Towards an Existential Types Model for Java Wildcards

Nick Cameron[1], Erik Ernst[2], and Sophia Drossopoulou[1]

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

Abstract. Wildcards extend Java generics by softening the mismatch between subtype and parametric polymorphism. Although they are a key part of the Java 5.0 programming language, a type system including wildcards has never been proven type sound. Wildcards have previously been formalised as existential types. In this paper we extend FGJ, a featherweight formalisation of Java with generics, with existential types. We prove that this calculus, ∃J, is type sound, and illustrate how it models wildcards in the Java Programming Language. ∃J is not a full model for Java wildcards, because it does not support lower bounds for wildcards. We discuss why ∃J can not be easily extended with lower bounds, and how full Java wildcards could be modelled in a type sound way.

1 Introduction

Wildcards [5,14] have been part of the Java programming language since September 2004 (version 5.0) and are an important part of its type system. Wildcard types make Java generics more usable and powerful and are used throughout the Java libraries. However, to our knowledge, the issue of type safety has not yet been resolved for wildcards. WildFJ [8] describes Java 5.0 fairly closely but has not yet been proven sound. Therefore, a better understanding of the type theoretic background of wildcards is necessary.

Existential types can hide information, and they have been used for abstract data types, modules, and similar features [3,9,11]. They have also been used to model variance in generics and virtual types [7]. Existential types are reckoned to model Java wildcards (another language feature for subtype variance) even more closely [8,14].

We take a step towards solving the open and difficult question of type soundness for Java with wildcards by extending FGJ [6] with existential types, rather than modeling wildcards directly. In the resulting calculus, ∃J, existentially quantified type variables may have upper, but not lower, bounds. Naively adding lower bounds causes problems with the proof of type soundness. Existential types in ∃J are quantified by a single type variable. To fully express wildcard types from Java, multiple type variables must be quantified together. Thus, ∃J does not provide a complete solution, but we consider it a first, significant step toward proving type soundness of wildcards.

Our contributions are a description, formalisation and type soundness proof of an object-oriented programming language with existential types for subtype variance, ∃J; we discuss the correspondence between Java wildcards and existential types, and the difficulties in using an existential types calculus as a full model for Java with wildcards, in particular, including lower bounds in the calculus.

The next section briefly describes existential types and Java wildcards. Section 3 presents ∃J and the soundness proof. Section 4 discusses the relation of ∃J to Java with wildcards, and outlines future work. Finally, Sect. 5 concludes.

2 Background

In this section we describe the previous uses and formalisations of existential types, how existential types have been used to address the subtype variance problem, and introduce Java wildcards.

2.1 Existential Types for Abstract Data Types

Existential types have been widely studied as a polymorphic type system used for data abstraction and information hiding, for example to model abstract data types and objects [2,3,4,9,10,11]. Here, type variables may be quantified existentially; a quantified type hides information about the actual type (the *witness type*). An entity with such a type can be regarded as an opaque package. It can be created by a *close* (or *pack*) expression; the components of the package can only be used (*open*ed or *unpack*ed) in a context that preserves the hiddenness of the witness type. Partial knowledge of the witness type can be expressed and preserved via bounds on the existentially quantified type variables.

2.2 Parametric Polymorphism in Java

Parametric polymorphism is implemented in Java using generics [1,5], whereby classes (and types) or methods may be parameterised by a list of type parameters; the actual type parameters may be class types (possibly parameterised) or type variables. For example, a very simple container class could be defined as:

```
class Box<X> {
    X data;
    X get() { return data; }
    void set(X x) { data = x; }
}
```

X is the formal type parameter. The box type may be instantiated as Box<String>, Box<Object>, etc. When a type variable Y is in scope, we may also write Box<Y>.

Type variables may be given bounds using the extends keyword. For example, assuming a hierarchy of classes where Poodle extends Dog, extends Animal, extends `Object`, we may declare `class BoundedBox<X extends Dog>`. In this case, we may instantiate the types `BoundedBox<Poodle>` and `BoundedBox<Dog>`, but not `BoundedBox<Animal>` or `BoundedBox<Object>`.

Generic types are invariant with respect to subtyping of their parameters; in the above example `Box<Poodle>` is not a subtype of `Box<Dog>`. Although this relationship (covariance) seems logical and desirable, it is actually unsound:

```
Box<Poodle> boxOfPoodles = new Box<Poodle>();
Box<Dog> boxOfDogs = boxOfPoodles;    \\illegal in Java
boxOfDogs.set(new Rottweiler());
Poodle p = boxOfPoodles.get();    \\arghh! We got a rottweiler!
```

2.3 Existential Types for Subtype Variance

There have been many different proposals for incorporating subtype variance in parametrically polymorphic languages in a type safe way. These include structural virtual types [13], variant parametric types [7] and wildcards [14]. Variant parametric types of [7] are the closest to (and the inspiration for) Java wildcards; the authors used a restricted form of existential types for their formalism and proof of type soundness. Variant parametric types were extended into wildcard parametric types in [15], where an alternative formalism to [8] is presented.

Existential types were first mentioned in the context of subtype variance in [12], the concept was developed in [7]. Bounded existential types allow type safe variance since they only reveal partial information about the hidden types.

2.4 Wildcards

A wildcard type [14] is a type with `?` (the wildcard) as an actual type parameter, for example `Box<?>` — a box of some type. The wildcard parameter may be bounded above (eg `Box<? extends Dog>`) or below (`Box<? super Dog>`); the former type acts covariantly with respect to its type parameter (`Box<Poodle>` is a subtype of `Box<? extends Dog>`), the latter contravariantly (`Box<Animal>` is a subtype of `Box<? super Dog>`).

Crucial to understanding wildcards is that a wildcard hides the actual type argument given for the corresponding type parameter—so `?` in `Box<? super Dog>` may hide the type `Animal`, e.g., when an actual value of this type is `new Box<Animal>()`. The wildcard's bound is a bound on this actual type argument, not a bound on the type of objects with the hidden type— so this box may contain a `new Cat()`, even if `?` hides `Animal` and not `Cat`. The type checker cannot know which type `?` hides, so no other value than `null` can be used as an argument to `set`; conversely, every type that `?` can hide is a subtype of `Dog`, and the value returned by `get` is again a subtype of that, so it is safe to consider that return value to have type `Dog`.

Variant parametric types [7] and wildcards express similar types; for example, using variant parametric types `Box<? extends Dog>` is expressed as `Box<+Dog>`.

Both mechanisms can be formalised using existential types [7]. However, as opposed to variant parametric types, wildcards allow *capture conversion*. This is the conversion of a wildcard to a type variable. The effect of capture conversion is *wildcard capture*: a wildcard type may be used where a generic type is expected. This is most obvious during method invocation:

```
<X> List<X> m1(Box<X> x) {..}
List<?> m2(Box<?> y) {  return this.m1(y);  }
```

The use of `y` as a `Box<X>` in the call `this.m1(y)` is legal even though `Box<?>` is not a subtype of `Box<X>` (this would be unsound [5]); the wildcard is capture converted to a fresh type variable which is substituted for `X`. Such an example could not be represented in a type correct way using variant parametric types [7].

In the same way that existential types can be used to model variant parametric types, they can be used as a model for wildcards [8,14]. Furthermore, it has been suggested that wildcard capture is equivalent to opening an existential type [8,14]. This correspondence is explored in more depth in Sect. 4.1.

3 ∃J

In this section we describe ∃J, an object oriented language with generics and existential types. We extended FGJ (itself an extension of Featherweight Java) [6], by adding existential quantification to the syntax of types, and expressions which introduce and eliminate existential types.

Our notation (and style of presentation) is taken from FGJ. In particular, the overbar notation ($\overline{\mathtt{x}}$) denotes a sequence of tokens, $\emptyset$ represents the empty sequence, and an overbar over multiple tokens dentotes a sequence of these tokens (for example, $\overline{\mathtt{a}\ \mathtt{b}}$ for $\mathtt{a}_0\ \mathtt{b}_0\mathtt{,}\ \mathtt{a}_1\ \mathtt{b}_1\mathtt{,}\ \mathtt{a}_2\ \mathtt{b}_2\mathtt{,}\ldots$). We use a comma to concatenate two sequences and implicitly require that there are no duplicates in the resulting sequence. We use $\lhd$ as shorthand for `extends` in Java. Like FGJ, and in contrast to Java, ∃J does not include type inference. Hence, all actual type parameters (to methods and classes) must be specified explicitly. We allow alpha-renaming of type variables in the usual (scope respecting) way.

3.1 Syntax

The syntax of ∃J is given in Fig. 1. The interesting expressions are `open` and `close`: they are discussed in more depth in Sect. 3.4. Values include `new` expressions as in FGJ, and also `close` expressions to allow existentially typed values. The syntax of types consists of class types (`N`) and type variables (`X`) (together non-existential types (`R`)) and existentially quantified types. Existential types are quantified by a single type parameter rather than a sequence of them (as in [8]). This takes after traditional existential types, for example [11], together with the well-formedness constraints on environments this restricts the expressivity of ∃J compared to Java, see also Sect. 4.1. A further distinction is made

Q	::=	class C<Δ> $\triangleleft$ N $\{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{M}}\}$	*class declarations*
M	::=	<Δ> T m($\overline{\mathtt{T\,x}}$) {return e;}	*method declarations*
e	::=	x | this | e.f | e.<$\overline{\mathtt{P}}$>m($\overline{\mathtt{e}}$) | new C<$\overline{\mathtt{P}}$>($\overline{\mathtt{e}}$)	*expressions*
		| open e, δ as x in e | close e with δ hiding T	
v	::=	new C<$\overline{\mathtt{P}}$>($\overline{\mathtt{v}}$) | close v with δ hiding T	*values*

N	::=	C<$\overline{\mathtt{P}}$>	*class types*	δ	::=	X$\triangleleft$ T	*type bounds*
R	::=	N | X	*non-existential types*	Δ	::=	$\overline{\delta}$	*type environments*
T, U	::=	$\overline{\exists\delta}$.R	*types*	Γ	::=	$\overline{\mathtt{x{:}T}}$	*environments*
K	::=	$\overline{\exists\delta}$.N	*non-variable types*				
V	::=	$\overline{\exists\delta}$.X	*variable types*	x, y			*variables*
P	::=	K | X	*type parameters*	C, D, E, F			*classes*
				X, Y, Z			*type variables*

Fig. 1. Syntax of ∃J.

between variable (V) and non-variable (K) types. This simplifies the formalism and proofs but does not introduce any further types. Type parameters exclude only types of the form $\exists\mathtt{X}\triangleleft\ \mathtt{T.V}$, this simplifies the soundness proof and follows Java. Environments (Γ) map variables to their types, and type environments (Δ) map type variables to their bounds.

3.2 Subtyping

$$\frac{}{\Delta \vdash \mathtt{R} <: \mathtt{R}}\quad (\exists\text{S-}\textsc{Reflex})$$

$$\frac{\Delta \vdash \mathtt{R} <: \mathtt{R}'' \qquad \Delta \vdash \mathtt{R}'' <: \mathtt{R}'}{\Delta \vdash \mathtt{R} <: \mathtt{R}'}\quad (\exists\text{S-}\textsc{Trans})$$

$$\frac{}{\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X})}\quad (\exists\text{S-}\textsc{Bound})$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\ \triangleleft\ \mathtt{P}'}\mathtt{>}\ \triangleleft\ \mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{P}}\mathtt{>} <: [\overline{\mathtt{P/X}}]\mathtt{N}}\quad (\exists\text{S-}\textsc{Sub-Class})$$

$$\frac{\Delta \vdash \mathtt{U} <: \mathtt{U}' \qquad \Delta, \mathtt{X} \triangleleft \mathtt{U} \vdash \mathtt{T} <: \mathtt{T}'}{\Delta \vdash \exists\mathtt{X} \triangleleft \mathtt{U.T} <: \exists\mathtt{X} \triangleleft \mathtt{U}'\mathtt{.T}'}\quad (\exists\text{S-}\textsc{Full})$$

Fig. 2. ∃J subtyping.

Subtyping (Fig. 2) is very similar to that of FGJ. The use of type R ensures that most subtype rules only apply to non-existential types; only ∃S-Full (taken from the 'full' variant of System $F_{<}$) applies to existential types (∃S-Full may also apply to an existential type if the upper bound of a type variable is existentially quantified; however, it does not allow an existential type to be the subtype of a non-existential type). This gives that two existential types are subtypes if their quantified types are subtypes (covariance) and if the more precise type has a more restrictive upper bound.

Note that, in contrast to the Java programming language, there is no subtype relation between existential and non-existential types (except as bounds — an unquantified type variable may have an existential type as its upper bound). This is explained in more detail in Sect. 4.1.

$$\frac{}{\Delta \vdash \texttt{Object}\ \text{OK}}$$
(∃F-Object)

$$\frac{\texttt{X} \in \Delta}{\Delta \vdash \texttt{X}\ \text{OK}}$$
(∃F-Var)

$$\frac{\texttt{class C<}\overline{\texttt{X} \lhd \texttt{T}}\texttt{>} \lhd \texttt{N}\ \{\ldots\} \qquad \Delta \vdash \overline{\texttt{P}}\ \text{OK} \qquad \Delta \vdash \overline{\texttt{P <: [}\overline{\texttt{P/X}}\texttt{]T}}}{\Delta \vdash \texttt{C<}\overline{\texttt{P}}\texttt{>}\ \text{OK}}$$
(∃F-Class)

$$\frac{\Delta \vdash \texttt{T}\ \text{OK} \qquad \Delta, \texttt{X} \lhd \texttt{T} \vdash \texttt{U}\ \text{OK}}{\Delta \vdash \exists \texttt{X} \lhd \texttt{T.U}\ \text{OK}}$$
(∃F-Exist)

$$\frac{}{\vdash \emptyset\ \text{OK}}$$
(∃F-Empty)

$$\frac{\Delta \vdash \texttt{T}\ \text{OK} \qquad \vdash \Delta\ \text{OK}}{\vdash \Delta, \texttt{X} \lhd \texttt{T}\ \text{OK}}$$
(∃F-Env)

Fig. 3. ∃J well-formed types and type environments.

3.3 Well-formedness

Well-formedness rules for types and type environments are given in Fig. 3. The well-formedness rules for type-environments are more constrictive than may be expected for Java. Under our rules foward references are entirely forbidden; however, in Java some forward references are allowed (for example in Java `<X extends Box<Y>, Y extends Box<X>>` would be a legal set of formal type parameters, but `<X extends Y, Y extends X>` would not be, whereas in ∃J both are illegal). Although there is some loss of expressivity, this is not an important restriction in ∃J because the interesting effect is felt when such forward references appear in existential types. However, existential types with forward references are not permitted in ∃J since existential quantification only occurs with a single type variable (as opposed to a type environment, ie multiple type variables, as in WildFJ [8]). The issue is side stepped in GJ [6], where type variables may only have a class type as an upper bound.

3.4 Typing

The type rules are given in Fig. 5. Of interest is that we require the receiver in method call and field access, and the arguments in method call, to have non-existential type (`R`). This forces the use of an `open` expression, corresponding to wildcard capture in Java.

$$\mathit{fields}(\mathtt{Object}) = (\emptyset;\emptyset)$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}}\mathtt{>}\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{M}}\}\\ \mathit{fields}([\overline{\mathtt{P/X}}]\mathtt{N}) = (\overline{\mathtt{U}};\overline{\mathtt{g}})\end{array}}{\mathit{fields}(\mathtt{C<}\overline{\mathtt{P}}\mathtt{>}) = (\overline{\mathtt{U}}, [\overline{\mathtt{P/X}}]\overline{\mathtt{T}}; \overline{\mathtt{g}}, \overline{\mathtt{f}})}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}}\mathtt{>}\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mBody}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}) = \mathit{mBody}(\mathtt{m}, [\overline{\mathtt{P/X}}]\mathtt{N})}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}}\mathtt{>}\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{M}}\}\\ \mathtt{<}\Delta\mathtt{>\ U\,m\,(}\overline{\mathtt{U\,x}}\mathtt{)}\ \{\mathtt{return\ e_0;}\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mBody}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}) = (\overline{\mathtt{x}}; [\overline{\mathtt{P/X}}]\mathtt{e_0})}$$

$$\frac{\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}}\mathtt{>}\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{M}}\} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mType}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}) = \mathit{mType}(\mathtt{m}, [\overline{\mathtt{P/X}}]\mathtt{N})}$$

$$\frac{\begin{array}{c}\mathtt{class\ C<}\overline{\mathtt{X}\lhd\ \mathtt{T}}\mathtt{>}\ \lhd\ \mathtt{N}\ \{\overline{\mathtt{T\,f;}}\ \overline{\mathtt{M}}\}\\ \mathtt{<}\Delta\mathtt{>\ U\,m\,(}\overline{\mathtt{U\,x}}\mathtt{)}\ \{\mathtt{return\ e_0;}\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mType}(\mathtt{m}, \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}) = [\overline{\mathtt{P/X}}](\Delta.\overline{\mathtt{U}} \rightarrow \mathtt{U})}$$

$$\frac{}{\mathit{bound}_{\Delta}(\mathtt{K}) = \mathtt{K}}$$

$$\frac{\Delta(\mathtt{X}) = \mathtt{T}}{\mathit{bound}_{\Delta}(\mathtt{X}) = \mathit{bound}_{\Delta}(\mathtt{T})}$$

$$\frac{}{\mathit{bound}_{\Delta}(\exists\delta\mathtt{.T}) = \exists\delta\mathtt{.}\mathit{bound}_{\Delta,\delta}(\mathtt{T})}$$

Fig. 4. Auxiliary functions for ∃J.

$$\frac{}{\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ (\exists\text{T-VAR})$$

$$\frac{}{\Delta;\Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this})}\ (\exists\text{T-THIS})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e} : \mathtt{R}\\ \mathit{fields}(\mathit{bound}_{\Delta}(\mathtt{R})) = (\overline{\mathtt{T}};\ \overline{\mathtt{f}})\end{array}}{\Delta;\Gamma \vdash \mathtt{e.f}_i : \mathtt{T}_i}\ (\exists\text{T-FIELD})$$

$$\frac{\begin{array}{c}\Delta \vdash \overline{\mathtt{P}}\ \text{OK} \qquad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{R}\\ \mathit{mType}(\mathtt{m},\ \mathit{bound}_{\Delta}(\mathtt{R})) = \overline{\mathtt{X}\lhd\ \mathtt{T}}\mathtt{.}\overline{\mathtt{U}} \rightarrow \mathtt{U}\\ \Delta;\Gamma \vdash \overline{\mathtt{e} : \mathtt{U}'} \qquad \overline{\Delta \vdash \mathtt{U}' <: [\overline{\mathtt{P/X}}]\mathtt{U}}\\ \overline{\Delta \vdash \mathtt{P} <: [\overline{\mathtt{P/X}}]\mathtt{T}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e.<}\overline{\mathtt{P}}\mathtt{>m(}\overline{\mathtt{e}}\mathtt{)} : [\overline{\mathtt{P/X}}]\mathtt{U}}\ (\exists\text{T-INVK})$$

$$\frac{\begin{array}{c}\Delta \vdash \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}\ \text{OK}\\ \mathit{fields}(\mathtt{C<}\overline{\mathtt{P}}\mathtt{>}) = (\overline{\mathtt{U}};\ \overline{\mathtt{f}})\\ \Delta;\Gamma \vdash \overline{\mathtt{e} : \mathtt{T}} \qquad \Delta \vdash \overline{\mathtt{T} <: \mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ C<}\overline{\mathtt{P}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}}\ (\exists\text{T-NEW})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_1 : \mathtt{U}' \qquad \Delta \vdash \mathtt{U}' <: \exists\delta\mathtt{.U}\\ \Delta \vdash \exists\delta\mathtt{.U}\ \text{OK} \qquad \vdash \Delta,\delta\ \text{OK}\\ \Delta,\delta;\Gamma,\mathtt{x{:}U} \vdash \mathtt{e}_2 : \mathtt{T}' \qquad \Delta,\delta \vdash \mathtt{T}' <: \mathtt{T}\\ \delta = \mathtt{X}\lhd\ \mathtt{T}'' \qquad \mathtt{X} \notin \mathit{fv}(\mathtt{T})\end{array}}{\Delta;\Gamma \vdash \mathtt{open\ e_1,}\ \delta\ \mathtt{as\ x\ in\ e_2} : \mathtt{T}}\ (\exists\text{T-OPEN})$$

$$\frac{\begin{array}{c}\delta = \mathtt{X}\lhd\mathtt{T}' \qquad \Delta \vdash \mathtt{T}'\ \text{OK}\\ \Delta;\Gamma \vdash \mathtt{e} : \mathtt{U}' \qquad \Delta \vdash \mathtt{U}' <: \mathtt{[T/X]U}\\ \Delta \vdash \mathtt{T} <: \mathtt{[T/X]T}'\end{array}}{\Delta;\Gamma \vdash \mathtt{close\ e\ with}\ \delta\ \mathtt{hiding\ T} : \exists\delta\mathtt{.U}}\ (\exists\text{T-CLOSE})$$

Fig. 5. ∃J expression typing rules.

The open expression (∃T-Open) takes an expression with existential type ($\mathtt{e}_1$) and unpacks it in the scope of a second sub-expression ($\mathtt{e}_2$). The unpacked expression is bound to a fresh variable `x`. The second expression is type checked under the surrounding environment (Γ) extended with $\mathtt{x{:}U}$, and the surrounding type environment (Δ) extended with the quantifying type variable (δ).

The close expression (∃T-Close) is also type checked in a similar way to traditional existential types. An expression is 'packed' in a close expression and

$$\frac{\begin{array}{c}\Delta,\Delta' \vdash \texttt{U},\bar{\texttt{U}}\ \textsc{ok} \qquad \vdash \Delta,\Delta'\ \textsc{ok} \qquad \texttt{class C<}\overline{\texttt{X}\lhd\ \texttt{T}}\texttt{>}\ \lhd\ \texttt{N}\ \{\ldots\} \\ \Delta,\Delta';\overline{\texttt{x:U}},\ \texttt{this:C<}\bar{\texttt{X}}\texttt{>} \vdash \texttt{e}_0\texttt{:T} \qquad \Delta,\Delta' \vdash \texttt{T} <: \texttt{U} \qquad \mathit{override}_{\Delta,\Delta'}(\texttt{m},\texttt{N},\Delta'.\bar{\texttt{U}} \rightarrow \texttt{U})\end{array}}{\Delta \vdash \texttt{<}\Delta'\texttt{>U m(}\overline{\texttt{U x}}\texttt{)}\ \{\texttt{return e}_0\}\ \textsc{ok}\ \texttt{in C}}$$

(∃T-Method)

$$\frac{\begin{array}{c}\mathit{mType}(\texttt{m},\texttt{N}) = \Delta'.\bar{\texttt{T}} \rightarrow \texttt{T}' \\ \Delta \vdash \texttt{T} <: \texttt{T}'\end{array}}{\mathit{override}_{\Delta}(\texttt{m},\texttt{N},\Delta'.\bar{\texttt{T}} \rightarrow \texttt{T})}$$

(∃T-Override)

$$\frac{\mathit{mType}(\texttt{m},\texttt{N}) \quad \mathit{undefined}}{\mathit{override}_{\Delta}(\texttt{m},\texttt{N},\Delta'.\bar{\texttt{T}} \rightarrow \texttt{T})}$$

(∃T-OverrideUndef)

$$\frac{\begin{array}{c}\Delta \vdash\ \texttt{N},\ \bar{\texttt{T}}\ \textsc{ok} \qquad \vdash \Delta\ \textsc{ok} \qquad \Delta \vdash \bar{\texttt{M}}\ \textsc{ok}\ \texttt{in C} \\ \mathit{fields}(\texttt{N}) = (\overline{\texttt{T}'},\overline{\texttt{f}'}) \qquad \bar{\texttt{f}} \cap \overline{\texttt{f}'} = \emptyset\end{array}}{\vdash \texttt{class C<}\Delta\texttt{>}\ \lhd\ \texttt{N}\ \{\overline{\texttt{T f;}}\ \bar{\texttt{M}}\}\ \textsc{ok}}$$

(∃T-Class)

Fig. 6. ∃J class and method typing rules.

$$\frac{\mathit{fields}(\texttt{C<}\bar{\texttt{P}}\texttt{>}) = (\bar{\texttt{U}};\ \bar{\texttt{f}})}{\texttt{new C<}\bar{\texttt{P}}\texttt{>(}\bar{\texttt{v}}\texttt{).f}_i \leadsto \texttt{v}_i}$$

(∃R-Field)

$$\frac{\begin{array}{c}\mathit{mBody}(\texttt{m},\texttt{C<}\overline{\texttt{P}'}\texttt{>}) = (\bar{\texttt{x}};\texttt{e}_0) \\ \mathit{mType}(\texttt{m},\texttt{C<}\overline{\texttt{P}'}\texttt{>}) = \overline{\texttt{X}\lhd\ \texttt{T}}.\bar{\texttt{U}}\ \rightarrow\ \texttt{U}\end{array}}{\begin{array}{c}\texttt{new C<}\overline{\texttt{P}'}\texttt{>(}\overline{\texttt{v}'}\texttt{).<}\bar{\texttt{P}}\texttt{>m(}\bar{\texttt{v}}\texttt{)} \leadsto \\ [\overline{\texttt{v/x}},\texttt{ new C<}\overline{\texttt{P}'}\texttt{>(}\overline{\texttt{v}'}\texttt{)/this, }\overline{\texttt{P/X}}]\texttt{e}_0\end{array}}$$

(∃R-Invk)

$$\frac{}{\texttt{open close v with X}\ \lhd\texttt{T}_1\ \texttt{hiding T, X}\ \lhd\texttt{T}_2\ \texttt{as x in e} \leadsto [\texttt{T/X, v/x}]\texttt{e}}$$

(∃R-Open-Close)

Fig. 7. ∃J computation rules.

its type is quantified with the given type variables. We must keep track of the hidden and hiding types in the syntax to ensure sound reduction.

The typing rules for methods and classes are given in Fig. 6. They make use of the well-formedness rules for type environments (Fig. 3), these are specified externally of the method and class typing rules to simplify the soundness proof.

3.5 Operational Semantics

The operational semantics of ∃J is given through computation (Fig. 7) and congruence rules. The latter are straightforward and have been elided. ∃R-Open-Close is taken almost directly from the world of existential types [11]. It is used to reduce an `open` expression where the first sub-expression (the expression that is opened) is a `close` value, it eliminates the `close` and open expressions and the scoped sub-expression of the open expression is the result (with the appropriate substitutions). For example:

```
open
  close new Box<Poodle>() with X extends Dog hiding Poodle,
X extends Dog as x in
  this.<X>m(x);
```

reduces to: `this.<Poodle>m(new Box<Poodle>())`. The `close` subexpression in the initial expression has type $\exists$`X`$\lhd$ `Dog.Box<X>`. Assuming `m` has type `<X`$\lhd$ `Dog>.Box<X>` $\rightarrow$ `Dog` then both the intial and reduced expressions have type `Dog`.

3.6 Type Soundness

Type soundness is proven by showing progress and preservation (subject reduction) properties. These state that any well-typed expression is a value or can be reduced to a well-typed expression, and that if a well-typed expression reduces to a second expression then the type of this expression is a subtype of the type of the original expression.

The main difficulty in proving type soundness has been accomodating the `open` and `close` expressions, and adjusting the subtyping and well-formedness rules for handling bounds. We expended a great deal of effort attempting to handle lower bounds in the system, and to handle upper bounds as similarly as possible to Java. We had several generations of lemmas to handle the various attempts. In the end we have a system that is closer to traditional existential types and a little further from Java. The parts of the proofs that were most interesting were often the `open` expression cases (for example in the proof of theorem 2). Those involving detailed manipulation of type environments (for example our substitution lemma) were the hardest to get entirely correct.

Theorem 1 (progress). *For any well-formed expression,* $\mathtt{e}$ *where* $\emptyset;\emptyset \vdash \mathtt{e} : \mathtt{T}$*, either there exists* $\mathtt{e}'$ *where* $\mathtt{e} \rightsquigarrow \mathtt{e}'$ *or* $\mathtt{e}$ *is a value,* $\mathtt{v}$*.*

Theorem 2 (subject reduction). *For any* Δ*,* Γ *where* $\vdash \Delta$ OK *and* $\Delta \vdash \Gamma$ OK *and any expressions* $\mathtt{e}$ *and* $\mathtt{e}'$ *where* $\mathtt{e} \rightsquigarrow \mathtt{e}'$ *and* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *then* $\Delta;\Gamma \vdash \mathtt{e}' : \mathtt{T}'$ *and* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$*.*

4 Discussion

We now discuss the relation between ∃J and Java with wildcards, the difficulties in adding lower bounds to ∃J and how a complete, type sound model for Java with wildcards may be developed.

4.1 ∃J as a Model for Java Wildcards

The correspondence between wildcard types and existential types has been discussed elsewhere [8,14]. In summary, a wildcard becomes an existentially quantified type variable, quantified immediatly outside the class type. Wildcard bounds are trnslated into bounds on the quantified type variable. Multiple wildcards are translated into unique type variables. The following table gives an overview:

```
Box<?>               ⟶  ∃X.Box<X>
Box<Box<?>>          ⟶  Box<∃X.Box<X>>
Box<? extends Dog>   ⟶  ∃X◁ Dog.Box<X>
Pair<?,?>            ⟶  ∃X.∃Y.Pair<X,Y>
```

∃J has similar subtyping properties, between existential types, as wildcard types in Java (covariance with respect to the bound, subclassing, reflexivity and transitivity). However, as opposed to Java, there is no subtyping between existential and non-existential types. So, although `Box<Dog>` is a subtype of `Box<?>` in Java, it is not a subtype of `∃X.Box<X>` in ∃J. To translate Java to ∃J, wherever such subtyping occurs, a `close` expression is inserted; for example (omitting bounds for clarity):

```
void m1(Box<?> x) {...}
void m2(Box<Dog> y) { this.m1(y); }
```

is translated to:

```
void m1(∃X.Box<X> x) {...}
void m2(Box<Dog> y)} { this.m1(close y with X hiding Dog); }
```

Similarly, wildcard capture, performed implicitly in Java, is translated to the surface syntax in ∃J. It has previously been noted that wildcard capture is similar to opening an existential type [8,14]; in ∃J both an open and close expression is required, the latter to prevent the escape of any introduced type variable; for example:

```
<X>Box<X> m1(Box<X> x) {...}
Box<?> m2(Box<?> y) { this.m1(y); }
```

is translated to (note how opening the existential type allows us to provide an actual type parameter to `m1`):

```
<X>Box<X> m1(Box<X> x) {...}
∃Z.Box<Z> m2(∃Y.Box<Y> y) {
  open y,Y as y2 in
     close
       this.<Y>m1(y2)    \\has type Box<Y>
     with Z hiding Y;    \\has type ∃Z.Box<Z>
}
```

One interpretation of this relationship is that Java wildcards provide existential types to the main stream, without the hassle of opening and closing. The remaining challenge is then to show that this does not compromise type safety.

The most obvious obmission in ∃J is the lack of lower bounds. Moreover, ∃J can not model certain classes and types due to the restrictive combination of quantifying existential types by a single type parameter and the well-formedness rules for environments. In Java classes may be specified where formal type parameters are used as actual type parameters in bounds, for example `class C<X`

$\lhd$ `C<Y>, Y` $\lhd$ `C<X>>...`, by complex use of wildcard capture, wildcard (existential) types can be expressed that have a similar relationship in the bounds. Addressing this issue is further work, but should be less significant than lower bounds for the soundness proof.

4.2 Problems with Adding Lower Bounds

The first problem is that by straightforwardly adding lower bounds and the obvious lower bound subtyping rule, we allow (by transitivity) subtypes that are not linked by subclassing. For example, imagine `A` and `B` are direct subclasses of `Object`, by declaring a type variable with lower bound `A` and upper bound `B`, within the scope of the type variable we may deduce that `A` is a subtype of `B`, even though this is clearly unsound. Restricting the lower bound of a type variable to a subclass of the upper bound is the obvious solution, but this is complicated by the lack of subtyping between existential and non-existential types and can (undesirably) restrict the bounds of a type variable. Note that in Java the bounds of a wildcard are restricted: only one may be specified and the other may be inherited from the definition; this simplifies the situation. The underlying problem here is that in translating Java to ∃J we assume that subtyping involving an existential and non-existential type can be translated using a `close` expression as in section 4.1. However, there are cases where subtyping occurs without any expression being present, for example, when checking the well-formedness of bounds.

The second problem is that, in the presence of the obvious lower bound subtyping rule, a subtype of a non-existential type may be existentially quantified. This is not possible in ∃J (by the ∃S-Bound an existential type may be the supertype of a non-existential type, but this is benign), but the property is necessary to prove soundness. This is apparent in the congruence case for field access and method call; here the type rules require a sub-expression with non-existential type, if (as is possible with lower bounds) this sub-expression may reduce to an expression with existential type, then the type rule may not be applied, and the subject reduction property does not hold.

4.3 Towards a Full Model for Wildcards

As explained above, an extension of ∃J to model full Java with wildcards, and proof of type soundness is not straightforward.

We see three possibilites: the first possibilty would use large step semantics. This would address the following problems: if we extend ∃J in a naïve way to include lower bounds, then we can have expressions which have non-existential type, but which reduce in one step to an expresssion with existential type. If the original expression appeared in a context which required an expression with non-existential type, eg field access, then subject reduction would not hold. On the other hand, if we expect all values to have non-existential type then the problem should not arise with large step semantics.

The second possibilty would incorporate the close expression into subtyping (as is done in WildFJ) and replacing the open expression with a capture expression that performs an open and close as descibed in Sect. 4.1. We have previously explored this approach and found problems with the proof of type soundness; however, these may be solvable.

The third possibilty would involve proving type soundness for a system more in the spirit of WildFJ, that is, without explicit open and close expressions. Restrictions on syntax could help with the well-formedness checks.

5 Conclusion

We have shown that existential types used for variance in a generic, object-oriented setting are type sound. Our model includes explicit open and close expressions and therefore goes further to model the unique features of Java wildcards than earlier systems [7]. We have discussed the correspondence between ∃J and Java with wildcards, and highlighted the difficulties associated with adding lower bounds to our calculus. ∃J is a first, significant step towards proving soundness for Java with wildcards.

Acknowledgements We'd like to thank the anonynous reviewers of this paper and a previous, rejected, paper for their many helpful and thorough comments.

References

1. Gilad Bracha. Generics in the Java programming language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.
2. Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Research report 56, DEC Systems Research Center, 1990.
3. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.
4. Giorgio Ghelli and Benjamin Pierce. Bounded existentials and minimal typing. *Theoretical Computer Science*, 193(1-2):75–96, 1998.
5. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition.* Addison-Wesley, Boston, Mass., 2005.
6. 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.
7. 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).
8. 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.
9. 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.

10. 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.
11. Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.
12. Kresten Krab Thorup. Genericity in Java with virtual types. In *ECOOP '97: European Conference on Object-Oriented Programming*, volume 1241, pages 444–471. Springer, 1997.
13. 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.
14. 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.
15. 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.

$$\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{e.f} \rightsquigarrow \texttt{e}'\texttt{.f}}$$

(∃RC-Field)

$$\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{e.<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} \rightsquigarrow \texttt{e}'\texttt{.<}\overline{\texttt{T}}\texttt{>m(}\overline{\texttt{e}}\texttt{)}}$$

(∃RC-Inv-Recv)

$$\frac{\texttt{e}_i \rightsquigarrow \texttt{e}'_i}{\texttt{e.<}\overline{\texttt{T}}\texttt{>m(...e}_i\texttt{...)} \rightsquigarrow \texttt{e.<}\overline{\texttt{T}}\texttt{>m(...e}'_i\texttt{...)}}$$

(∃RC-Inv-Arg)

$$\frac{\texttt{e}_i \rightsquigarrow \texttt{e}'_i}{\texttt{new C<}\overline{\texttt{T}}\texttt{>(...e}_i\texttt{...)} \rightsquigarrow \texttt{new C<}\overline{\texttt{T}}\texttt{>(...e}'_i\texttt{...)}}$$

(∃RC-New-Arg)

$$\frac{\texttt{e}_1 \rightsquigarrow \texttt{e}'_1}{\texttt{open e}_1\texttt{ as x:N in e}_2 \rightsquigarrow \texttt{open e}'_1\texttt{ as x:N in e}_2}$$

(∃RC-Open)

$$\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{close e with }\delta\texttt{ hiding T} \rightsquigarrow \texttt{close e}'\texttt{ with }\delta\texttt{ hiding T}}$$

(∃RC-Close)

Fig. 8. ∃J congruence rules.

A Proof of type soundness

In all these lemmas and theorems we assume that the program is well formed, that is $\vdash$ `class C...` OK for all classes, `C`.

A.1 Progress

Lemma 1 (Canonical Forms). *Take any Δ, Γ and* `v`*. If $\Delta, \Gamma \vdash \mathtt{v} : \mathtt{T}$ then $\mathtt{T} \neq \mathtt{X}$. If $\Delta; \Gamma \vdash \mathtt{v} : \mathtt{N}$, then $\mathtt{v} = \mathtt{new\ N(\overline{v})}$. If $\Delta; \Gamma \vdash \mathtt{v} : \exists\mathtt{X} \lhd \mathtt{U.T}$ then $\mathtt{v} = \mathtt{close\ v'\ with\ X} \lhd \mathtt{U\ hiding\ T'}$ and $\Delta; \Gamma \vdash \mathtt{v'} : \mathtt{U'}$ and $\Delta \vdash \mathtt{U'} <: \mathtt{[T'/X]T}$.*

Proof. By inspection of the type rules and syntax of values, with induction on the type rules to prove the second case.

Lemma 2. *For any non-variable type,* `K`*, type environment Δ and types* `T` *and* `U`*. If $\Delta \vdash \mathtt{K} <: \exists\mathtt{X} \lhd \mathtt{U.T}$ then $\mathtt{K} = \exists\mathtt{X} \lhd \mathtt{U'.K'}$.*

Proof. By simple induction over the subtype rules.

Theorem 1 (progress). *For any well-formed expression,* `e` *where $\emptyset; \emptyset \vdash \mathtt{e} : \mathtt{T}$, either there exists* `e'` *where $\mathtt{e} \leadsto \mathtt{e'}$ or* `e` *is a value,* `v`*.*

Proof. By structural induction on the derivation of $\emptyset; \emptyset \vdash \mathtt{e} : \mathtt{T}$ using lemma 1. Since $\Gamma = \emptyset$, we can not apply rules ∃T-This or ∃T-Var. The only interesting case is ∃T-Open, where we note that we only need to apply the inductive hypothesis to the typing judgment of $\mathtt{e}_1$, and thus do not have problems with the environments of the typing judgement of $\mathtt{e}_2$. Also, if $\mathtt{e}_1$ is a value, then it must have the form `close...` *since a subtype of an existential type is an existential type (if it is a non-variable type, which it is, by lemma 1, by lemma 2) and an existentially typed expression must have this form by lemma 1. Furthermore, we know that if the expression is in this form then the two type variables are the same (as required by the reduction rule) by rules ∃T-Open and ∃T-Close.*

Detailed hand-written proofs of some cases can be found at: http://www.doc.ic.ac.uk/~ncameron/existsj/Th1.pdf.

A.2 Preservation

Lemma 3 (weakening).

1. *If $\Delta \vdash \mathtt{T'} <: \mathtt{T}$ then $\Delta, \delta \vdash \mathtt{T'} <: \mathtt{T}$.*
2. *If $\Delta \vdash \mathtt{T}$ OK then $\Delta, \delta \vdash \mathtt{T}$ OK.*
3. *If $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ then $\Delta, \delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$*
4. *If $bound_{\Delta}(\mathtt{T}) = \mathtt{K}$ then $bound_{\Delta,\delta}(\mathtt{T}) = \mathtt{K}$.*

Proof. By structural induction on the derivation of $\Delta \vdash \mathtt{T'} <: \mathtt{T}$, $\Delta \vdash \mathtt{T}$ OK, $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{T}$ and $bound_{\Delta}(\mathtt{T})$ respectively. All cases are trivial or require application of earlier sub-lemmas.

Lemma 4 (Subtyping preserves field types). *For any non-existential types* $\mathtt{R}$ *and* $\mathtt{R}'$ *and environment* Δ *where* $\vdash \Delta$ OK, *if* $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{R})) = (\overline{\mathtt{U}};\overline{\mathtt{f}})$ *and* $\Delta \vdash \mathtt{R}' <: \mathtt{R}$ *then* $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{R}')) = (\overline{\mathtt{U}'};\overline{\mathtt{f}'})$ *and for all* i *such that* $0 \leq i < |\overline{\mathtt{U}}|$: $\mathtt{U}'_i = \mathtt{U}_i$ *and* $\mathtt{f}_i = \mathtt{f}'_i$.

Proof. *By structural induction on the derivation of* $\Delta \vdash \mathtt{R}' <: \mathtt{R}$. *The interesting cases are:*
Case ∃S-Sub-Class**:** *By induction on the* fields *function.*
Case ∃S-bound**:** *By noting that* $\mathit{bound}_\Delta(\Delta(\mathtt{X})) = \mathit{bound}_\Delta(\mathtt{X})$ *by the definition of* $\mathit{bound}()$.

Lemma 5 (Subtyping preserves method type). *For any non-existential types* $\mathtt{R}$ *and* $\mathtt{R}'$ *and method* $\mathtt{m}$ *and environment* Δ *where* $\vdash \Delta$ OK. *If* $\mathit{mType}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{R})) = \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{T}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}$ *and* $\Delta \vdash \mathtt{R}' <: \mathtt{R}$ *then* $\mathit{mType}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{R}')) = \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{T}}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}'$ *and* $\overline{\mathtt{X} \lhd \mathtt{T}} \vdash \mathtt{U}' <: \mathtt{U}$.

Proof. *By structural induction on the derivation of* $\Delta \vdash \mathtt{R}' <: \mathtt{R}$. *The interesting cases are:*
Case ∃S-Sub-Class**:** *By* ∃T-Override.
Case ∃bound**:** *By noting that* $\mathit{bound}_\Delta(\Delta(\mathtt{X})) = \mathit{bound}_\Delta(\mathtt{X})$ *by the definition of* $\mathit{bound}()$.

Lemma 6 (Substitution lemma). *For all* Δ, Δ', $\overline{\mathtt{X}}$, $\overline{\mathtt{T}}$, *where* $\Delta = \Delta_1, \overline{\mathtt{X} \lhd \mathtt{U}}, \Delta_2$ *and none of* $\overline{\mathtt{X}}$ *appear in* Δ_1 *and* $\vdash \Delta$ OK *and* $\Delta' = \Delta_1, [\overline{\mathtt{T}/\mathtt{X}}]\Delta_2$ *and* $\Delta_1 \vdash \mathtt{T} <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}$:

1. *If* $\mathit{bound}_\Delta(\mathtt{T}) = \mathtt{K}$ *then* $\mathit{bound}_{\Delta'}([\overline{\mathtt{T}/\mathtt{X}}]\mathtt{T}) = \mathtt{K}'$ *and* $\Delta' \vdash \mathtt{K}' <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{K}$
2. *If* $\mathit{fields}(\mathit{bound}_\Delta(\mathtt{R})) = (\overline{\mathtt{U}};\overline{\mathtt{f}})$ *then* $\mathit{fields}(\mathit{bound}_{\Delta'}([\overline{\mathtt{T}/\mathtt{X}}]\mathtt{R})) = (\overline{\mathtt{U}'};\overline{\mathtt{f}'})$ *and for all* i *such that* $0 \leq i < |\overline{\mathtt{U}}|$: $\mathtt{U}'_i = [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}_i$ *and* $\mathtt{f}_i = \mathtt{f}'_i$.
3. *For all methods* $\mathtt{m}$, *if* $\Delta_1 \vdash \mathtt{R}$ OK *and* $\mathit{mType}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{R})) = \mathtt{<}\overline{\mathtt{X}' \lhd \mathtt{T}'}\mathtt{>}\overline{\mathtt{U}'} \to \mathtt{U}$ *then* $\mathit{mType}(\mathtt{m}, \mathit{bound}_{\Delta'}([\overline{\mathtt{T}/\mathtt{X}}]\mathtt{R})) = \mathtt{<}\overline{\mathtt{X}' \lhd [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{T}'}\mathtt{>}[\overline{\mathtt{T}/\mathtt{X}}]\overline{\mathtt{U}'} \to \mathtt{U}'$ *and* $\overline{\mathtt{X}' \lhd \mathtt{T}'} \vdash \mathtt{U}' <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}$.
4. *If* $\Delta \vdash \mathtt{U} <: \mathtt{U}'$ *then* $\Delta' \vdash [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U} <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}'$.
5. *If* $\Delta \vdash \mathtt{T}$ OK *then* $\Delta' \vdash [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{T}$ OK.
6. *If* $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{U}$ *then* $\Delta'; [\overline{\mathtt{T}/\mathtt{X}}]\Gamma \vdash [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{e} : \mathtt{U}'$ *and* $\Delta' \vdash \mathtt{U}' <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}$.

Proof. 1. *By structural induction on the derivation of* $\mathit{bound}_\Delta(\mathtt{T})$. *The interesting case is the inductive case where* $\mathtt{T} = \mathtt{Y}$. *There are three sub-cases:* $\mathtt{Y} \in \mathit{dom}(\Delta_1)$ *gives* $[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{Y} = \mathtt{Y}$, *the result is easy by the inductive hypothesis;* $\mathtt{Y} \in \mathit{dom}(\Delta_2)$ *again gives* $[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{Y} = \mathtt{Y}$, *we observe that* $([\overline{\mathtt{T}/\mathtt{X}}]\Delta)(\mathtt{Y}) = [\overline{\mathtt{T}/\mathtt{X}}](\Delta(\mathtt{Y}))$ *and then apply the inductive hypothesis;* $\mathtt{Y} = \mathtt{X}_i$ *gives* $[\overline{\mathtt{T}/\mathtt{X}}]\mathtt{Y} = \mathtt{T}_i$, *by simple induction we have that* $\Delta' \vdash \mathtt{T}_i <: [\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}_i \Rightarrow \Delta' \vdash \mathit{bound}_{\Delta'}(\mathtt{T}_i) <: \mathit{bound}_{\Delta'}([\overline{\mathtt{T}/\mathtt{X}}]\mathtt{U}_i)$.
A detailed hand-written proof can be found at:
http://www.doc.ic.ac.uk/∼ncameron/existsj/Lemma6.pdf.

2. *By lemma 6.1 and noting that since* $fields(bound_{\Delta}(\mathtt{R}))$ *is defined then* $bound_{\Delta}(\mathtt{R}) = \mathtt{N}$ *for some* $\mathtt{N}$*. By easy induction over the subtype rules we have that two class types in a subtype relationship are subclasses. Finally by simple induction over the derivation of* $fields(\mathtt{N})$ *we get the necessary relationship between the reults of applying* $fields$ *and that* $fields(\mathtt{[}\overline{\mathtt{T/X}}\mathtt{]N}) = \mathtt{[}\overline{\mathtt{T/X}}\mathtt{]}fields(\mathtt{N})$*.*
   A detailed hand-written proof can be found at:
   http://www.doc.ic.ac.uk/∼ncameron/existsj/Lemma6.pdf.
3. *By a similar argument to lemma 6.2.*
   A detailed hand-written proof can be found at:
   http://www.doc.ic.ac.uk/∼ncameron/existsj/Lemma6.pdf.
4. *By structural induction on the derivation of* $\Delta \vdash \mathtt{T} <: \mathtt{T}'$*.*
5. *By structural induction on the derivation of* $\Delta \vdash \mathtt{T}$ OK*. A similar argument as in lemmas 6.1 and 6.2 is used for case* ∃F-Var*.*
6. *By structural induction on the derivation of* $\Delta; \Gamma \vdash \mathtt{e} : \mathtt{U}$*. The non-trivial cases are:*
   Case ∃T-Field: *By lemmas 4 and 6.2.*
   Case ∃T-Invk: *By lemma 6.5, inductive hypothesis, lemmas 5 and 6.2 and lemma 6.5.*
   Case ∃T-New: *By lemmas 6.2 and 6.5.*
   Case ∃T-Open: *By lemma 6.5 and 6.4 and transitivity and noting that since* $\mathtt{T}$ *is well-formed, it can not introduce free type variables when substituted into a type.*
   Case ∃T-Close: *By the inductive hypothesis and* ∃S-Full*.*

Lemma 7. *For all* Δ, $\mathtt{R}$, $\mathtt{T}$*, if* $\Delta \vdash \mathtt{T} <: \mathtt{R}$ *then* $\mathtt{T} = \mathtt{R}'$*.*

Proof. *By simple induction on the derivation of* $\Delta \vdash \mathtt{T} <: \mathtt{R}$*.*

Lemma 8 (Term substitution preserves typing). *If* $\Delta; \Gamma, \mathtt{x}:\mathtt{U} \vdash \mathtt{e} : \mathtt{T}$ *and* $\Delta; \Gamma \vdash \mathtt{e}' : \mathtt{U}'$ *and* $\Delta \vdash \mathtt{U}' <: \mathtt{U}$ *where* $\vdash \Delta$ OK *then* $\Delta; \Gamma \vdash \mathtt{[e'/x]e} : \mathtt{T}'$ *and* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$*.*

Proof. *By structural induction on the derivation of* $\Delta; \Gamma, \mathtt{x}:\mathtt{U} \vdash \mathtt{e} : \mathtt{T}$*.*
Case ∃T-Var: $\mathtt{e = y}$*, gives subcases* $\mathtt{x = y}$ *and* $\mathtt{x} \neq \mathtt{y}$*.*
Case ∃T-This: *As case* ∃T-Var*.*
Case ∃T-Field: *By lemmas 7 and 4.*
Case ∃T-Invk: *By lemmas 7, 5 and 6.5.*
Case ∃T-New: *Easy.*
Case ∃T-Open: *Note that due to scoping,* $\mathtt{x}$ *in the lemma is distinguished from* $\mathtt{x}$ *in the open expression. The proof is easy, by applying the inductive hypothesis to each sub-expression and transitivity to the results, the final result is given by reflexivity.*
Case ∃T-Close: *By the inductive hypothesis, transitivity and* ∃S-Full*.*

Lemma 9. *For all* Δ, $\mathtt{C<}\overline{\mathtt{P}}\mathtt{>}$, $\mathtt{m}$*, such that* $\vdash \Delta$ OK, $\Delta \vdash \mathtt{C<}\overline{\mathtt{P}}\mathtt{>}$ OK, $mType(\mathtt{m,\ C<}\overline{\mathtt{P}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{X} \lhd \mathtt{U}'}\mathtt{>}\overline{\mathtt{U}} \to \mathtt{U}$, $mBody(\mathtt{m,\ C<}\overline{\mathtt{P}}\mathtt{>}) = (\overline{\mathtt{x}}; \mathtt{e})$*, there exists* $\mathtt{U}'$ *such that* $\Delta \vdash \mathtt{U}'$ OK *and* $\Delta, \overline{\mathtt{X} \lhd \mathtt{T}}; \overline{\mathtt{x}:\mathtt{U}}, \mathtt{this}:\mathtt{C<}\overline{\mathtt{P}}\mathtt{>} \vdash \mathtt{e} : \mathtt{U}'$ *and* $\Delta, \overline{\mathtt{X} \lhd \mathtt{T}} \vdash \mathtt{U}' <: \mathtt{U}$*.*

Proof. *By induction on the derivation of mBody*($\mathtt{m,\ C<\overline{P}>}$).

Base case: *We first use* T-Class *and* T-Method *for* $\mathtt{m}$ *in* $\mathtt{C}$. *The result is given by applying lemmas 6.6 and 6.4.*

Inductive case: *We use the typing and well-formedness rules for classes, followed by lemmas 3 and 6.5 to get the premises of the inductive hypothesis, applying this and observing that* $\Delta \vdash \mathtt{C<\overline{P}>} <: \overline{\mathtt{[P/Y]}}\mathtt{N}$ *(where* $\mathtt{N}$ *is the direct superclass of* $\mathtt{C}$*) by* ∃T-Sub-Class*, and finally lemma 8 and transitivity, gives the result.*

A detailed hand-written proof can be found at:
http://www.doc.ic.ac.uk/∼ncameron/existsj/Lemma8.pdf.

We define $\Delta \vdash \Gamma$ OK as $\forall \mathtt{x}:\mathtt{T} \in \Gamma : \Delta \vdash \mathtt{T}$ OK.

Theorem 2 (subject reduction). *For any* Δ, Γ *where* $\vdash \Delta$ OK *and* $\Delta \vdash \Gamma$ OK *and any expressions* $\mathtt{e}$ *and* $\mathtt{e}'$ *where* $\mathtt{e} \leadsto \mathtt{e}'$ *and* $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}$ *then* $\Delta;\Gamma \vdash \mathtt{e}' : \mathtt{T}'$ *and* $\Delta \vdash \mathtt{T}' <: \mathtt{T}$.

Proof. *By structural induction on the derivation of* $\mathtt{e} \leadsto \mathtt{e}'$. *The interesting cases are:*

Base case ∃R-Invk: *By application of* ∃T-Invk *and* ∃T-New *to find* $\mathtt{T}$. *We then apply lemmas 9, 3 and 8 and 6.6 to find* $\mathtt{T}'$ *and finally lemmas 3 and 6.4 and transitivity to obtain the subtype relation.*

Base case ∃R-Open-Close: *By first noting that* $\Delta \vdash \exists\mathtt{X}\lhd\ \mathtt{U}.\mathtt{T} <: \exists\mathtt{X}\lhd\ \mathtt{U}'.\mathtt{T}'$ *implies* $\Delta \vdash \mathtt{U} <: \mathtt{U}'$ *and* $\Delta, \mathtt{X}\lhd\ \mathtt{U} \vdash \mathtt{T} <: \mathtt{T}'$*, then applying lemmas 6.4, 6.6, 8, and transitivity and noting that* $\mathtt{X} \notin dom(\Delta)$ *(from well-formedness of* $\Delta, \mathtt{X}\ldots$ *from* ∃T-Open*) and* $\Delta \vdash \Gamma$ OK *implies* $\mathtt{[T/X]}\Gamma = \Gamma$.

Inductive case ∃RC-Field: *By lemmas 4 and 7.*

Inductive case ∃RC-Inv-Recv: *By lemmas 5, 6.4 and 7.*

Inductive case ∃RC-Open: *By the inductive hypothesis and transitivity.*

Inductive case ∃RC-Close: *By the inductive hypothesis and transitivity.*

Detailed hand-written proofs of some cases can be found at:
http://www.doc.ic.ac.uk/∼ncameron/existsj/Th2.pdf.