) can be easily translated to a TameFJ program, as we discuss in Sect. 4. Part of that translation is to perform Java's inference of type parameters for method calls (except where this involves wildcards). As is common [17], we regard this as a separate pre-processing step and do not model this in TameFJ.

TameFJ is an extension of FGJ [13]. The major extension to FGJ is the addition of existential types, used to model wildcard types. Typing, subtyping and reduction rules must be extended to accommodate these new types, and to handle wildcard capture.

We use existential types in the surface syntax and, in contrast to Wild FJ, do not create them during type checking; this simplifies the formal system and our proofs significantly. In particular, capture conversion is dealt with more easily in our system because fresh type variables do not have to be supplied. We also 'pack' existential types more declaratively, by using subtyping, rather than explicitly constructing existential types. This means that we avoid obtaining the awkward[1] type $\exists \mathtt{X}.\mathtt{X}$, found both in [17] and our previous work[2] [5].

[1] There is no corresponding type in Java, so it is unclear how such a type should behave.

[2] Such a type is required in earlier work because the construction $\exists \Delta.\mathtt{T}$ appears in the conclusion of type rules, where $\mathtt{T}$ is a previously derived type. Since $\mathtt{T}$ may

e	::=	x | e.f | e.<$\overline{\texttt{P}}$>m($\overline{\texttt{e}}$) | new C<$\overline{\texttt{T}}$>($\overline{\texttt{e}}$)	*expressions*
Q	::=	class C<$\overline{\texttt{X}\lhd\ \texttt{T}}$> $\lhd$ N {$\overline{\texttt{T f;}}$ $\overline{\texttt{M}}$}	*class declarations*
M	::=	<$\overline{\texttt{X}\lhd\ \texttt{T}}$> T m($\overline{\texttt{T x}}$) {return e;}	*method declarations*
v	::=	new C<$\overline{\texttt{T}}$>($\overline{\texttt{v}}$)	*values*
N	::=	C<$\overline{\texttt{T}}$> | Object<>	*class types*
R	::=	N | X	*non-existential types*
T, U	::=	$\exists\Delta$.N | $\exists\emptyset$.X	*types*
P	::=	T | $\star$	*type parameters*
Δ	::=	$\overline{\texttt{X}\rightarrow[\texttt{B}_l\ \texttt{B}_u]}$	*type environments*
Γ	::=	$\overline{\texttt{x:T}}$	*variable environments*
B	::=	T | $\bot$	*bounds*
x			*variables*
C			*classes*
X, Y			*type variables*

Fig. 1. Syntax of TameFJ.

TameFJ has none of the limitations of our previous approach [5]; we allow lower bounds, have more flexible type environments, allow quantification of more than one type variable in an existential type, and have more flexible subtyping. Thus, together with the absence of open and close expressions, TameFJ is much closer to the Java programming language.

3.1 Notation and Syntax

TameFJ is a calculus in the FJ [13] style. We use vector notation for sequences; for example, $\overline{\texttt{x}}$ stands for a sequence of '`x`'s. We use $\emptyset$ to denote the empty sequence. We use a comma to concatenate two sequences. We implicitly assume that concatenation of two sequences of mappings only succeeds if their domains are disjoint. We use $\lhd$ as a shorthand for `extends` and $\rhd$ for `super`. The function $fv()$ returns the free variables of a type or expression, and $dom()$ returns the domain of a mapping. We assume that all type variables, variables, and fields are named uniquely.

The syntax for TameFJ is given in Fig. 1. The syntax for expressions and class and method declarations is very similar to Java, except that we allow $\star$ as a type parameter in method invocations. In TameFJ (and as opposed to Java) all

be a type variable, one may construct $\exists$X.X; this can not happen in our calculus. Under a standard interpretation of existential types, types of the form $\exists\texttt{X}\lhd$ T.X have no observably different behaviour from T because Java subtyping already involves subclass polymorphism. Rigorous justification of this fact is outside the scope of this paper, but is part of planned future work.

actual type parameters to a method invocation must be given. However, where a type parameter is existentially quantified (corresponding to a wildcard in Java), we may use $\star$ to mark that the parameter should be inferred. Such types can not be named explicitly because they can not be named outside of the scope of their type. The marker $\star$ is not a replacement for `?` in Java; $\star$ can not be used as a parameter in TameFJ types, and `?` can not be used as a type parameter to method calls in Java. Note that we treat `this` as a regular variable.

The syntax of types is that of FGJ [13] extended with existential types. Non-existential types consist of class types (e.g., `C<D<>>`) and type variables, `X`. Types (`T`) are existential types, that is a non-existential type (`R`) quantified by an environment (Δ, i.e., a sequence of formal type variables and their bounds), for example, $\exists$`X` $\rightarrow$ `[`$\exists\emptyset$`.D<>` $\exists\emptyset$`.Object<>].C<X>`. Type variables may only be quantified by the empty environment, e.g., $\exists\emptyset$`.X`. In the text and examples, we use the shorthands `C` for `C<>`, $\exists$`X.C<X>` for $\exists$`X`$\rightarrow$`[`$\bot$ `Object<>].C<X>`, and `R` for $\exists\emptyset$`.R`.

Existential types in TameFJ correspond to types parameterised by wildcards in Java. Using `T` as an upper or lower bound on a formal type variable corresponds to using `extends T` or `super T`, respectively, to bound a wildcard. This correspondence is discussed further in Sect. 4. The bottom type, $\bot$, is used only as a lower bound and is used to model the situation in Java where a lower bound is omitted.

Substitution in TameFJ is defined in the usual way with a slight modification. For the sake of consistency formal type variables are quantified by the empty set when used as a type in a program ($\exists\emptyset$`.X`). Therefore, we define substitution on such types to replace the whole type, which is `[T/X]`$\exists\emptyset$`.X` $=$ `T`.

A variable environment, Γ, maps variables to types. A type environment, Δ, maps type variables to their bounds. Where the distinction is clear from the context, we use "environment" to refer to either sort of environment.

3.2 Subtyping

The subclassing relation between non-existential types ($\sqsubset\!:$), reflects the class hierarchy. Subclassing of type variables is restricted to reflexivity because they have no place in the subclass hierarchy. Subtyping ($<\!:$) extends subclassing by adding subtyping between existential types and between type variables and their bounds. Extended subclassing ($\sqsubset\!:$) is an intermediate relation that expresses the class hierarchy (with the addition of a bottom type) and the behaviour of wildcards and type variables *as type parameters*; it is used mainly to simplify the proofs of soundness. All three relations are defined in Fig. 2.

The rule XS-Env, adapted from Wild FJ [17], gives all the interesting variance properties for wildcard types. It gives a subtype relationship between two existentially quantified class types, where the type parameters of the subtype are 'more precise' than those of the supertype. The following relationships are given by this rule, given the class hierarchy described in Sect. 2 and using the shorthands described in Sect. 3.1:

Subclasses: $\boxed{\vdash \mathtt{R} \sqsubset\!\!: \mathtt{R}}$

$$\frac{\mathtt{class\ C}\langle\overline{\mathtt{X}\lhd \mathtt{T}_u}\rangle \lhd \mathtt{N}\ \{\ldots\}}{\vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle \sqsubset\!\!: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N}}\ (\text{SC-Sub-Class}) \qquad \frac{}{\vdash \mathtt{R} \sqsubset\!\!: \mathtt{R}}\ (\text{SC-Reflex}) \qquad \frac{\vdash \mathtt{R} \sqsubset\!\!: \mathtt{R}'' \qquad \vdash \mathtt{R}'' \sqsubset\!\!: \mathtt{R}'}{\vdash \mathtt{R} \sqsubset\!\!: \mathtt{R}'}\ (\text{SC-Trans})$$

Extended subclasses: $\boxed{\Delta \vdash \mathtt{B} \sqsubset\!\!: \mathtt{B}}$

$$\frac{\mathtt{class\ C}\langle\overline{\mathtt{X}\lhd \mathtt{T}_u}\rangle \lhd \mathtt{N}\ \{\ldots\}}{\Delta \vdash \exists\Delta'.\mathtt{C}\langle\overline{\mathtt{T}}\rangle \sqsubset\!\!: \exists\Delta'.[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N}}\ (\text{XS-Sub-Class}) \qquad \frac{}{\Delta \vdash \bot \sqsubset\!\!: \mathtt{B}}\ (\text{XS-Bottom}) \qquad \frac{}{\Delta \vdash \mathtt{B} \sqsubset\!\!: \mathtt{B}}\ (\text{XS-Reflex})$$

$$\frac{\Delta \vdash \mathtt{B} \sqsubset\!\!: \mathtt{B}'' \quad \Delta \vdash \mathtt{B}'' \sqsubset\!\!: \mathtt{B}'}{\Delta \vdash \mathtt{B} \sqsubset\!\!: \mathtt{B}'}\ (\text{XS-Trans}) \qquad \frac{dom(\Delta') \cap fv(\exists\overline{\mathtt{X}\to[\mathtt{B}_l\ \mathtt{B}_u]}.\mathtt{N}) = \emptyset \qquad fv(\overline{\mathtt{T}}) \subseteq dom(\Delta, \Delta') \quad \Delta, \Delta' \vdash \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{B}_l <\!\!: \mathtt{T}} \qquad \Delta, \Delta' \vdash \overline{\mathtt{T} <\!\!: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{B}_u}}{\Delta \vdash \exists\Delta'.[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{N} \sqsubset\!\!: \exists\overline{\mathtt{X}\to[\mathtt{B}_l\ \mathtt{B}_u]}.\mathtt{N}}\ (\text{XS-Env})$$

Subtypes: $\boxed{\Delta \vdash \mathtt{B} <\!\!: \mathtt{B}}$

$$\frac{\Delta \vdash \mathtt{B} \sqsubset\!\!: \mathtt{B}'}{\Delta \vdash \mathtt{B} <\!\!: \mathtt{B}'}\ (\text{S-SC}) \qquad \frac{\Delta \vdash \mathtt{B} <\!\!: \mathtt{B}'' \qquad \Delta \vdash \mathtt{B}'' <\!\!: \mathtt{B}'}{\Delta \vdash \mathtt{B} <\!\!: \mathtt{B}'}\ (\text{S-Trans}) \qquad \frac{\Delta(\mathtt{X}) = [\mathtt{B}_l\ \mathtt{B}_u]}{\begin{array}{l}\Delta \vdash \exists\emptyset.\mathtt{X} <\!\!: \mathtt{B}_u \\ \Delta \vdash \mathtt{B}_l <\!\!: \exists\emptyset.\mathtt{X}\end{array}}\ (\text{S-Bound})$$

Fig. 2. TameFJ subclasses, extended subclasses, and subtypes.

$\emptyset \vdash$ `Shape` $\sqsubset\!\!:$ `Shape`
$\emptyset \vdash$ `List<Shape>` $\sqsubset\!\!:$ $\exists$`X.List<X>`
$\emptyset \vdash$ `List<Shape>` $\sqsubset\!\!:$ $\exists$`X`$\to$`[Circle Object].List<X>`
$\emptyset \vdash$ $\exists$`X`$\to$`[Circle Shape].List<X>` $\sqsubset\!\!:$ $\exists$`X`$\to$`[Circle Object].List<X>`
$\emptyset \vdash$ $\exists$`X.Pair<X, X>` $\sqsubset\!\!:$ $\exists$`Y,Z.Pair<Y, Z>`

That type parameters are 'more precise' is expressed in terms of a substitution, $[\overline{\mathtt{T}/\mathtt{X}}]$, where $\overline{\mathtt{X}}$ are some of the parameters of the supertype and $\overline{\mathtt{T}}$ are the corresponding parameters in the subtype. The subtype checks in the premises of XS-Env ensure that $\overline{\mathtt{T}}$ are 'more precise' than $\overline{\mathtt{X}}$; that is, that $\overline{\mathtt{T}}$ are within the bounds of $\overline{\mathtt{X}}$. The first premise ensures that free variables in the supertype can not be captured in the subtype, thus forbidding erroneous subtypes such as $\Delta \vdash \exists\mathtt{X.C<X>} \sqsubset\!\!: \mathtt{C<X>}$. The second premise ensures that variables are not introduced to the subtype which are not bound either in Δ or Δ'. This is a limited form of well-formedness constraint on the subtype, and is only used in the details of the proof of soundness.

Most of the type rules and lemmas are expressed in terms of subtyping, however, the standard object-oriented features of the language (such as field and method lookup) are defined around subclassing. We therefore need lemmas that link subtyping with subclassing. This is done in two stages: lemma 17 links

subtyping to extended subclassing, and lemma 35 links extended subclassing to subclassing.

Lemma 17 (*uBound* refines subtyping) *If* $\Delta \vdash \mathtt{T} <: \mathtt{T}'$ *and* $\vdash \Delta$ OK *then* $\Delta \vdash uBound_\Delta(\mathtt{T}) \sqsubset: uBound_\Delta(\mathtt{T}')$.

This lemma states that if two types are subtypes then their upper bounds are extended subclasses. The *uBound* function (defined in Fig. 7) returns a non-variable type by recursively finding the upper bound of a type until a non-variable type is reached. The interesting cases in the proof are from the S-BOUND rule; where $\mathtt{T} = \exists\emptyset.\mathtt{X}$ and $\mathtt{T}' = \mathtt{B}_u$, then by the definition of *uBound*, we have that $uBound(\exists\emptyset.\mathtt{X}) = uBound(\mathtt{B}_u)$, and are done by reflexivity. The other S-BOUND sub-case is where $\mathtt{T} = \mathtt{B}_l$ and $\mathtt{T}' = \exists\emptyset.\mathtt{X}$, here we use $\Delta \vdash uBound(\mathtt{B}_l) \sqsubset: uBound(\mathtt{B}_u)$ from F-ENV and $uBound(\exists\emptyset.\mathtt{X}) = uBound(\mathtt{B}_u)$, again from the definition of *uBound*. A corollary to this lemma is that any two non-variable types, which are subtypes, are also subclasses.

Lemma 35 (Extended subclassing gives subclassing) *If* $\Delta \vdash \exists\Delta'.\mathtt{R}' \sqsubset: \exists\overline{\mathtt{X}\rightarrow\mathtt{[B}_l\ \mathtt{B}_u\mathtt{]}}.\mathtt{R}$ *and* $\Delta \vdash$ OK *then there exists* $\overline{\mathtt{T}}$ *where* $\vdash \mathtt{R}' \sqsubset: \mathtt{[}\overline{\mathtt{T/X}}\mathtt{]R}$ *and* $\Delta, \Delta' \vdash \mathtt{T} <: \mathtt{[}\overline{\mathtt{T/X}}\mathtt{]B}_u$ *and* $\Delta, \Delta' \vdash \mathtt{[}\overline{\mathtt{T/X}}\mathtt{]B}_l <: \mathtt{T}$ *and* $fv(\overline{\mathtt{T}}) \subseteq dom(\Delta, \Delta')$.

This lemma states that for any types in an extended subclass relationship, a substitution can be found where there is a subclass relationship between the subtype and the substituted supertype. The difference between subclassing and extended subclassing is, essentially, the XS-ENV rule. This rule finds an extended subclass of an existential type by substituting away its existential type variables. This substitution corresponds to the one in the conclusion of the lemma.

3.3 Well-formedness

Rules for judging well-formed types and type environments are given in Fig. 3. The rules for well-formed type environments are the most interesting. There are two motivating issues: we must not allow type variables which have upper and lower bounds that are unrelated in the class hierarchy; and we must restrict forward references.

The first issue can cause a problem where an environment could judge a subtype relation that does not reflect the class hierarchy. For example, an environment containing `Z→[Fish Plant]` could judge (by using rule S-Bound and transitivity) that `Fish` is a subtype of `Plant`, which is presumably incorrect. We therefore check that the bounds of a type variable are related by subtyping under an environment without that type variable. We also require the stronger subclass relationship to hold for the upper bounds of the type variable's immediate bounds. This ensures that subtype relationships judged by a well-formed environment respect the class hierarchy. We need this property to prove lemma 17, described in Sect. 3.2.

Forward references are only allowed to occur as *parameters* of the bounding type. In the well-formedness rule, this is addressed by allowing forward references

Well-formed types: $\boxed{\Delta \vdash \mathtt{B}\ \text{OK},\ \Delta \vdash \mathtt{P}\ \text{OK},\ \Delta \vdash \mathtt{R}\ \text{OK}}$

$$\frac{\mathtt{X} \in \Delta}{\Delta \vdash \mathtt{X}\ \text{OK}}\ \text{(F-Var)} \qquad \frac{}{\Delta \vdash \bot\ \text{OK}}\ \text{(F-Bottom)} \qquad \frac{}{\Delta \vdash \mathtt{Object<>}\ \text{OK}}\ \text{(F-Object)}$$

$$\frac{}{\Delta \vdash \star\ \text{OK}}\ \text{(F-Star)} \qquad \frac{\mathtt{class\ C<}\overline{\mathtt{X} \lhd \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \quad \Delta \vdash \overline{\mathtt{T}}\ \text{OK} \quad \Delta \vdash \mathtt{T} <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{T}_u}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{OK}}\ \text{(F-Class)} \qquad \frac{\Delta \vdash \Delta'\ \text{OK} \quad \Delta, \Delta' \vdash \mathtt{R}\ \text{OK}}{\Delta \vdash \exists \Delta'.\mathtt{R}\ \text{OK}}\ \text{(F-Exist)}$$

Well-formed type environments: $\boxed{\Delta \vdash \Delta\ \text{OK}}$

$$\frac{}{\Delta \vdash \emptyset\ \text{OK}}\ \text{(F-Env-Empty)} \qquad \frac{\Delta, \mathtt{X} \rightarrow [\mathtt{B}_l\ \mathtt{B}_u], \Delta' \vdash \mathtt{B}_l\ \text{OK} \quad \Delta, \mathtt{X} \rightarrow [\mathtt{B}_l\ \mathtt{B}_u], \Delta' \vdash \mathtt{B}_u\ \text{OK} \quad \Delta \vdash uBound_\Delta(\mathtt{B}_l) \sqsubset: uBound_\Delta(\mathtt{B}_u) \quad \Delta \vdash \mathtt{B}_l <: \mathtt{B}_u \quad \Delta, \mathtt{X} \rightarrow [\mathtt{B}_l\ \mathtt{B}_u] \vdash \Delta'\ \text{OK}}{\Delta \vdash \mathtt{X} \rightarrow [\mathtt{B}_l\ \mathtt{B}_u], \Delta'\ \text{OK}}\ \text{(F-Env)}$$

Fig. 3. TameFJ well-formed types and type environments.

Method typing: $\boxed{\Delta \vdash \mathtt{M}\ \text{OK in}\ \mathtt{C}}$

$$\frac{\Delta' = \overline{\mathtt{Y} \rightarrow [\bot\ \ \mathtt{T}_u]} \quad \Delta \vdash \Delta'\ \text{OK} \quad \Delta, \Delta' \vdash \mathtt{T}, \overline{\mathtt{T}}\ \text{OK} \quad \mathtt{class\ C<}\overline{\mathtt{X}\ldots}\mathtt{>} \lhd \mathtt{N}\ \{\ldots\} \quad \Delta, \Delta'; \overline{\mathtt{x}:\mathtt{T}},\ \mathtt{this}:\exists\emptyset.\mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e} : \mathtt{T} \mid \emptyset \quad override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y} \lhd \mathtt{T}_u}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T})}{\Delta \vdash \mathtt{<}\overline{\mathtt{Y} \lhd \mathtt{T}_u}\mathtt{>T\ m(}\overline{\mathtt{T}\ \mathtt{x}}\mathtt{)\ \{return\ e\}}\ \text{OK in}\ \mathtt{C}}\ \text{(T-Method)}$$

$$\frac{mType(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{U}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T}}{override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{U}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T})}\ \text{(T-Override)} \qquad \frac{mType(\mathtt{m}, \mathtt{N}) \quad undefined}{override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{U}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T})}\ \text{(T-OverrideUndef)}$$

Class typing: $\boxed{\vdash \mathtt{Q}\ \text{OK}}$

$$\frac{\Delta = \overline{\mathtt{X} \rightarrow [\bot\ \ \mathtt{T}_u]} \quad \emptyset \vdash \Delta\ \text{OK} \quad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \text{OK} \quad \Delta \vdash \overline{\mathtt{M}}\ \text{OK in}\ \mathtt{C}}{\vdash \mathtt{class\ C<}\overline{\mathtt{X} \lhd \mathtt{T}_u}\mathtt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\ \mathtt{f};}\ \overline{\mathtt{M}}\}\ \text{OK}}\ \text{(T-Class)}$$

Fig. 4. TameFJ class and method typing rules.

when checking that the bounds are well-formed types, but not when checking the subtype and subclass relationships of the bounds. This reflects Java where (in a

class or method declaration) `<X◁ Y, Y◁ Object>` is illegal, due to the forward reference in the bound of `X`; however, `<X◁ List<Y>, Y◁ Object>` is legal.

3.4 Typing

Method and class type checking judgements are given in Fig. 4 and are mostly straightforward. The only interesting detail is the correct construction of type environments for checking well-formedness of types and type environments. The *override* relation allows method overriding, but does not allow overloading.

Expression typing: $\boxed{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta}$

$$\frac{}{\Delta; \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \mid \emptyset} \text{(T-Var)}$$

$$\frac{\Delta \vdash \mathtt{C\langle\overline{T}\rangle}\ \text{OK} \quad fields(\mathtt{C}) = \overline{\mathtt{f}} \quad \overline{fType(\mathtt{f}, \mathtt{C\langle\overline{T}\rangle}) = \mathtt{U}} \quad \Delta; \Gamma \vdash \mathtt{e} : \mathtt{U} \mid \overline{\emptyset}}{\Delta; \Gamma \vdash \mathtt{new\ C\langle\overline{T}\rangle(\overline{e})} : \exists\emptyset.\mathtt{C\langle\overline{T}\rangle} \mid \emptyset} \text{(T-New)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} : \exists\Delta'.\mathtt{N} \mid \emptyset \quad fType(\mathtt{f}, \mathtt{N}) = \mathtt{T}}{\Delta; \Gamma \vdash \mathtt{e.f} : \mathtt{T} \mid \Delta'} \text{(T-Field)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{U} \mid \Delta' \quad \Delta, \Delta' \vdash \mathtt{U} <: \mathtt{T} \quad \Delta \vdash \Delta'\ \text{OK} \quad \Delta \vdash \mathtt{T}\ \text{OK}}{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \emptyset} \text{(T-Subs)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} : \exists\Delta'.\mathtt{N} \mid \emptyset \quad mType(\mathtt{m}, \mathtt{N}) = \mathtt{\langle\overline{Y\lhd B}\rangle\overline{U} \to U} \quad \Delta \vdash \overline{\mathtt{P}}\ \text{OK} \quad \Delta; \Gamma \vdash \overline{\mathtt{e} : \exists\Delta.\mathtt{R} \mid \emptyset} \quad match(sift(\overline{\mathtt{R}}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}), \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{T}}) \quad \overline{\Delta, \Delta', \overline{\Delta} \vdash \mathtt{T} <: \mathtt{[\overline{T}/\overline{Y}]B}} \quad \Delta, \Delta', \overline{\Delta} \vdash \exists\emptyset.\mathtt{R} <: \mathtt{[\overline{T}/\overline{Y}]U}}{\Delta; \Gamma \vdash \mathtt{e.\langle\overline{P}\rangle m(\overline{e})} : \mathtt{[\overline{T}/\overline{Y}]U} \mid \Delta', \overline{\Delta}} \text{(T-Invk)}$$

Fig. 5. TameFJ expression typing rules.

The typing rules are given in Fig. 5. Auxiliary functions used in typing are given in Figs. 6 and 7.

The type checking judgement has the form $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$, and should be read as

> expression `e` has type `T` under the environments Δ and Γ, guarded by environment Δ'.

Δ' contains variables that have been unpacked from an existential type during type checking. These variables are used with Δ to judge some premises of a rule. Any free variables in `T` are bound in either Δ or Δ'.

T-SUBS is an extended subsumption rule; when Δ' is empty it allows an expression to be typed with a supertype of the expression's type in the usual way. The T-SUBS rule can also be used to 'remove' the guarding environment from the judgement. Type checking of a TameFJ expression is complete when a type is found using an empty guarding environment (non-empty guarding environments may only occur at intermediate stages in the derivation tree). This ensures that no bound type variables escape the scope in which they are unpacked. The scope covers the conclusions, some premises, and the derivations of these premises in the type rule in which the variables are unbound.

Auxiliary Functions: $\boxed{uBound_\Delta(\mathtt{B}) \text{ and } match(\overline{\mathtt{R}}, \overline{\mathtt{U}}, \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{T}}) \text{ and } sift(\overline{\mathtt{R}}, \overline{\mathtt{U}}, \overline{\mathtt{Y}})}$

$$uBound_\Delta(\mathtt{B}) = \begin{cases} uBound_\Delta(\mathtt{B}_u), & if\ \mathtt{B} = \exists\emptyset.\mathtt{X},\ \ where\ \Delta(\mathtt{X}) = [\mathtt{B}_l\ \mathtt{B}_u] \\ \mathtt{B}, & otherwise \end{cases}$$

$$\frac{\begin{array}{c}\forall j\ where\ \mathtt{P}_j = \star : \mathtt{Y}_j \in fv(\overline{\mathtt{R}'}) \qquad \forall i\ where\ \mathtt{P}_i \neq \star : \mathtt{T}_i = \mathtt{P}_i \\ \vdash \mathtt{R} \sqsubset: \overline{[\mathtt{T}/\mathtt{Y}, \mathtt{T}'/\mathtt{X}]\mathtt{R}'} \\ dom(\overline{\Delta}) = \overline{\mathtt{X}} \qquad fv(\overline{\mathtt{T}}, \overline{\mathtt{T}'}) \cap \overline{\mathtt{Y}}, \overline{\mathtt{X}} = \emptyset\end{array}}{match(\overline{\mathtt{R}}, \overline{\exists\Delta.\mathtt{R}'}, \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{T}})}$$

$$\frac{\mathtt{X} \in \overline{\mathtt{Y}}}{sift((\mathtt{R}, \overline{\mathtt{R}}),\ (\exists\emptyset.\mathtt{X}, \overline{\mathtt{U}}),\ \overline{\mathtt{Y}}) = sift(\overline{\mathtt{R}},\ \overline{\mathtt{U}},\ \overline{\mathtt{Y}})}$$

$$\frac{\mathtt{X} \notin \overline{\mathtt{Y}} \qquad sift(\overline{\mathtt{R}},\ \overline{\mathtt{U}},\ \overline{\mathtt{Y}}) = (\overline{\mathtt{R}'},\ \overline{\mathtt{U}'})}{sift((\mathtt{R}, \overline{\mathtt{R}}),\ (\exists\emptyset.\mathtt{X}, \overline{\mathtt{U}}),\ \overline{\mathtt{Y}}) = ((\mathtt{R}, \overline{\mathtt{R}'}),\ (\exists\emptyset.\mathtt{X}, \overline{\mathtt{U}'}))}$$

$$\frac{}{sift(\emptyset,\ \emptyset,\ \overline{\mathtt{Y}}) = (\emptyset,\ \emptyset)} \qquad \frac{sift(\overline{\mathtt{R}},\ \overline{\mathtt{U}},\ \overline{\mathtt{Y}}) = (\overline{\mathtt{R}'},\ \overline{\mathtt{U}'})}{sift((\mathtt{R}, \overline{\mathtt{R}}),\ (\exists\Delta.\mathtt{N}, \overline{\mathtt{U}}),\ \overline{\mathtt{Y}}) = (\mathtt{R}, \overline{\mathtt{R}'},\ \exists\Delta.\mathtt{N}, \overline{\mathtt{U}'})}$$

Fig. 6. Auxiliary functions for TameFJ.

Typing of variables and '`new`' expressions is done in the usual way. The lookup function $fields$ returns a sequence of the field names in a class, and $fType$ takes a field and a class type and returns the field's type.

The type checking of field access and method invocation expressions follow similar patterns: sub-expressions are type checked and their types are unpacked, then some work is done using these unpacked types, and a result type is found. The rule T-SUBS may then be used to find a final result type that does not require a guarding environment.

In the following paragraphs we describe unpacking and packing, followed by descriptions of type checking using T-FIELD and T-INVK, accompanied with examples.

Lookup Functions

$$\frac{}{fields(\texttt{Object}) = \emptyset} \qquad \frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{D<...>}\ \{\overline{\texttt{U f;}}\ \overline{\texttt{M}}\} \quad fields(\texttt{D}) = \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{g}}, \overline{\texttt{f}}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{U f;}}\ \overline{\texttt{M}}\} \qquad \texttt{f} \notin \overline{\texttt{f}}}{fType(\texttt{f}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = fType(\texttt{f}, [\overline{\texttt{T}/\texttt{X}}]\texttt{N})} \qquad \frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{U f;}}\ \overline{\texttt{M}}\}}{fType(\texttt{f}_i, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}/\texttt{X}}]\texttt{U}_i}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{U f;}}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mBody(\texttt{m}, [\overline{\texttt{T}/\texttt{X}}]\texttt{N})} \qquad \frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{U' f;}}\ \overline{\texttt{M}}\} \quad \texttt{<}\overline{\texttt{Y}\lhd\ \texttt{T}'_u}\texttt{>}\ \texttt{U m}\,(\overline{\texttt{U x}})\ \{\texttt{return e}_0\texttt{;}\} \in \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}; [\overline{\texttt{T}/\texttt{X}}]\texttt{e}_0)}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{U f;}}\ \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = mType(\texttt{m}, [\overline{\texttt{T}/\texttt{X}}]\texttt{N})} \qquad \frac{\texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}_u}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{U' f;}}\ \overline{\texttt{M}}\} \quad \texttt{<}\overline{\texttt{Y}\lhd\ \texttt{T}'_u}\texttt{>}\ \texttt{U m}\,(\overline{\texttt{U x}})\ \{\texttt{return e}_0\texttt{;}\} \in \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}/\texttt{X}}](\texttt{<}\overline{\texttt{Y}\lhd\ \texttt{T}'_u}\texttt{>}\overline{\texttt{U}} \to \texttt{U})}$$

Fig. 7. Method and field lookup functions for TameFJ.

Unpacking an existential type ($\exists\Delta.\texttt{R}$) entails separating the environment (Δ) from the quantified type (R). Δ can be used to judge premises of a rule and must be added to the guarding environment in the rule's conclusion. R can be used without quantification in the rule; bound type variables in R will now be free, we must take care that these do not escape the scope of the type rule.

If the result of type checking an expression contains escaping type variables (indicated by a non-empty guarding environment), then we must find a supertype (using T-Subs) in which there are no free variables, and use this as the expression's type. In the case that an escaping type variable occurs as a type parameter (e.g., X in C<X>), then the type may be packed to an existential type (e.g., $\exists$X.C<X>) using the subtyping rule XS-Env. In the case that the type variable is the whole type, i.e., $\exists\emptyset$.X, then the upper bound of X can be used as the result type by using S-Bound.

Field access In T-Field, the $fType$ function applied to the unpacked type (N) of the receiver gives the type of the field (T). Because T may contain type variables bound in the environment Δ', the judgement must be guarded by Δ'.

Example — Field access The following example of the derivation of a type for a field access expression demonstrates the sequence of unpacking, finding the field type, and finding a supertype that does not contain free variables. In the example, the type labelled 1 is unpacked to 2. The type labelled 3 would escape its scope, and so its supertype (4) must be used as the result of type checking. We assume that the TreeNode<Y> class declaration has a field datum with type Y and that $\Gamma =$ x:$\exists$X$\to$[$\bot$ Shape].TreeNode<X>.

$$\dfrac{\dfrac{\dfrac{\emptyset;\Gamma \vdash \texttt{x} : \exists \texttt{X}{\rightarrow}\texttt{[}\bot\ \texttt{Shape]}\texttt{.TreeNode<X>}^1 \mid \emptyset \quad fType(\texttt{datum}, \texttt{TreeNode<X>}^2) = \texttt{X}^3}{\emptyset;\Gamma \vdash \texttt{x.datum} : \texttt{X}^3 \mid \texttt{X}{\rightarrow}\texttt{[}\bot\ \texttt{Shape]}^2}\ \text{(T-Field)} \quad \emptyset, \texttt{X}{\rightarrow}\texttt{[}\bot\ \texttt{Shape]} \vdash \texttt{X}^3 <: \texttt{Shape}^4 \quad \emptyset \vdash \texttt{X}{\rightarrow}\texttt{[}\bot\ \texttt{Shape]}\ \text{OK} \quad \emptyset \vdash \texttt{Shape}^4\ \text{OK}}{\emptyset;\Gamma \vdash \texttt{x.datum} : \texttt{Shape}^4 \mid \emptyset}\ \text{(T-Subs)}$$

□

Method Invocation In T-Invk, function *mType* applied to the unpacked type (N) of the receiver gives the method's signature, $\texttt{<}\overline{\texttt{Y}\lhd\ \texttt{B}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}$. We use the unpacked types ($\overline{\texttt{R}}$) of the actual parameters and the *match* function to infer any 'missing' (actual) type parameters (denoted by $\star$ in our syntax, following Wild FJ). The (possibly inferred) actual type parameters are substituted for formal ($\texttt{[}\overline{\texttt{T/Y}}\texttt{]}$) in the method's type signature. After substitution, the actual type parameters ($\overline{\texttt{T}}$) must be within the formal bounds ($\overline{\texttt{B}}$), and the types of the actual parameters must be subtypes of the types of the formal parameters ($\overline{\texttt{U}}$). These checks are performed under the type environment $\Delta, \Delta', \overline{\Delta}$. Similarly to T-Field, we must guard the conclusion of the type rule with the environments extracted by unpacking ($\Delta', \overline{\Delta}$).

The substitution $\texttt{[}\overline{\texttt{T/Y}}\texttt{]}$ is determined using the types of actual ($\overline{\texttt{R}}$) and formal parameters ($\overline{\texttt{U}}$). These types are filtered using the *sift* function before being passed to *match*. This ensures that where the type of a formal parameter is one of the formal type parameters ($\texttt{U}_i \in \overline{\texttt{Y}}$), the formals and actuals at this position are not used for inference. Hence, we only infer the value of a type variable based on its usage as a type parameter in the formal type of a value argument.

Type parameter inference is done using the *match* relation (Fig. 6). All formal type parameters ($\overline{\texttt{Y}}$) are substituted by types $\overline{\texttt{T}}$. These types are either given explicitly, or are inferred if left unspecified (i.e., marked with $\star$). The first premise of *match* ensures that any unspecified type parameter can be inferred, i.e., it appears as a type parameter in a type of at least one of the method's formal value parameters. The second premise ensures that each specified type parameter is used in the returned sequence. The remaining premises find a substitution that allows subclassing between the formal and actual parameter types. Part of this substitution will be the substitution of actual type parameters for formals, and these actual type parameters are $\overline{\texttt{T}}$. The remainder ($\overline{\texttt{T}'}$) account for existentially quantified type variables in the formal parameter types. These are forgotten, since in T-Invk we use full subtyping which allows us to use the XS-Env rule to fulfil the same role.

Examples — Method invocation *Example 1* from Sect. 2.1 demonstrates method invocation with a simple case of wildcard capture. The existential type ∃Z.Tree<Z> is unpacked to Tree<Z>, and Z is inferred and substituted for X. The return type (List<Z>) is then packed to the existential type ∃Z.List<Z>. We show how the example can be type checked using the T-Invk and T-Subs

rules (the bounds of type variables are omitted for clarity); the type labelled 1 is unpacked to 2 and the type labelled 3 is packed to 4. We omit from the derivation tree the call to *sift* for clarity, note that $sift(\texttt{Tree<Z>}^2, \texttt{Tree<X>}, \texttt{X}) = (\texttt{Tree<Z>}^2, \texttt{Tree<X>})$

$$\dfrac{\dfrac{\begin{array}{c}\emptyset; \texttt{this:C} \vdash \texttt{this} : \texttt{C} \mid \emptyset \\ mType(\texttt{walk}, \texttt{C}) = \texttt{<X> Tree<X>}\rightarrow\texttt{List<X>} \\ \emptyset; \texttt{this:C} \vdash \texttt{y} : \exists \texttt{Z.Tree<Z>}^1 \mid \emptyset \\ match(\texttt{Tree<Z>}^2\texttt{, Tree<X>, }\star\texttt{, X, Z}^2) \\ \texttt{Z}^2 \vdash \texttt{Tree<Z>}^2 <: \texttt{Tree<Z>}\end{array}}{\emptyset; \texttt{this:C} \vdash \texttt{this.<}\star\texttt{>walk(y)} : \texttt{List<Z>}^3 \mid \texttt{Z}^2}\ \text{(T-Invk)} \qquad \begin{array}{c}\texttt{Z}^2 \vdash \texttt{List<Z>}^3 <: \exists \texttt{Z.List<Z>}^4 \\ \emptyset \vdash \texttt{Z}^2\ \text{OK} \\ \emptyset \vdash \exists \texttt{Z.List<Z>}^4\ \text{OK}\end{array}}{\emptyset; \texttt{this:C} \vdash \texttt{this.<}\star\texttt{>walk(y)} : \exists \texttt{Z.List<Z>}^4 \mid \emptyset}\ \text{(T-Subs)}$$

□

Example 2 from Sect. 2.1 expresses types which can not be denoted using Java syntax. Using the syntax of existential types, it becomes clear why type checking fails at `1`. Namely, for the expression at `1` to be type correct, a `T` would need to be found so that $match(\texttt{Pair<U, V>}, \texttt{Pair<X, X>}, \star, \texttt{X}, \texttt{T})$. From the definition of *match* we see that `T` would have to satisfy $\vdash \texttt{Pair<U, V>} \sqsubset: \texttt{[T/X]Pair<X, X>}$; no such `T` exists, and hence *match*ing, and thus type checking, fails.

```
<X>Pair<X, X> make(List<X> x) {}
<X>void compare(Pair<X, X> x) {}

void m()
{
    ∃U,V.Pair<U, V> p;
    ∃Z.List<Z> b;

    this.<⋆>compare(p);                 //1, type incorrect
    this.<⋆>compare(this.<⋆>make(b));   //2, OK
}
```

□

Type Inference As is usual with formal type systems, we consider type inference to be performed in a separate phase before type checking. Due to the presence of existential types, some inferred type parameters can not be named and are marked with $\star$. These parameters must be inferred during type checking. In T-Invk we only allow the inference of types where they are used as parameters to an actual parameter type (e.g., `X` in `<X>void m(Tree<X> x)...`). This is enforced by the *sift* function (defined in Fig. 6), which excludes pairs of actual and formal parameter types where the formal parameter type is a formal type variable of the method.

Computation: $\boxed{\texttt{e} \rightsquigarrow \texttt{e}}$

$$\frac{fields(\texttt{C}) = \overline{\texttt{f}}}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{v}}\texttt{).f}_i \rightsquigarrow \texttt{v}_i} \quad \text{(R-Field)}$$

$$\frac{\begin{array}{c}\texttt{v = new N(}\overline{\texttt{v}'}\texttt{)} \qquad \overline{\texttt{v = new N(}\overline{\texttt{v}''}\texttt{)}} \\ mBody(\texttt{m}, \texttt{N}) = (\overline{\texttt{x}}; \texttt{e}_0) \qquad mType(\texttt{m}, \texttt{N}) = \texttt{<}\overline{\texttt{Y}\lhd\ \texttt{B}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U} \\ match(sift(\overline{\texttt{N}}, \overline{\texttt{U}}, \overline{\texttt{Y}}), \overline{\texttt{P}}, \overline{\texttt{Y}}, \overline{\texttt{T}})\end{array}}{\texttt{v.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{v}}\texttt{)} \rightsquigarrow \texttt{[}\overline{\texttt{v/x}}\texttt{, v/this, }\overline{\texttt{T/Y}}\texttt{]e}_0} \quad \text{(R-Invk)}$$

Congruence: $\boxed{\texttt{e} \rightsquigarrow \texttt{e}}$

$$\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{e.f} \rightsquigarrow \texttt{e}'\texttt{.f}} \quad \text{(RC-Field)} \qquad \frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \rightsquigarrow \texttt{e}'\texttt{.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}} \quad \text{(RC-Inv-Recv)}$$

$$\frac{\texttt{e}_i \rightsquigarrow \texttt{e}'_i}{\texttt{e.<}\overline{\texttt{P}}\texttt{>m(..e}_i\texttt{..)} \rightsquigarrow \texttt{e.<}\overline{\texttt{P}}\texttt{>m(..e}'_i\texttt{..)}} \quad \text{(RC-Inv-Arg)} \qquad \frac{\texttt{e}_i \rightsquigarrow \texttt{e}'_i}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(..e}_i\texttt{..)} \rightsquigarrow \texttt{new C<}\overline{\texttt{T}}\texttt{>(..e}'_i\texttt{..)}} \quad \text{(RC-New-Arg)}$$

Fig. 8. TameFJ reduction rules.

3.5 Operational Semantics

The operational semantics of TameFJ are defined in Fig. 8. Most rules are simple and similar to those in FGJ. The interesting rule is R-Invk, which requires actual type parameters which do not include $\star$, these are found using the *match* relation. Avoiding the substitution of $\star$ for a formal type variable in the method body prevents the creation of invalid expressions, such as `new C<*>()`. Since we are dealing only with values when using this rule, there will be no existential types and so all type parameters *could* be specified. However, there is no safe way to substitute the appropriate types for $\star$s during execution because each $\star$ may mark a different type. In this rule, *mBody* (defined in Fig. 6) is used to lookup the body (an expression) and the formal parameters of the method.

3.6 Type Soundness

We show type soundness for TameFJ by proving progress and subject reduction theorems [27], stated below. We prove these with empty environments since, at runtime, variables and type variables should not appear in expressions. A non-empty guarding environment is required in the statement of the progress theorem, because we use structural induction over the type rules; if this environment were empty, the inductive hypothesis could not be applied in the case of T-Subs.

In the remainder of this section, we summarise some selected lemmas; we list most other lemmas in the appendix. We give full proofs in the extended version of this paper, available from:

http://www.doc.ic.ac.uk/~ncameron/papers/cameron_ecoop08_full.pdf

Theorem 1 (Progress) *For any* $\Delta, \mathtt{e}, \mathtt{T}$*, if* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T} \mid \Delta$ *then either* $\mathtt{e} \rightsquigarrow \mathtt{e}'$ *or there exists a* $\mathtt{v}$ *such that* $\mathtt{e} = \mathtt{v}$*.*

Theorem 2 (Subject Reduction) *For any* $\mathtt{e}, \mathtt{e}', \mathtt{T}$*, if* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T} \mid \emptyset$ *and* $\mathtt{e} \rightsquigarrow \mathtt{e}'$ *then* $\emptyset;\emptyset \vdash \mathtt{e}' : \mathtt{T} \mid \emptyset$*.*

To prove these two theorems, 40 supporting lemmas are required. These establish 'foundational' properties of the system, properties of substitution, properties of subtyping and subclassing (discussed in Sect. 3.2), which functions and relations always give well-formed types, and properties specific to each case of subject reduction and progress. Two of the most interesting lemmas concern the *match* relation:

Lemma 36 (Subclassing preserves *match*ing (receiver)) *If* $\Delta \vdash \exists\Delta_1.\mathtt{N}_1 \sqsubset: \exists\Delta_2.\mathtt{N}_2$ *and* $mType(m, \mathtt{N}_2) = \mathtt{<}\overline{\mathtt{Y}_2 \rightarrow \mathtt{[}\mathtt{B}_{2l}\ \mathtt{B}_{2u}\mathtt{]}}\mathtt{>}\overline{\mathtt{U}_2}\rightarrow\mathtt{U}_2$ *and* $mType(m, \mathtt{N}_1) = \mathtt{<}\overline{\mathtt{Y}_1 \rightarrow \mathtt{[}\mathtt{B}_{1l}\ \mathtt{B}_{1u}\mathtt{]}}\mathtt{>}\overline{\mathtt{U}_1}\rightarrow\mathtt{U}_1$ *and* $match(sift(\overline{\mathtt{R}}, \overline{\mathtt{U}_2}, \overline{\mathtt{Y}_2}), \overline{\mathtt{P}}, \overline{\mathtt{Y}_2}, \overline{\mathtt{T}})$ *and* $\emptyset \vdash \Delta$ OK *and* $\Delta, \Delta' \vdash \overline{\mathtt{T}}$ OK *then* $match(sift(\overline{\mathtt{R}}, \overline{\mathtt{U}_1}, \overline{\mathtt{Y}_1}), \overline{\mathtt{P}}, \overline{\mathtt{Y}_1}, \overline{\mathtt{T}})$*.*

Lemma 37 (Subclassing preserves *match*ing (arguments)) *If* $\Delta \vdash \overline{\exists\Delta_1.\mathtt{R}_1 \sqsubset: \exists\Delta_2.\mathtt{R}_2}$ *and* $match(sift(\overline{\mathtt{R}_2}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}), \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{T}})$ *and* $fv(\overline{\mathtt{U}}) \cap \overline{\mathtt{Z}} = \emptyset$ *and* $\overline{\Delta_2} = \overline{\mathtt{Z}\rightarrow\mathtt{[}\mathtt{B}_l\ \mathtt{B}_u\mathtt{]}}$ *and* $\emptyset \vdash \Delta$ OK *and* $\Delta \vdash \overline{\exists\Delta_1.\mathtt{R}_1}$ OK *and* $\Delta \vdash \overline{\mathtt{P}}$ OK *then there exists* $\overline{\mathtt{U}'}$ *where* $match(sift(\overline{\mathtt{R}_1}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}), \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{[U'/Z]T}})$ *and* $\Delta, \overline{\Delta_1} \vdash \overline{\mathtt{U}' <: \mathtt{[U'/Z]}\mathtt{B}_u}$ *and* $\Delta, \overline{\Delta_1} \vdash \overline{\mathtt{[U'/Z]}\mathtt{B}_l <: \mathtt{U}'}$ *and* $\vdash \overline{\mathtt{R}_1} \sqsubset: \overline{\mathtt{[U'/Z]}\mathtt{R}_2}$ *and* $fv(\overline{\mathtt{U}'}) \subseteq \Delta, \overline{\Delta_1}$*.*

Lemma 36 states that if match succeeds with the formal parameter types of a superclass, then match will succeed where the formal parameter types are taken from the (extended) subclass (and the other arguments remain unchanged). Since overriding methods must have the same parameter types and formal type variables as the methods they override, the proof should be straightforward. However, it is complicated by extended subclassing of existential types; for example, if a method `m` is declared to have a parameter with type `Z` in the class declaration of `class C<Z◁ Object>`, then the type of `m`'s formal parameter will have type `X` when looked up in `∃X.C<X>` and `A` in `C<A>`. `X` may not be a subtype of `A`, even if `C<A>` is an extended subclass of `∃X.C<X>`. We show in the proof that such issues do not affect $\overline{\mathtt{T}}$, because these types are found only from the actual parameter types of the method call.

Lemma 37 performs a similar duty, but for the types of the actual parameters. The conclusion defines a 'valid' substitution which is given by lemma 35 (see Sect. 3.2). The types $\overline{\mathtt{T}}$ in *match* are found from the actual parameter types and so, in contrast to lemma 36, these types are affected by the substitution in the conclusion of the lemma.

Lemma 31 (Inversion Lemma (object creation))
If $\Delta; \Gamma \vdash \texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)} : \texttt{T} \mid \Delta'$ *then* $\Delta' = \emptyset$ *and* $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}$ OK *and* $fields(\texttt{C}) = \overline{\texttt{f}}$ *and* $\overline{fType(\texttt{f}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \texttt{U}}$ *and* $\Delta; \Gamma \vdash \overline{\texttt{e} : \texttt{U} \mid \emptyset}$ *and* $\Delta \vdash \exists\emptyset.\texttt{C<}\overline{\texttt{T}}\texttt{>} <: \texttt{T}$.

Lemma 33 (Inversion Lemma (method invocation)) *If* $\Delta; \Gamma \vdash \texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} : \texttt{T} \mid \Delta'$ *and* $\emptyset \vdash \Delta$ OK *and* $\Delta \vdash \Delta'$ OK *and* $\forall \texttt{x} \in dom(\Gamma) : \Delta \vdash \Gamma(\texttt{x})$ OK *then there exists* Δ_n *where* $\Delta', \Delta_n = \Delta'', \overline{\Delta}$ *and* $\Delta \vdash \Delta', \Delta_n$ OK *and* $\Delta; \Gamma \vdash \texttt{e} : \exists\Delta''.\texttt{N} \mid \emptyset$ *and* $mType(\texttt{m}, \texttt{N}) = \texttt{<}\overline{\texttt{Y}\lhd\texttt{B}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U}$ *and* $\Delta; \Gamma \vdash \overline{\texttt{e} : \exists\Delta.\texttt{R} \mid \emptyset}$ *and* $match(sift(\overline{\texttt{R}}, \overline{\texttt{U}}, \overline{\texttt{Y}}), \overline{\texttt{P}}, \overline{\texttt{Y}}, \overline{\texttt{T}})$ *and* $\Delta \vdash \overline{\texttt{P}}$ OK *and* $\Delta, \Delta'', \overline{\Delta} \vdash \overline{\texttt{T} <: \texttt{[}\overline{\texttt{T/Y}}\texttt{]B}}$ *and* $\Delta, \Delta'', \overline{\Delta} \vdash \exists\emptyset.\texttt{R} <: \texttt{[}\overline{\texttt{T/Y}}\texttt{]U}$ *and* $\Delta, \Delta'', \Delta_n \vdash \texttt{[}\overline{\texttt{T/Y}}\texttt{]U} <: \texttt{T}$.

The formulation of the inversion lemmas is made more interesting by the presence of the guarding environment (Δ') in the typing judgement ($\Delta; \Gamma \vdash \texttt{e} : \texttt{T} \mid \Delta'$). In the case of object creation (lemma 31) we show that the guarding environment must be empty. Intuitively, this is because no existential types may be unpacked in the application of T-New, and T-Subs can only shrink the guarding environment, but not add to it. This property of object creation is used heavily in the proof of subject reduction since values in TameFJ are object creation expressions.

Method invocation is more complex; the guarding environment of T-Invk is formed from the environments unpacked from the types of the receiver and arguments, but these may be re-packed by applying T-Subs. The conclusion of lemma 33 is that there exists some environment, Δ_n, which, when concatenated with Δ' will be equal to the unpacked environments from the receiver and arguments.

Alpha conversion and Barendregt's variable convention As well as the standard use of alpha conversion to rename bound variables in existential types, we also need to be able to rename type variables in the guarding environment, as in the following lemma:

Lemma 7 (Alpha renaming of guarding environments)
If $\Delta; \Gamma \vdash \texttt{e} : \texttt{T} \mid \overline{\texttt{X}\rightarrow\texttt{[B}_l\ \texttt{B}_u\texttt{]}}$ *and* $\overline{\texttt{Y}}$ *are fresh, then*
$\Delta; \Gamma \vdash \texttt{e} : \texttt{[}\overline{\texttt{Y/X}}\texttt{]T} \mid \overline{\texttt{Y}\rightarrow\texttt{[[}\overline{\texttt{Y/X}}\texttt{]B}_l\ \texttt{[}\overline{\texttt{Y/X}}\texttt{]B}_u\texttt{]}}$.

Lemma 7 guarantees that we can rename variables in Δ' and T and preserve typing. Thus, the guarding environment can be thought of as binding its type variables; the scope of the binding is T, the result of type checking. Note that we do not need to rename types in e. This is because any type variables in the domain of the guarding environment ($\overline{\texttt{X}}$) come from unpacked existential types, and so can not be explicitly named in the expression syntax; instead they would be marked with $\star$.

In order to reduce the number of places where we need to apply alpha conversion in our proofs, we make use of Barendregt's variable convention [2]; i.e.,

we assume that bound and free variables are distinct. For example, consider the proof of lemma 2:

Lemma 2 (Subsititution preserves *match*ing) *If* $match(\overline{\mathtt{R}}, \overline{\exists \Delta.\mathtt{R}'}, \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{U}})$ *and* $(\overline{\mathtt{X}} \cup fv(\overline{\mathtt{T}})) \cap \overline{\mathtt{Y}}) = \emptyset$ *then* $match(\overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{R}}, \overline{[\overline{\mathtt{T}/\mathtt{X}}]\exists \Delta.\mathtt{R}'}, \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{P}}, \overline{\mathtt{Y}}, \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}})$.

We reach a point in the proofs where we have shown that $\vdash \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{R}} \sqsubset: \overline{[\overline{\mathtt{T}/\mathtt{X}}]}[\overline{\mathtt{U}/\mathtt{Y}}, \overline{\mathtt{U}'/\mathtt{Z}}]\mathtt{R}'$, $dom(\overline{\Delta}) = \overline{\mathtt{Z}}$, and $(\overline{\mathtt{X}} \cup fv(\overline{\mathtt{T}})) \cap \overline{\mathtt{Y}}) = \emptyset$; we wish to show $\vdash \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{R}} \sqsubset: \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}/\mathtt{Y}}, \overline{[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}'/\mathtt{Z}}][\overline{\mathtt{T}/\mathtt{X}}]\mathtt{R}'$ and for this we require that $\overline{\mathtt{Z}}$ are not free in $\overline{\mathtt{T}}$. We would have used alpha conversion on $\overline{\exists \Delta.\mathtt{R}'}$ to accomplish this; however, this would have required extensive renaming throughout the proof. Instead we use the variable convention and assume that $\overline{\mathtt{Z}}$ are fresh at the point of becoming free and we can proceed with an elegant proof.

The use of Barendregt's variable convention is not always safe [25]. A sufficient condition is that all rules are equivariant and that any binders in a rule do not appear free in that rule's conclusion [25]. Since TameFJ satisfies these conditions, using Barendregt's convention *is* safe.

4 Translating Java to TameFJ

In this section we describe a possible translation from the Java subset which accommodates wildcards into TameFJ.

As said in the introduction, we work in a setting where we expect the first phase to have happened. Here we describe the second phase, and define it in Fig. 10. In Fig. 9 we give the syntax of the relevant subset of Java types, which are also those of Wild FJ.

$\mathtt{N}_s$	::=	$\mathtt{C<}\overline{\mathtt{T}_s}\mathtt{>}$	*Java class types*
$\mathtt{T}_s$	::=	$\mathtt{C<}\overline{\mathtt{P}_s}\mathtt{>} \mid \mathtt{X}$	*Java types*
$\mathtt{P}_s$	::=	$\mathtt{T}_s \mid \mathtt{?} \mid \mathtt{?} \lhd \mathtt{T}_s \mid \mathtt{?} \rhd \mathtt{T}_s$	*Java type parameters*

Fig. 9. Syntax of Java types.

The second phase is defined in terms of the functions $\mathcal{T}$, $\mathcal{P}$, and $\mathcal{M}$, where $\mathcal{T}$ translates Java types to TameFJ types; $\mathcal{P}$ translates a type parameter to an environment and a TameFJ type; and $\mathcal{M}$ gives the minimal types out of two. The function $\mathcal{T}$ maps each occurrence of a wildcard, ?, in a Java type onto an existentially quantified type variable. To do this, it uses the function $\mathcal{P}$, which maps any Java type onto an environment and a TameFJ type. $\mathcal{T}$ uses the collected environments to create an existential type, using the $\mathcal{M}$ function to find the appropriate upper bounds, and replaces each type argument by its image through $\mathcal{P}$. Note that, in order to reduce the notational complexity, the

translation of non-wildcard type parameters introduces a type variable which is never used; this is harmless.

We now highlight some of the finer points of the translation in terms of examples.

$$\frac{\texttt{class C<X}\lhd \texttt{T}_s\texttt{>...} \qquad \overline{\mathcal{P}_\Delta(\texttt{P}_s) = (\texttt{Y} \to \texttt{[U}_s\ \texttt{U}'_s\texttt{]}, \texttt{T})}}{\mathcal{T}_\Delta(\texttt{C<}\overline{\texttt{P}_s}\texttt{>}) = \exists \texttt{Y} \to\ \texttt{[}\mathcal{T}_\Delta(\texttt{U}_s)\ \ \mathcal{M}_\Delta(\mathcal{T}_\Delta(\texttt{U}'_s), \texttt{[}\overline{\texttt{Y/X}}\texttt{]}\mathcal{T}_\Delta(\texttt{T}_s))\texttt{]}\ \texttt{.C<}\overline{\texttt{T}}\texttt{>}}$$

$$\frac{}{\mathcal{T}_\Delta(\texttt{X}) = \exists\emptyset\texttt{.X}} \qquad \frac{\Delta \vdash \texttt{T} <: \texttt{T}'}{\mathcal{M}_\Delta(\texttt{T}, \texttt{T}') = \texttt{T} = \mathcal{M}_\Delta(\texttt{T}', \texttt{T})} \qquad \frac{\Delta \vdash \texttt{T} \not<: \texttt{T}' \qquad \Delta \vdash \texttt{T}' \not<: \texttt{T}}{\mathcal{M}_\Delta(\texttt{T}', \texttt{T}) = \texttt{T}}$$

$$\frac{\texttt{X}\ \textit{is fresh}}{\begin{array}{c}\mathcal{P}_\Delta(\texttt{?}) = (\texttt{X} \to \texttt{[}\bot\ \ \exists\emptyset\texttt{.Object]}, \texttt{X}) \\ \mathcal{P}_\Delta(\texttt{?} \lhd \texttt{T}_s) = (\texttt{X} \to \texttt{[}\bot\ \ \mathcal{T}_\Delta(\texttt{T}_s)\texttt{]}, \texttt{X}) \\ \mathcal{P}_\Delta(\texttt{?} \rhd \texttt{T}_s) = (\texttt{X} \to \texttt{[}\mathcal{T}_\Delta(\texttt{T}_s)\ \ \exists\emptyset\texttt{.Object]}, \texttt{X}) \\ \mathcal{P}_\Delta(\texttt{T}_s) = (\texttt{X} \to \texttt{[}\bot\ \ \exists\emptyset\texttt{.Object]}, \mathcal{T}_\Delta(\texttt{T}_s))\end{array}}$$

Fig. 10. Translation from Java types to TameFJ types.

A wildcard that occurs as a type parameter is replaced by a quantified type variable. Bounds on the wildcard become bounds on the quantifying type variable. Where bounds are not given we use $\exists\emptyset$`.Object` as the default upper bound and $\bot$ as the default lower bound. For instance, `C<?` $\lhd$ `Shape>` is translated to $\exists$`X`$\to$`[`$\bot\ \exists\emptyset$`.Shape].C<X>`, and the translation of `C<?` $\rhd$ `Shape>` amounts to $\exists$`X`$\to$`[`$\exists\emptyset$`.Shape` $\exists\emptyset$`.Object].C<X>`. We must distinguish different occurrences of the wildcard symbol by translating them to distinct type variables. Hence, `Pair<?, ?>` translates to $\exists$`X,Y.Pair<X, Y>`. Finally, nested wildcards are quantified at the immediately enclosing level, so `C<C<C<?>>>` translates to $\exists\emptyset$`.C<`$\exists\emptyset$`.C<`$\exists$`X.C<X>>>`.

A subtle aspect of the translation is that wildcards can inherit their upper bound from the upper bound of the corresponding formal type variable in the class declaration. Since we want to avoid doing this in the calculus, we must take care of this in the translation, which is achieved as in the following example: for a class `C` declared as `class C<Z` $\lhd$ `Circle>...`, the type `C<?>` is translated to $\exists$`X`$\to$`[`$\bot\ \exists\emptyset$`.Circle].C<X>`.

When an upper bound is declared both for a wildcard and in the corresponding class declaration, then the 'smallest' type is taken as the upper bound, if the types are subtypes of each other ($\mathcal{M}$). Hence, `C<?` $\lhd$ `Shape>` is translated to the same type as in the previous example, and is *not* a type error. Finally, if the bounds are unrelated, then the bound from the declaration is taken as the upper bound of the wildcard, which means that even the type `C<?` $\lhd$ `Serializable>` is translated into the same type as the previous two examples.

This last behaviour implies that the Java type analysis uses a more general type for some expressions than it would have to in order to maintain soundness (in the example it could have used the intersection of `Circle` and `Serializable`, but it just uses `Circle`), and this means that some reasonable and actually type safe programs will be rejected by the Java compiler. However, it poses no problems for the soundness of Java, nor for our translation.

The most interesting aspect of the translation is where wildcards meet F-bounds. An F-bounded type is a type where the formal type variable is bounded by an expression in which the variable itself occurs. These types are crucial for modelling common idioms such as subject-observer in Java generics [23]. In the following example both instantiations of `F` using wildcards are legal.

```
class F<X ◁ F<X>> {...}
void m(F<?> x1, F<? ◁ F<?>> x2) {...}
```

The translation of the types `F<?>` and `F<? ◁ F<?>>` is not immediately obvious, because in Java there is no finite type expression for the least supertype of all legal type arguments to `F`, i.e., the upper bound of the type argument `X` is not denotable in Java. However, in TameFJ this upper bound *is*, in fact, denotable: it is just `∃Y→[⊥ F<Y>].F<Y>`. Indeed, our translation of `F<?>` gives this type. In the case of `F<? ◁ F<?>>` where the wildcard is translated to the fresh variable `Y`, the upper bound will be the least subtype of `∃Z.F<Z>` (the translation of the given bound; where `Z` is fresh) and `F<Y>` (the bound derived from the class declaration). Since the latter is more strict, it is used, even though this appears to contradict the rule of using fresh type variables for each wildcard; in fact it does no such thing, the second wildcard *is* translated to a fresh type variable, but is then forgotten.

5 Related Work

In this section we discuss related work. We distinguish three categories: the evolution of wildcards, formal and informal specifications of Java wildcards, and related systems with type soundness results.

Wildcards are a form of *use-site* variance. This means that the variance of a type is determined at the instantiation of the type. The first uses of variant generic types in object oriented languages were *declaration-site* variance, where the variance of a type is determined by the class declaration. Use-site variance was first expressed in terms of structural virtual types [23]. The concept developed into Variant Parametric Types [14] which were extended to give Java wildcards.

Wildcards in Java are officially (and informally) described in the Java Language Specification [11]. Wildcards and generics are described in detail in [3]. Wildcards were first described in a research paper in [24], again informally, but with some description of their formal properties and of the correspondence to existential types. The most important formal description of wildcards is the Wild FJ calculus [17], referred to throughout this paper. Wildcards have also

been described in terms of access restriction [26] and flow analysis [8] (actually Variant Parametric Types).

Variant Parametric Types [14] could be thought of as a partial model for Java wildcards (notably missing wildcard capture, but different in several subtler ways also). The calculus in [14] was proven type sound and as such it can be regarded as a partial soundness result for wildcards. In [5] we describe a sound partial model for wildcards using a more traditional existential types approach. In particular, the calculus has explicit open and close expressions, as opposed to the implicit versions found in this paper and in other approaches [14,17]. Subtyping of existential types in [5] is taken from the full variant of System $F_{<:}$ with existential types [10], rather than the wildcards style subtyping, exemplified in the XS-Env rule in this paper. The soundness result for that system follows those of FGJ and traditional existential types closely. However, it is only a partial result; the system lacks lower bounds amongst other restrictions.

6 Conclusion and Future Work

In this paper we have presented a formal model for Java with wildcards, TameFJ, and a type soundness proof for this formalism. To the best of our knowledge, this is the first type sound model for wildcards that captures *all* the significant features for soundness. We have shown through discussion and a formal translation that TameFJ is a satisfactory model for Java wildcards.

Future Work We are investigating several directions for future work. The most straightforward is to extend our model to include imperative features. Previous work with existential types found issues that only occurred in an imperative setting [12]; although we do not believe these issues affect our result, a proof for an imperative system would settle this matter once and for all. To complete the argument for type soundness in Java, we would like to prove soundness for the translation described in Sect. 4, expanded to expressions. Another interesting property for Java wildcards would be the decidability of typing and type inference. Such questions have been investigated elsewhere [15,19], but there is no complete answer specifically for Java.

We would like to apply the tools developed for this work, i.e., existential types for variance, in other settings. For example, ownership types, where an 'any' or '?' parameter or ad hoc existential types often appear [1,4,16,28]; or virtual classes [9,18]. We would also like to further develop the use of existential types to give programmers a better understanding of how to use wildcards.

Acknowledgements We are deeply grateful to Alex Summers for introducing us to Barendregt's variable convention, and to Christian Urban, Mariangiola Dezani-Ciancaglini, and again Alex Summers, for discussions on the convention. We had illuminating discussions with Alex Buckley about the Java language and spec and the implementation of Wildcards, with Atushi Igarashi about Featherweight Java and FGJ, and with Dave Clarke about existential types and other approaches. We thank the anonymous reviewers for their helpful comments.

References

1. Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, pages 1–25, 2004.
2. Henk Barendregt. *The Lambda Calculus*. North-Holland, Amsterdam, revised edition, 1984.
3. Gilad Bracha. Generics in the Java programming language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
4. Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Ownership. In *OOPSLA 07*, October 2007.
5. Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an existential types model for java wildcards. In *9th Workshop on Formal Techniques for Java-like Programs*, 2007.
6. Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Research report 56, DEC Systems Research Center, 1990.
7. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
8. Wei-Ngan Chin, Florin Craciun, Siau-Cheng Khoo, and Corneliu Popeea. A flow-based approach for variant parametric types. In *Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '06)*. ACM Press, October 2006.
9. Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: More types for virtual classes.
   http://slurp.doc.ic.ac.uk/pubs/tribe.pdf, December 2005.
10. Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75–96, 1998.
11. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.
12. Dan Grossman. Existential types for imperative languages. In *ESOP '02: Proceedings of the 11th European Symposium on Programming Languages and Systems*, pages 21–35, London, UK, 2002. Springer-Verlag.
13. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. An earlier version of this work appeared at OOPSLA'99.
14. Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006. An earlier version appeared as "On variance-based subtyping for parametric types" at (ECOOP'02).
15. Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD'07), Nice, France, January 2007.
16. Yi Lu and John Potter. On ownership and accessibility. In *ECOOP*, pages 99–123, 2006.
17. Mads Torgersen and Erik Ernst and Christian Plesner Hansen. Wild FJ. In *12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12), Long Beach, California*, New York, NY, USA, 2005. ACM Press.
18. O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM Press.

19. Karl Mazurak and Steve Zdancewic. Note on "Type Inference for Java 5: Wildcards, F-Bounds, and Undecidability", 2006. http://www.cis.upenn.edu/šstevez/note.html.
20. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 37–51, New York, NY, USA, 1985. ACM Press.
21. Benjamin C. Pierce. Bounded quantification is undecidable. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 305–315, New York, NY, USA, 1992. ACM Press.
22. Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.
23. Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 186–204, London, UK, 1999. Springer-Verlag.
24. Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. *Journal of Object Technology*, 3(11):97–116, 2004. Special issue: OOPS track at SAC 2004, Nicosia/Cyprus.
25. Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In Frank Pfenning, editor, *CADE*, volume 4603 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2007.
26. Mirko Viroli and Giovanni Rimassa. On access restriction with Java wildcards. *Journal of Object Technology*, 4(10):117–139, 2005. Special issue: OOPS track at SAC 2005, Santa Fe/New Mexico. The earlier version in the proceedings of SAC '05 appeared as Understanding access restriction of variant parametric types and Java wildcards.
27. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
28. Tobias Wrigstad and Dave Clarke. Existential owners for ownership types. *JOT*, 2007. vol. 6, no. 4, May-June 2007, pp. 141-159.

A Summary of Lemmas

For all lemmas and theorems we require the additional premise that the program is well-formed, i.e., for all class declarations, $\texttt{Q}$, in the program, $\vdash \texttt{Q}$ OK. Lemmas in the text have not been repeated, some lemmas have been omitted; full proofs of all lemmas can be downloaded from:

http://www.doc.ic.ac.uk/~ncameron/papers/cameron_ecoop08_full.pdf

Lemma 1 (Substitution preserves subclassing). *If* $\vdash \texttt{R} \sqsubset: \texttt{R}'$ *then* $\vdash \texttt{[}\overline{\texttt{T/X}}\texttt{]R} \sqsubset: \texttt{[}\overline{\texttt{T/X}}\texttt{]R}'$.

Proof is by structural induction on the derivation of $\vdash \texttt{R} \sqsubset: \texttt{R}'$.

Lemma 3 (Substitution on $\overline{\texttt{U}}$ preserves $sift$). *If* $sift(\overline{\texttt{R}}, \overline{\texttt{U}}, \overline{\texttt{Y}}) = (\overline{\texttt{R}_r}, \overline{\texttt{T}_r})$ *and* $(fv(\overline{\texttt{T}}) \cup \overline{\texttt{X}}) \cap \overline{\texttt{Y}} = \emptyset$ *then* $sift(\overline{\texttt{R}}, \overline{\texttt{[T/X]U}}, \overline{\texttt{Y}}) = (\overline{\texttt{R}_r}, \overline{\texttt{[T/X]T}_r})$.

Proof is by structural induction on the derivation of $sift(\overline{\texttt{R}}, \overline{\texttt{U}}, \overline{\texttt{Y}}) = (\overline{\texttt{R}_r}, \overline{\texttt{T}_r})$.

Lemma 4 (Substitution on $\overline{\mathtt{R}}$ preserves $sift$). *If* $sift(\overline{\mathtt{R}}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}) = (\overline{\mathtt{R}_r}, \overline{\mathtt{T}_r})$ *and* f *is a mapping from and to types in the syntactic category* R. *then* $sift(\overline{f(\mathtt{R})}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}) = (\overline{f(\mathtt{R}_r)}, \overline{\mathtt{T}_r})$.

Proof is by structural induction on the derivation of $sift(\overline{\mathtt{R}}, \overline{\mathtt{U}}, \overline{\mathtt{Y}}) = (\overline{\mathtt{R}_r}, \overline{\mathtt{T}_r})$.

Lemma 11 (Weakening of Typing).

If $dom(\Delta, \Delta', \Delta''') \cap dom(\Delta'') = \emptyset$ *and* $dom(\Gamma, \Gamma'') \cap dom(\Gamma') = \emptyset$ *and* $\Delta, \Delta'; \Gamma, \Gamma'' \vdash \mathtt{e} : \mathtt{T} \mid \Delta'''$ *then* $\Delta, \Delta'', \Delta'; \Gamma, \Gamma', \Gamma'' \vdash \mathtt{e} : \mathtt{T} \mid \Delta'''$ *and*

Proof is by structural induction on the derivation of $\Delta, \Delta'; \Gamma, \Gamma'' \vdash \mathtt{e} : \mathtt{T} \mid \Delta'''$.

Lemma 13 (Extension of type environments preserves well-formedness). *If* $\Delta \vdash \Delta'$ OK *and* $\Delta, \Delta' \vdash \Delta''$ OK *then* $\Delta \vdash \Delta', \Delta''$ OK.

Proof is by structural induction on the derivation of $\Delta \vdash \Delta'$ OK.

Lemma 21 (Subsititution preserves typing).

If $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta''$ *and* $\Delta_1 \vdash \overline{\mathtt{T} <: \mathtt{[\overline{T/X}]B}_u}$ *and* $\Delta_1 \vdash \overline{\mathtt{[\overline{T/X}]B}_l <: \mathtt{T}}$ *and* $\Delta = \Delta_1, \overline{\mathtt{X} \rightarrow \mathtt{[B}_l\ \mathtt{B}_u\mathtt{]}}, \Delta_2$ *and* $\Delta' = \Delta_1, \mathtt{[\overline{T/X}]}\Delta_2$ *and* $\overline{\mathtt{X}} \cap fv(\Delta_1) = \emptyset$ *and* $\Delta_1 \vdash \overline{\mathtt{T}}$ OK *and* $\emptyset \vdash \Delta_1$ OK *and* $\Delta_1, \overline{\mathtt{X} \rightarrow \mathtt{[B}_l\ \mathtt{B}_u\mathtt{]}} \vdash \Delta_2$ OK *then* $\Delta'; \mathtt{[\overline{T/X}]}\Gamma \vdash \mathtt{[\overline{T/X}]e} : \mathtt{[\overline{T/X}]T} \mid \mathtt{[\overline{T/X}]}\Delta''$.

Proof is by structural induction on the derivation of $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta''$.

Lemma 22 (Superclasses are well-formed). *If* $\vdash \mathtt{R} \sqsubset: \mathtt{R}'$ *and* $\Delta \vdash \mathtt{R}$ OK *and* $\emptyset \vdash \Delta$ OK *then* $\Delta \vdash \mathtt{R}'$ OK.

Proof is by structural induction on the derivation of $\vdash \mathtt{R} \sqsubset: \mathtt{R}'$.

Lemma 23 (Subclassing preserves field types).

If $\vdash \mathtt{N} \sqsubset: \mathtt{N}'$ *and* $fType(\mathtt{f}, \mathtt{N}') = \mathtt{T}$ *then* $fType(\mathtt{f}, \mathtt{N}) = \mathtt{T}$.

Proof is by structural induction on the derivation of $\vdash \mathtt{N} \sqsubset: \mathtt{N}'$.

Lemma 24 (Subclassing preserves method return type). *If* $\vdash \mathtt{N}_1 \sqsubset: \mathtt{N}_2$ *and* $mType(\mathtt{m}, \mathtt{N}_2) = \mathtt{<\overline{Y \lhd\ T}_u\mathtt{>}\overline{T} \rightarrow T}$ *then* $mType(\mathtt{m}, \mathtt{N}_1) = \mathtt{<\overline{Y \lhd\ T}_u\mathtt{>}\overline{T} \rightarrow T}$.

Proof is by structural induction on the derivation of $\vdash \mathtt{N}_1 \sqsubset: \mathtt{N}_2$.

Lemma 25 (Expression substitution preserves typing). *If* $\Delta; \Gamma, \mathtt{x:U} \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$ *and* $\Delta; \Gamma \vdash \mathtt{e'} : \mathtt{U'} \mid \emptyset$ *and* $\Delta \vdash \mathtt{U'} <: \mathtt{U}$ *and* $\Delta \vdash \mathtt{U}$ OK *then* $\Delta; \Gamma \vdash \mathtt{[e'/x]e} : \mathtt{T} \mid \Delta'$. *Proof is by structural induction on the derivation of* $\Delta; \Gamma, \mathtt{x:U} \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$.

Lemma 29 (*match* gives well-formed types). *If* $\Delta \vdash \overline{\mathtt{P}}$ OK *and* $\Delta \vdash \overline{\exists \Delta'.\mathtt{R}}$ OK *and* $\emptyset \vdash \Delta$ OK *and* $match(\overline{\mathtt{R}}, \overline{\exists \Delta'.\mathtt{R}'}, \overline{\mathtt{P}}, \overline{\mathtt{Y}}, \overline{\mathtt{T}})$ *then* $\Delta, \overline{\Delta} \vdash \overline{\mathtt{T}}$ OK.

Lemma 30 (Typing gives well-formed types).

If $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$ *and* $\emptyset \vdash \Delta$ OK *and* $\forall \mathtt{x} \in dom(\Gamma) : \Delta \vdash \Gamma(\mathtt{x})$ OK *then* $\Delta, \Delta' \vdash \mathtt{T}$ OK.

Proof is by structural induction on the derivation of $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T} \mid \Delta'$.