On Subtyping, Wildcards, and Existential Types

Nicholas Cameron
Victoria University of Wellington
ncameron@ecs.vuw.ac.nz

Sophia Drossopoulou
Imperial College London
scd@doc.ic.ac.uk

ABSTRACT

Wildcards are an often confusing part of the Java type system: the behaviour of wildcard types is not fully specified by subtyping, due to wildcard capture, and the rules for type checking are often misunderstood. Their very formulation seems somehow 'different' from the rest of the Java type system, which is based on a simple, nominal hierarchy.

We investigate subtyping in models for Java with and without generics and wildcards. We separate subclassing from subtyping, unify subtyping for class and wildcard types using existential types, and show that Java wildcards emerge naturally from the combination of inclusion and parametric polymorphism.

1. INTRODUCTION

The original Java type system was mostly straightforward: classes indicate types, a class name stands for objects of that class and its subclasses; therefore, subclasses are also subtypes and, furthermore, inclusion polymorphic. For example, the declaration `class Circle extends Shape {...}` gives that `Circle` is a subtype of `Shape` and that a variable with type `Circle` can be used anywhere that a variable of type `Shape` is expected.

However, the type system of Java 5.0 is less straightforward: parameterised classes take *invariant* type parameters, e.g., `List<Circle>` is *not* a subtype of `List<Shape>`. On the other hand, wildcards introduce variance into the type system, e.g., `List<? extends Circle>` *is* a subtype of `List<? extends Shape>`. As a result, the type system is complex and sometimes surprising, and the behaviour of wildcards seems incongruous with the rest of the system.

Wildcard types in Java are usually modelled using existential types [21, 17, 9, 16]. Existential types straightforwardly give the correct typing and subtyping behaviour for wildcards. Existential types have also been used, in several ways [1, 5, 12, 14, 20, 6], to describe the behaviour of class types in object-oriented languages. In this paper we use existential types to model both class and wildcard types. As a result, we obtain a type system where existential types are an integral and natural part, rather than an afterthought. We distinguish subtyping from polymorphism, the latter is only given by existential types. Therefore, polymorphism (whether arising through subclasses or through wildcards) is treated in a uniform way. By contrast in previous work [9], polymorphism due to subclassing is represented implicitly, whereas polymorphism due to wildcards is represented explicitly using existential types.

We generalise the concept of class names to *brands*; a brand is an intermediate stage in the description of types. A brand does not describe a set of values, but is used to specify types, which do. Brands are defined in terms of subtyping, but we treat polymorphism as a property of types (as opposed to brands). In Java, types and brands (as well as subtyping and polymorphism) are conflated: `Shape` is a type in the variable declaration `Shape x`, but a brand in the parametric type `C<Shape>` or the class declaration `class C<X extends Shape> ...`.

We describe our system in three stages: In Sect. 2, we introduce a system which corresponds to Featherweight Java (FJ) [15], using existential types to model Java types expressed through classes. In Sect. 3, we add type parameters, and thus have a system which corresponds to Featherweight Generic Java (FGJ) [15]. In Sect. 4, we allow a slight generalisation to the syntax of brands, giving a system that corresponds to Java with wildcards.

The contributions of this paper are: a unified model of subtyping in Java using existential types; a proof that our pre-generics model is equivalent with Featherweight Java [15], the de-facto standard model of the Java type system; a discussion of subtyping in Java based on this unified model.

2. JAVA 1.4 TYPES AND SUBTYPES

In Java 1.4 (i.e., Java without generics), types consist of class or interface[1] names. Subtype polymorphism (also called inclusion polymorphism) follows directly from subclassing. Subtyping in Java is *nominal*, as opposed to *structural*: subtyping follows from the declared relation between classes, rather than from the structure of objects.

Existential types have been used to model inclusion polymorphism in foundational models of object-oriented languages [1, 5, 12, 14, 20]. These calculi generally model objects as records of data and functions and use structural subtyping. An alternative approach for nominal type systems is to assume that class types are *exact* (i.e., subclassing does not

[1]We do not consider interfaces in this paper.

FTfJP '09, July 6 2009, Genova, Italy

lead to polymorphism) [6] and use existential types to restore polymorphism. Class types in Java can be thought of as such existential types [16, 7]. For example, the type `@Shape` (an exact `Shape`) denotes only objects which instantiate `Shape`; the type $\exists$`X<: Shape.X` (we use `<:` to denote an upper bound on a type variable) denotes objects with an unknown type which is a subclass of `Shape`, this includes instantiations of `Shape` and `Circle`, and is equivalent to the Java type `Shape`. We follow this second approach.

2.1 $\exists$FJ

In this section we present $\exists$FJ, a small model for Java, similar in scope and style to Featherweight Java (FJ) [15]. In $\exists$FJ, class names and types are not conflated, and the distinction between subclassing and subtyping is made explicit.

`e`	::= `x` $\mid$ `e.f` $\mid$ `e.m(`$\overline{\texttt{e}}$`)` $\mid$ `new C(`$\overline{\texttt{e}}$`)`	*expressions*
`v`	::= `new C(`$\overline{\texttt{v}}$`)`	*values*
`Q`	::= `class C` $\lhd$ `N {`$\overline{\texttt{T f;}}$ $\overline{\texttt{M}}$`}`	*class declarations*
`M`	::= `T m(`$\overline{\texttt{T x}}$`) {return e;}`	*method declarations*
`N`	::= `C`	*class names*
`P`	::= `N` $\mid$ `X`	*brands*
`T`	::= $\exists\Sigma.\mathcal{X}$	*source types*
Δ	::= $\overline{\texttt{X<: P}}$	*type environments*
Σ	::= $\overline{\mathcal{X}\texttt{<: P}}$	*binding type environments*
Γ	::= $\overline{\texttt{x:T}}$	*environments*
`A`	::= `P` $\mid$ `T` $\mid$ $\mathcal{X}$	*general types*
`x`		*variables*
`C, D`		*classes*
`X, Y`		*free type variables*
$\mathcal{X}$, $\mathcal{Y}$		*bound type variables*

Figure 1: Syntax of $\exists$FJ.

The syntax of $\exists$FJ is given in Fig. 1. Expressions and class declarations follow FJ (we use the shorthand $\lhd$ for `extends` in Java); the differences are in the syntax of types. We distinguish brands (`P`) from source types (`T`, which are existential types bounded by brands; we call these source types since they are the only types which correspond to Java types, and thus may appear in a source program) and between subclassing (given by brand subtyping) and subtype polymorphism (given by existential subtyping)[2]. General types (`A`) may be either brands or source types.

Our motivation is to have a nominal equivalent of the distinction between records for the implementation of objects and existential types for their interface types, found in older existential types models for objects [1, 5, 12, 14, 20]. As in systems with exact types, subtyping is in the formalism (in our case to compare the bounds of existential types), but does not directly lead to polymorphism: only existential types are polymorphic and thus polymorphism is made explicit in the types.

[2]That our actual subtype relation is not strictly divided is a convenience of the formalisation: it allows for easier extension to the wildcards system and would allow for a simple extension with exact types.

As a convenience, we differentiate between bound and free type variables in the syntax: free type variables use an upper-case, type-writer font (`X`), (existentially) bound variables use an upper-case, curly font ($\mathcal{X}$)[3]. When an existential type is unpacked, bound variables must be replaced with free variables. Distinguishing between bound and free variables allows us to syntactically avoid type variables being used as source types.

$$\frac{\texttt{X} \in \Delta}{\Delta \vdash \texttt{X}\ \text{OK}}\ \text{(F-Var)} \qquad \frac{\texttt{class C} \lhd \texttt{D \{...\}}}{\Delta \vdash \texttt{C}\ \text{OK}}\ \text{(F-Class)}$$

$$\frac{\Delta \vdash \overline{\texttt{X<: [}\overline{\texttt{X}/\mathcal{X}}\texttt{]P}}\ \text{OK} \qquad \Delta, \overline{\texttt{X<: [}\overline{\texttt{X}/\mathcal{X}}\texttt{]P}} \vdash \texttt{[}\overline{\texttt{X}/\mathcal{X}}\texttt{]}\mathcal{X}\ \text{OK}}{\Delta \vdash \exists\overline{\mathcal{X}\texttt{<: P}}.\mathcal{X}\ \text{OK}}\ \text{(F-Exist)}$$

Figure 2: $\exists$FJ well-formed types.

$$\frac{}{\Delta \vdash \texttt{A} <: \texttt{A}}\ \text{(S-Reflex)} \qquad \frac{\Delta \vdash \texttt{A} <: \texttt{A}'' \quad \Delta \vdash \texttt{A}'' <: \texttt{A}'}{\Delta \vdash \texttt{A} <: \texttt{A}'}\ \text{(S-Trans)}$$

$$\frac{}{\Delta \vdash \texttt{X} <: \Delta(\texttt{X})}\ \text{(S-Bound)} \qquad \frac{\texttt{class C} \lhd \texttt{N \{...\}}}{\Delta \vdash \texttt{C} <: \texttt{N}}\ \text{(S-Sub-Class)}$$

$$\frac{\Delta, \overline{\texttt{X}\lhd\ \texttt{[}\overline{\texttt{X}/\mathcal{X}}\texttt{]P}} \vdash \texttt{[}\overline{\texttt{X}/\mathcal{X}}\texttt{]A} <: \texttt{A}' \qquad \overline{\texttt{X}} \cap fv(\texttt{A}') = \emptyset}{\Delta \vdash \exists\overline{\mathcal{X}\texttt{<: P}}.\texttt{A} <: \texttt{A}'}\ \text{(S-Open)}$$

$$\frac{\Delta \vdash \texttt{A} <: \overline{\texttt{[}\overline{\texttt{A}/\mathcal{X}}\texttt{]P}} \qquad fv(\overline{\texttt{A}}) \subseteq dom(\Delta)}{\Delta \vdash \texttt{[}\overline{\texttt{A}/\mathcal{X}}\texttt{]A} <: \exists\overline{\mathcal{X}\texttt{<: P}}.\texttt{A}}\ \text{(S-Close)}$$

Figure 3: $\exists$FJ subtyping.

Rules for well-formed types (which have the shape $\Delta \vdash$ `A` OK) are given in Fig. 2 and are unsurprising. Subtyping ($\Delta \vdash$ `A` $<:$ `A`) is defined in Fig. 3. Rules S-Close and S-Open introduce and eliminate existential types, respectively. They are almost identical to those found in Pizza [19], and $\mathcal{EX}_{impl}$ and $\mathcal{EX}_{upto}$ [22]. Type variables may only appear in a subtype derivation if introduced by a subtyping rule because they cannot be written by the programmer as source types. We omit a well-formed enviornment judegment ($\Delta \vdash \Delta$ OK) as it is complex to define in a sound manner [9], we only assume that it gives the standard properties concerning free variables.

Rules for type checking expressions ($\Delta; \Gamma \vdash$ `e` : `A`) are given in Fig. 4. Most are standard. Receivers of field accesses and method calls must have types which are class names; since these are not source types, these must be found by existential elimination, subsumption, and subtyping.

We are only interested in typing derivations which conclude by assigning source types to expressions. However,

[3]In a well-formed type, 'free' variables are in fact bound in an environment, Δ.

$$\frac{\Delta \vdash \texttt{C}\ \text{OK} \qquad \mathit{fields}(\texttt{C}) = \overline{\texttt{f}} \qquad \mathit{fType}(\texttt{f}, \texttt{C}) = \texttt{T} \qquad \Delta; \Gamma \vdash \overline{\texttt{e} : \texttt{T}}}{\Delta; \Gamma \vdash \texttt{new C}(\overline{\texttt{e}}) : \exists \mathcal{X}\texttt{<:C}.\mathcal{X}}\ \text{(T-New)}$$

$$\frac{}{\Delta; \Gamma \vdash \texttt{x} : \Gamma(\texttt{x})}\ \text{(T-Var)} \qquad \frac{\Delta; \Gamma \vdash \texttt{e} : \texttt{N} \qquad \mathit{fType}(\texttt{f}, \texttt{N}) = \texttt{T}}{\Delta; \Gamma \vdash \texttt{e.f} : \texttt{T}}\ \text{(T-Field)}$$

$$\frac{\Delta; \Gamma \vdash \texttt{e} : \texttt{N} \qquad \mathit{mType}(\texttt{m}, \texttt{N}) = \overline{\texttt{T}} \to \texttt{T} \qquad \Delta; \Gamma \vdash \overline{\texttt{e} : \texttt{T}}}{\Delta; \Gamma \vdash \texttt{e.m}(\overline{\texttt{e}}) : \texttt{T}}\ \text{(T-Invk)} \qquad \frac{\Delta, \Delta'; \Gamma \vdash \texttt{e} : \texttt{A}' \qquad \Delta, \Delta' \vdash \texttt{A}' <: \texttt{A} \qquad \Delta \vdash \texttt{A}\ \text{OK}}{\Delta; \Gamma \vdash \texttt{e} : \texttt{A}}\ \text{(T-Subs)}$$

$$\frac{\Delta; \Gamma \vdash \texttt{e} : \exists \overline{\mathcal{X}\texttt{<:P}}.\mathcal{X}}{\Delta, \overline{\texttt{X<:}\ [\overline{\texttt{X}/\mathcal{X}}]\texttt{P}}; \Gamma \vdash \texttt{e} : [\overline{\texttt{X}/\mathcal{X}}]\mathcal{X}}\ \text{(T-Elim)}$$

Figure 4: ∃FJ typing rules.

intermediate steps in the derivation may assign brands to expressions.

Following Tame FJ [9], T-Subs allows type variables which are not used in the type of the conclusion to be removed from the type environment on the left of the turnstile. In conjunction with S-Close, this allows existential types to be introduced, and unpacked type variables to be removed from the environment. For example, if `x` has type `X` (bounded by `C`), `x` can be packed to $\exists\mathcal{X}\texttt{<:C}.\mathcal{X}$ and `X` can be removed from the typing environment. T-Elim allows existential types to be eliminated in typing derivations.

We have elided field and method lookup functions, operational semantics, and rules for type checking methods and classes. These follow FJ and can be found in appendix A.

2.2 Equivalence of ∃FJ and FJ

We have proved that ∃FJ is equivalent to FJ [15]; that is, a program will type check in ∃FJ if and only if it type checks in FJ. More formally:

> *Theorem* For all $\overline{\texttt{x}}$, $\overline{\texttt{C}}$, `e`, and `D`,
> $\emptyset; \overline{\texttt{x:}\exists\mathcal{X}\texttt{<:C}.\mathcal{X}} \vdash \texttt{e} : \exists\mathcal{X}\texttt{<:D}.\mathcal{X}$ in ∃FJ,
> if and only if $\overline{\texttt{x:C}} \vdash \texttt{e} : \texttt{D}$ in FJ.

We prove this theorem by defining a more general translation of ∃FJ types into FJ types (type variables are translated to their bounds, and class names in FJ to class names in ∃FJ) and then proving a more general version of this theorem. Required lemmas are given in appendix C and full proofs can be downloaded from

> http://www.doc.ic.ac.uk/~ncameron/papers/cameron_ftfjp09_proofs.pdf

3. JAVA WITH GENERICS

In Java, parametric polymorphism is implemented by *generics* [2, 3, 13]. Classes, types, and methods may be parameterised by formal type parameters; actual type parameters are provided when a class is instantiated or a method is called. For example, a list class is parameterised by the type of items in that list, this type variable can then be used in the body of the class:

```
class List<X> {
    X get() {...}
    void set(X x) {...}
    List<X> copy {...}
}
```

A list type can be instantiated as a list of strings: `List<String>`; a list of shapes: `List<Shape>`; or a list of any other type. Type checking member access takes into account the actual types; so, for example, the return type of `get` called on an object of type `List<Shape>` will be `Shape`.

The types that may instantiate a formal parameter may be restricted by giving the parameters a bound. This is done using the `extends` keyword. For example, `class C<X extends Shape>...` requires that any type which instantiates `X` is a subtype of `Shape`. Whilst type checking the body of class `C`, we can assume that `X` is a subtype of `Shape`.

3.1 ∃FGJ

$$\begin{array}{lll}
\texttt{e} ::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.<}\overline{\texttt{P}}\texttt{>m}(\overline{\texttt{e}}) \mid \texttt{new C<}\overline{\texttt{P}}\texttt{>}(\overline{\texttt{e}}) & & \textit{expressions} \\
\texttt{v} ::= \texttt{new C<}\overline{\texttt{P}}\texttt{>}(\overline{\texttt{v}}) & & \textit{values} \\
\\
\texttt{Q} ::= \texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \triangleleft\ \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} & & \textit{class declarations} \\
\texttt{M} ::= \texttt{<}\overline{\texttt{X <: P}}\texttt{>T m(}\overline{\texttt{T x}}\texttt{) \{return e;\}} & & \textit{method declarations} \\
\\
\texttt{N} ::= \texttt{C<}\overline{\texttt{P}}\texttt{>} & & \textit{class names} \\
\texttt{P} ::= \texttt{N} \mid \texttt{X} & & \textit{brands} \\
\texttt{T} ::= \exists\Sigma.\mathcal{X} & & \textit{source types}
\end{array}$$

Figure 5: Syntax of ∃FGJ.

Generics can be easily added to ∃FJ, similarly to adding generics to FJ in FGJ [15]. The existential types parts of the calculus are not affected. The syntax of a language with generics, ∃FGJ, is given in Fig. 5 (we elide the syntax of type environments and syntactic variables). Changes to the syntax and semantics are highlighted.

In ∃FGJ, we extend brands beyond simple class names to parameterised classes and type variables. The system of brand subtyping represents the inheritance relation and its point-wise extension with type parameters. As in ∃FJ, brands do not constitute a source type, nor does brand subtyping directly give subtype polymorphism.

Actual type parameters are brands because a type parameter is a description of a type, not the type of a variable. This is reflected in the invariance of generic types, and is discussed further in Sect. 5. For example, to instantiate a list one would use the brand `List<Dog>` which gives the source type $\exists\mathcal{X}\texttt{<:List<Dog>}.\mathcal{X}$. Note that `List<`$\exists\mathcal{X}\texttt{<:Dog}.\mathcal{X}$`>` is not syntactically valid. Class definitions are encoded by encoding types used in the definition; for example, the list given above is encoded as:

```
class List<X> {
    ∃Z<:X.Z get() {...}
    void set(∃Z<:X.Z x) {...}
    ∃Z<:List<X>.Z copy {...}
}
```

Most type rules do not change. Those that do are modified only to add type parameters, they become the same as in FGJ [15]:

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

$$\frac{\texttt{class C<}\overline{\texttt{X}}\texttt{<:P>}\lhd\ \texttt{N}\ \texttt{\{...\}}}{\Delta \vdash \texttt{C<}\overline{\texttt{P'}}\texttt{>} <: [\overline{\texttt{P'}}/\overline{\texttt{X}}]\texttt{N}}$$
(S-Sub-Class)

$$\frac{\Delta \vdash \texttt{C<}\overline{\texttt{P}}\texttt{>}\ \text{OK} \qquad fields(\texttt{C}) = \overline{\texttt{f}} \qquad fType(\texttt{f}, \texttt{C<}\overline{\texttt{P}}\texttt{>}) = \texttt{T} \qquad \Delta;\Gamma \vdash \overline{\texttt{e : T}}}{\Delta;\Gamma \vdash \texttt{new C<}\overline{\texttt{P}}\texttt{>(}\overline{\texttt{e}}\texttt{)} : \exists\mathcal{X}\texttt{<:C<}\overline{\texttt{P}}\texttt{>}.\mathcal{X}}$$
(T-New)

$$\frac{\Delta;\Gamma \vdash \texttt{e} : \texttt{N} \qquad \Delta;\Gamma \vdash \overline{\texttt{e : T}} \qquad mType(\texttt{m<}\overline{\texttt{P}}\texttt{>}, \texttt{N}) = \texttt{<}\overline{\texttt{X}}\texttt{<:P'>}\overline{\texttt{T}} \to \texttt{T} \qquad \Delta \vdash \overline{\texttt{P}}\ \text{OK} \qquad \Delta \vdash \overline{\texttt{P}} <: \overline{\texttt{P'}}}{\Delta;\Gamma \vdash \texttt{e.<}\overline{\texttt{P}}\texttt{>m(}\overline{\texttt{e}}\texttt{)} : \texttt{T}}$$
(T-Invk)

The auxiliary functions, class and method typing rules, and the operational semantics gain type parameters, but are otherwise boring; they are given in appendix A. No other rules are changed.

4. WILDCARDS

Java wildcards [21, 17] are actual type parameters indicated by '`?`'. They may be given upper or lower bounds using `extends` or `super`. Wildcards facilitate subtype variance: whereas generic types are invariant (e.g., `List<Cirlce>` is not a subtype of `List<Shape>`), wildcard types may be co- or contravariant or both (e.g., `List<? extends Cirlce>` is a subtype of `List<? extends Shape>` and `List<? super Shape>` is a subtype of `List<? super Cirlce>`). The syntax and behaviour of wildcard types are, superficially, unrelated to other Java types.

Wildcard types in Java have been modelled using existential types, in fact this has become the standard approach [17, 9]. In these models, wildcards are represented as existentially quantified type variables; for example, `List<?>` can be represented as $\exists$`X.List<X>`[4]. Bounds on wildcards are represented as bounds on the quantified variable (e.g., `List<? extends Shape>` as $\exists$`X<:Shape.List<X>`).

Variables with existential type are treated as opaque packages and they must be *unpacked* (that is, have quantified type variables replaced with fresh type variables) before use. In Java, this is known as *capture conversion*. For example, to call `get` on `List<? extends Shape>`, the type is capture converted to `List<Z>` where `Z` is fresh and has the upper bound `Shape`. The type of the `get` method is looked up for `List<Z>` and its return type is found to be `Z`. To prevent this variable escaping, the `Z`-free supertype `Shape` is used as the type of the expression.

e	::=	x | e.f | e.<$\overline{\texttt{P}}$>m($\overline{\texttt{e}}$) | new C<$\overline{\texttt{P}}$>($\overline{\texttt{e}}$) *expressions*
v	::=	new C<$\overline{\texttt{P}}$>($\overline{\texttt{v}}$) *values*
Q	::=	class C<$\overline{\texttt{X}}$ <: $\overline{\texttt{P}}$> $\lhd$ N {$\overline{\texttt{T f;}}$ $\overline{\texttt{M}}$} *class declarations*
M	::=	<$\overline{\texttt{X}}$ <: $\overline{\texttt{P}}$> T m($\overline{\texttt{T x}}$) {return e;} *method declarations*
N	::=	C<$\overline{\texttt{P}}$> *class names*
P	::=	$\exists\overline{\Sigma}$.N | X | $\mathcal{X}$ *brands*
T	::=	$\exists\Sigma.\mathcal{X}$ *source types*

Figure 6: Syntax of ∃WFJ.

4.1 ∃WFJ

We formalise Java with wildcards as ∃WFJ, a minor generalisation of ∃FGJ. In ∃WFJ, existential quantification fulfils two roles: enabling inclusion polymorphism (quantification at the type level) and variance of parametric types (quantification at the brand level). Both cases are governed by the same rules.

The syntax of ∃WFJ is given in Fig. 6. The only changes from the syntax of ∃FGJ (Fig. 5) are a natural generalisation of the syntax of types: brands may also be existentials or bound variables. This allows encoded wildcard types to be used as bounds on quantified variables in types (e.g., `C<?>` encoded as the type $\exists\mathcal{X}$`<:(`$\exists\mathcal{Y}$`.C<`$\mathcal{Y}$`>).`$\mathcal{X}$) and brands (e.g, `C<? extends D<?>>`, encoded as the brand $\exists\mathcal{X}$`<:(`$\exists\mathcal{Y}$`.D<`$\mathcal{Y}$`>).C<`$\mathcal{X}$`>`), and for deeply nested wildcard types (e.g., `C<C<?>>`, encoded as the brand `C<`$\exists\mathcal{X}$`.C<`$\mathcal{X}$`>>`).

The only changes to the type system follow this change: F-Exist and T-Elim are generalised following the generalisation of brands. This very small and natural generalisation to ∃FGJ gives a type system which supports Java wildcards without any additional work:

$$\frac{\Delta \vdash \overline{\texttt{X<:}[\overline{\texttt{X}}/\overline{\mathcal{X}}]\texttt{P}}\ \text{OK} \qquad \Delta, \overline{\texttt{X<:}[\overline{\texttt{X}}/\overline{\mathcal{X}}]\texttt{P}} \vdash [\overline{\texttt{X}}/\overline{\mathcal{X}}]\texttt{A}\ \text{OK}}{\Delta \vdash \exists\overline{\mathcal{X}\texttt{<:P}}.\texttt{A}\ \text{OK}}$$
(F-Exist)

$$\frac{\Delta;\Gamma \vdash \texttt{e} : \exists\overline{\mathcal{X}\texttt{<:P}}.\texttt{A}}{\Delta, \overline{\texttt{X<:}[\overline{\texttt{X}}/\overline{\mathcal{X}}]\texttt{P}};\Gamma \vdash \texttt{e} : [\overline{\texttt{X}}/\overline{\mathcal{X}}]\texttt{A}}$$
(T-Elim)

Type checking with wildcard types follows type checking with class types, discussed in Sect. 2.1. In particular, field or method lookup is unchanged, and for a receiver that is assigned an unquantified brand type, the use of type rules T-New, T-Field, or T-Invk are identical. An encoded wildcard type involves two levels of quantification; therefore, it must be unpacked twice (using T-Elim). It is also possible that the result type of T-Field or T-Invk will have to be re-packed to avoid escaping type variables. For example, when calling the copy method on a list with type `List<?>`, encoded as $\exists\mathcal{X}$`<:(`$\exists\mathcal{Y}$`.List<`$\mathcal{Y}$`>).`$\mathcal{X}$, method lookup will be performed on the brand `List<Z>`, where `Z` is fresh. The result of method lookup will be $\exists\mathcal{X}$`<:(List<Z>).`$\mathcal{X}$, which will be re-packed to $\exists\mathcal{X}$`<:(`$\exists\mathcal{Y}$`.List<`$\mathcal{Y}$`>).`$\mathcal{X}$ by T-Subs and the subtyping rules.

[4]We omit bounds for clarity, we could use `Object` as the upper bound in this case.

Restrictions. ∃WFJ is not a full model for Java with wildcards: we do not support lower bounds or type parameter inference at method calls, and we do not define well-formed type environments. Lower bounds can easily be added as in Tame FJ, they are elided to simplify the formalisation. Type inference of method parameters is discussed in Sect. 5.2. The intuition of well-formed environments is that all types in the range of the environment are well-formed. However, this is complicated to define due to the presence of F-bounds [11]. We have done this in Tame FJ [9], but this requires stratification of subtyping, which we have avoided for clarity.

Properties. We expect that ∃WFJ corresponds with (a subset of) Tame FJ [9], in the same way as ∃FJ with FJ [15]. However, we have not yet attempted to prove this and suspect an intermediate form of ∃WFJ, with a formalisation more similar to Tame FJ, will be required. The correspondence can be stated formally:

> *Conjecture* For all Δ, $\overline{\mathtt{x}}$, $\overline{\mathtt{P}}$, $\mathtt{e}$, and $\mathtt{P}$,
> $\Delta; \overline{\mathtt{x}:\exists\mathcal{X}\texttt{<:}\mathtt{P}.\mathcal{X}} \vdash \mathtt{e} : \exists\mathcal{X}\texttt{<:}\mathtt{P}.\mathcal{X}$ in ∃WFJ,
> if and only if $\Delta; \overline{\mathtt{x}:\mathtt{P}} \vdash \mathtt{e} : \mathtt{P}|\emptyset$ in (some subset of) Tame FJ.

5. DISCUSSION

In this section we discuss some interesting aspects of Java wildcards in the light of ∃WFJ, and some details of ∃WFJ itself.

5.1 Wildcards

Explaining Subtyping. By making subtype polymorphism explicit using existential quantification, we hope to make understanding generic and wildcard types easier. Programmers may expect generic types to be covariant because class types implicitly involve polymorphism: it could seem strange that `Circle` is a subtype of `Shape`, but `List<Circle>` is not a subtype of `List<Shape>`. By making subtype polymorphism explicit it becomes clear, there is only polymorphism at a given level of nesting if there is quantification at that level: $\exists\mathcal{X}\texttt{<:Circle}.\mathcal{X}$ is a subtype of $\exists\mathcal{X}\texttt{<:Shape}.\mathcal{X}$, indicated by the existential quantification; $\exists\mathcal{X}\texttt{<:List<Circle>}.\mathcal{X}$ is not a subtype of $\exists\mathcal{X}\texttt{<:List<Shape>}.\mathcal{X}$ because the parameters are not quantified. The related wildcard types *are* subtypes: $\exists\mathcal{X}\texttt{<:}(\exists\mathcal{Y}\texttt{<:Circle.List<}\mathcal{Y}\texttt{>)}.\mathcal{X}$ is a subtype of $\exists\mathcal{X}\texttt{<:}(\exists\mathcal{Y}\texttt{<:Shape.List<}\mathcal{Y}\texttt{>)}.\mathcal{X}$, indicated by existential quantification of both the outermost types and the type parameters.

Surface Syntax. We have shown that ∃WFJ emerges naturally from the combination of inclusion and parametric polymorphism. However, ∃WFJ is an existential types based model for Java wildcards, and there are some significant differences between the surface syntax of Java and ∃WFJ. The underlying semantics of the two systems are identical, and so our result is relevant. But existential types are significantly more expressive than wildcard types (there are types in Java which can be expressed but not denoted in Java, this is not the case in ∃WFJ); therefore, it is possible that there is a better surface syntax.

5.2 ∃WFJ

Type Parameter Inference. ∃WFJ does not model the type parameter inference that can take place at method call sites. This is modelled in Tame FJ [9] and other models. Java programs which make use of type parameter inference can be encoded in ∃WFJ (see appendix D) because the receiver's type can be capture converted, even though the types of parameters cannot, due to the lack of type parameter inference.

Type parameter inference has to be added on top of the naturally emerging calculus. However, the fundamental feature of capture conversion does appear for receivers. We view type parameter inference as a useful convenience, additional to the underlying model, which does not introduce any interesting typing features.

Unpacking Existential Types. In our calculi, existential types can be unpacked both in subtyping (using S-Open) and typing (using T-Elim). In ∃FJ and ∃FGJ, only one of these rules is required; however, in ∃WFJ both are required. This is one way in which wildcards do *not* follow simply from the unification of inclusion and parametric polymorphism. We have put both rules into ∃FJ and ∃FGJ because there is no reason to prefer one at the expense of the other. It also makes philosophical sense to have both rules (we should be able to consider subtyping without typing, but unpacking should be a separate operation from subsumption), even if they are not required for expressivity.

Type Soundness. ∃WFJ is not formalised with type soundness in mind; in its present form, the soundness proofs for Tame FJ [9] cannot easily be extended to ∃WFJ. We believe ∃WFJ is sound, but that proving this directly would require some technical changes to the calculus. We would need to stratify subtyping as in Tame FJ and introduce a normal form for existential types which would require changes to the type rules, such as adding guarding environments, as in Tame FJ.

Packing and Unpacking in Subtyping. We chose to use separate rules to pack and unpack types during subtyping (S-Close and S-Open in Fig. 3). We believe that this separation of concerns is clearer than combining both operations into a single rule (S-Env), as in Wild FJ [17] and Tame FJ [9]. These two approaches are equivalent, as shown in the first author's thesis [8]. The S-Env rule in our calculi would be:

$$\frac{\overline{\mathtt{Y}} \cap fv(\exists\overline{\mathtt{X}\texttt{<:}\mathtt{P}}.\mathtt{A}) = \emptyset \qquad fv(\overline{\mathtt{A}}) \subseteq dom(\Delta), \overline{\mathtt{Y}} \qquad \Delta, \overline{\mathtt{Y}\texttt{<:}\,[\overline{\mathtt{Y}/\mathcal{Y}}]\mathtt{P}'} \vdash \overline{[\mathtt{Y}/\mathcal{Y}]\mathtt{A}} \texttt{<:} \overline{[\mathtt{A}/\mathcal{X}]\mathtt{P}}}{\Delta \vdash \exists\overline{\mathcal{Y}\texttt{<:}\,\mathtt{P}'}.[\overline{\mathtt{A}/\mathcal{X}}]\mathtt{A} \texttt{<:} \exists\overline{\mathcal{X}\texttt{<:}\,\mathtt{P}}.\mathtt{A}}$$

6. RELATED WORK

Existential types have been used in many systems to model object-oriented types [1, 5, 12, 14, 20]. The main difference between these approaches and ∃WFJ is that we use nominal, rather than structural types. Also, we wish to model subtyping properties rather than encapsulation properties and so we abstract the interface of an object rather than its implementation. In both ways we are more similar to the existential types proposed to model hash types and non-exact types [6, 7, **?**, 16, 4]. It would be easy to add exact types to ∃WFJ, we would extend the syntax of types with `@P`, and extend subtyping with the axiom $\Delta \vdash \mathtt{@P} \texttt{<:} \mathtt{P}$.

Existential types have been used to explain [21] and model [17, 9, 10, 22] wildcard types in Java; our treatment of wild-

card types follow these earlier formalisations. Our innovation is to also use existential types to model subtype polymorphism.

The language Unity [18] supports a mixture of nominal and structural subtyping in order to provide a strong and flexible type system. As in ∃WFJ, a brand forms part of a type, not a type in its own right, and subclassing does not directly give subtyping. However, the relation of brands to types, and subclassing to subtyping are very different.

7. CONCLUSION AND FUTURE WORK

In this paper we have modelled the behaviour, in particular subtyping and polymorphism, of Java types in a unified way using existential types. We have discussed several interesting aspects of this approach and shown that, for class types at least, our approach is equivalent to the standard model for Java.

Future Work. We wish to show type soundness directly for ∃WFJ, and prove the conjecture in Sect. 4.1. In general, we are interested in investigating further properties of existential types and subtyping in object-oriented languages.

Acknowledgement. We would like to thank Alex Summers for suggesting the distinction between bound and free variables, and for many interesting and useful discussions about wildcards, and the anonymous reviewers for their close reading and useful comments. The first author's work was funded in part by a Build IT Postodoctoral fellowship; the second author was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

8. REFERENCES

[1] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An Interpretation of Objects and Object Types. In *Principles of Programming Languages (POPL)*, 1996.

[2] Gilad Bracha. Generics in the Java Programming Language, 2004. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1998.

[4] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. *Lecture Notes in Computer Science*, 3086:389–413, 2004.

[5] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing Object Encodings. *Information and Computation*, 155(1-2), 1999.

[6] Kim B. Bruce, Martin Odersky, and Philip Wadler. A Statically Safe Alternative to Virtual Types. In *European Conference on Object Oriented Programming (ECOOP)*, 1998.

[7] Kim B. Bruce, Leaf Petersen, and Adrian Fiech. Subtyping Is Not a Good "Match" for Object-Oriented Languages. In *European Conference on Object Oriented Programming (ECOOP)*, 1997.

[8] Nicholas Cameron. *Existential Types for Variance — Java Wildcards and Ownership Types*. PhD thesis, Imperial College London, 2009.

[9] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A Model for Java with Wildcards. In *European Conference on Object Oriented Programming (ECOOP)*, 2008.

[10] Nicholas Cameron, Erik Ernst, and Sophia Drossopoulou. Towards an Existential Types Model for Java Wildcards. In *Formal Techniques for Java-like Programs (FTfJP)*, 2007.

[11] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-Bounded Quantification for Object-Oriented Programming. In *Functional Programming Languages and Computer Architecture (FPCA)*, 1989.

[12] Kathleen Fisher and John C. Mitchell. Notes on Typed Object-Oriented Programming. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 1994.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.

[14] Martin Hofmann and Benjamin C. Pierce. A Unifying Type-Theoretic Framework for Objects. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 1994.

[15] 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.

[16] Atsushi Igarashi and Mirko Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *Transactions on Programming Languages and Systems*, 28(5):795–847, 2006. An earlier version appeared as "On variance-based subtyping for parametric types" at European Conference on Object Oriented Programming (ECOOP) 2002.

[17] Mads Torgersen and Erik Ernst and Christian Plesner Hansen. Wild FJ. In *Foundations of Object-Oriented Languages (FOOL)*, 2005.

[18] Donna Malayeri and Jonathan Aldrich. Integrating Nominal and Structural Subtyping. In *European Conference on Object Oriented Programming (ECOOP)*, 2008.

[19] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Principles of Programming. Languages (POPL)*, 1997.

[20] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[21] 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.

[22] Stefan Wehr and Peter Thiemann. Subtyping Existential Types. In *Formal Techniques for Java-like Programs (FTfJP)*, 2008.

APPENDIX

A. ELIDED DEFINITIONS

In this section we define the auxiliary functions, method and class typing rules, and operational semantics for ∃FJ and ∃FGJ.

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

$$\frac{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\}}{fType(\texttt{f}_i, \texttt{C}) = \texttt{T}_i} \qquad \frac{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{f} \notin \overline{\texttt{f}}}{fType(\texttt{f}, \texttt{C}) = fType(\texttt{f}, \texttt{N})}$$

$$\frac{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C}) = mBody(\texttt{m}, \texttt{N})} \qquad \frac{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{U m}\,(\overline{\texttt{U x}})\ \{\texttt{return e}_0\texttt{;}\} \in \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}; \texttt{e}_0)}$$

$$\frac{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C}) = mType(\texttt{m}, \texttt{N})} \qquad \frac{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{U m}\,(\overline{\texttt{U x}})\ \{\texttt{return e}_0\texttt{;}\} \in \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C}) = \overline{\texttt{U}} \to \texttt{U}}$$

Figure 7: Auxillary functions for ∃FJ.

$$\frac{\texttt{class C} \lhd \texttt{N}\ \{\ldots\} \quad \emptyset \vdash \texttt{T}, \overline{\texttt{T}}\ \text{OK} \quad \emptyset; \overline{\texttt{x:T}}, \texttt{this:}\exists\mathcal{X}\texttt{<:C}.\mathcal{X} \vdash \texttt{e:T} \quad override(\texttt{m}, \texttt{N}, \overline{\texttt{T}} \to \texttt{T})}{\vdash \texttt{T m(}\overline{\texttt{T x}}\texttt{)\ \{return e\}}\ \text{OK in C}}\ \text{(T-Method)}$$

$$\frac{mType(\texttt{m}, \texttt{N}) = \overline{\texttt{T}} \to \texttt{T}}{override(\texttt{m}, \texttt{N}, \overline{\texttt{T}} \to \texttt{T})}\ \text{(T-Override)} \qquad \frac{mType(\texttt{m}, \texttt{N})\quad undefined}{override(\texttt{m}, \texttt{N}, \overline{\texttt{T}} \to \texttt{T})}\ \text{(T-OverrideUndef)}$$

$$\frac{\emptyset \vdash \texttt{N}, \overline{\texttt{T}}\ \text{OK} \qquad \vdash \overline{\texttt{M}}\ \text{OK in C}}{\texttt{class C} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\}}\ \text{(T-Class)}$$

Figure 8: ∃FJ method and class typing rules.

$$\frac{fields(\texttt{C}) = \overline{\texttt{f}}}{\texttt{new C(}\overline{\texttt{v}}\texttt{).f}_i \rightsquigarrow \texttt{v}_i}\ \text{(R-Field)} \qquad \frac{\texttt{v = new C(}\overline{\texttt{v}'}\texttt{)} \quad mBody(\texttt{m}, \texttt{C}) = (\overline{\texttt{x}}; \texttt{e}_0)}{\texttt{v.m(}\overline{\texttt{v}}\texttt{)} \rightsquigarrow \texttt{[}\overline{\texttt{v/x}}\texttt{, v/this]e}_0}\ \text{(R-Invk)}$$

$$\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{e.f} \rightsquigarrow \texttt{e}'\texttt{.f}}\ \text{(RC-Field)} \qquad \frac{\texttt{e}_i \rightsquigarrow \texttt{e}_i'}{\texttt{new C(...e}_i\texttt{...)} \rightsquigarrow \texttt{new C(...e}_i'\texttt{...)}}\ \text{(RC-New-Arg)}$$

$$\frac{\texttt{e} \rightsquigarrow \texttt{e}'}{\texttt{e.m(}\overline{\texttt{e}}\texttt{)} \rightsquigarrow \texttt{e}'\texttt{.m(}\overline{\texttt{e}}\texttt{)}}\ \text{(RC-Inv-Recv)} \qquad \frac{\texttt{e}_i \rightsquigarrow \texttt{e}_i'}{\texttt{e.m(...e}_i\texttt{...)} \rightsquigarrow \texttt{e.m(...e}_i'\texttt{...)}}\ \text{(RC-Inv-Arg)}$$

Figure 9: ∃FJ reduction rules.

$$\frac{}{fields(\texttt{Object<>}) = \emptyset}$$

$$\frac{\texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \lhd \texttt{D<}\overline{\texttt{P}'}\texttt{>}\{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad fields(\texttt{D}) = \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{g}}, \overline{\texttt{f}}}$$

$$\frac{\texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\}}{fType(\texttt{f}_i, \texttt{C<}\overline{\texttt{P}'}\texttt{>}) = \texttt{[}\overline{\texttt{P}'\texttt{/X}}\texttt{]T}_i}$$

$$\frac{\texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{f} \notin \overline{\texttt{f}}}{fType(\texttt{f}, \texttt{C<}\overline{\texttt{P}'}\texttt{>}) = fType(\texttt{f}, \texttt{[}\overline{\texttt{P}'\texttt{/X}}\texttt{]N})}$$

$$\frac{\texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mBody(\texttt{m}, \texttt{C<}\overline{\texttt{P}'}\texttt{>}) = mBody(\texttt{m}, \texttt{[}\overline{\texttt{P}'\texttt{/X}}\texttt{]N})}$$

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

$$\frac{\texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\} \quad \texttt{m} \notin \overline{\texttt{M}}}{mType(\texttt{m}, \texttt{C<}\overline{\texttt{P}'}\texttt{>}) = mType(\texttt{m}, \texttt{[}\overline{\texttt{P}'\texttt{/X}}\texttt{]N})}$$

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

Figure 10: Auxillary functions for ∃GJ.

$$\frac{\Delta' = \texttt{<}\overline{\texttt{Y <: P}}\texttt{>} \quad \texttt{class C<}\overline{\texttt{X}\ldots}\texttt{>} \lhd \texttt{N}\ \{\ldots\} \quad \Delta \vdash \Delta'\ \text{OK} \quad \Delta, \Delta' \vdash \texttt{T}, \overline{\texttt{T}}\ \text{OK} \quad \Delta, \Delta'; \overline{\texttt{x:T}}, \exists\mathcal{X}\texttt{<:C<}\overline{\texttt{X}}\texttt{>}.\mathcal{X} \vdash \texttt{e:T} \quad override(\texttt{m<}\overline{\texttt{Y}}\texttt{>}, \texttt{N}, \texttt{<}\overline{\texttt{Y <: P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T})}{\Delta \vdash \texttt{<}\overline{\texttt{Y <: P}}\texttt{>T m(}\overline{\texttt{T x}}\texttt{)\ \{return e}\}\ \text{OK in C}}\ \text{(T-Method)}$$

$$\frac{mType(\texttt{m<}\overline{\texttt{X}}\texttt{>}, \texttt{N}) = \texttt{<}\overline{\texttt{X <: P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T}}{override(\texttt{m<}\overline{\texttt{X}}\texttt{>}, \texttt{N}, \texttt{<}\overline{\texttt{X <: P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T})}\ \text{(T-Override)}$$

$$\frac{mType(\texttt{m<}\overline{\texttt{X}}\texttt{>}, \texttt{N}) \quad undefined}{override(\texttt{m<}\overline{\texttt{X}}\texttt{>}, \texttt{N}, \texttt{<}\overline{\texttt{X <: P}}\texttt{>}\overline{\texttt{T}} \to \texttt{T})}\ \text{(T-OverrideUndef)}$$

$$\frac{\Delta = \overline{\texttt{X <: P}} \quad \emptyset \vdash \Delta\ \text{OK} \quad \Delta \vdash \texttt{N}, \overline{\texttt{T}}\ \text{OK} \quad \Delta \vdash \overline{\texttt{M}}\ \text{OK in C}}{\texttt{class C<}\overline{\texttt{X <: P}}\texttt{>} \lhd \texttt{N}\ \{\overline{\texttt{T f;}}\ \overline{\texttt{M}}\}}\ \text{(T-Class)}$$

Figure 11: ∃FGJ method and class typing rules.

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}}{\mathtt{new\ C<\overline{P}>(\overline{v}).f}_i \rightsquigarrow \mathtt{v}_i}$$
(R-Field)

$$\frac{\mathtt{v = new\ N(}\overline{\mathtt{v'}}\mathtt{)} \qquad \mathit{mBody}(\mathtt{m<\overline{P}>}, \mathtt{N}) = (\overline{\mathtt{x}}; \mathtt{e}_0)}{\mathtt{v.<\overline{P}>m(\overline{v})} \rightsquigarrow \mathtt{[\overline{v/x}, v/this]e}_0}$$
(R-Invk)

$$\frac{\mathtt{e} \rightsquigarrow \mathtt{e'}}{\mathtt{e.f} \rightsquigarrow \mathtt{e'.f}}$$
(RC-Field)

$$\frac{\mathtt{e}_i \rightsquigarrow \mathtt{e}'_i}{\mathtt{new\ C<\overline{P}>(...e}_i\mathtt{...)} \rightsquigarrow \mathtt{new\ C<\overline{P}>(...e}'_i\mathtt{...)}}$$
(RC-New-Arg)

$$\frac{\mathtt{e} \rightsquigarrow \mathtt{e'}}{\mathtt{e.<\overline{P}>m(\overline{e})} \rightsquigarrow \mathtt{e'.<\overline{P}>m(\overline{e})}}$$
(RC-Inv-Recv)

$$\frac{\mathtt{e}_i \rightsquigarrow \mathtt{e}'_i}{\mathtt{e.<\overline{P}>m(...e}_i\mathtt{...)} \rightsquigarrow \mathtt{e.<\overline{P}>m(...e}'_i\mathtt{...)}}$$
(RC-Inv-Arg)

Figure 12: ∃FGJ reduction rules.

B. FEATHERWEIGHT JAVA

In this section we define Featherweight Java (FJ) [15] using the notation used elsewhere in this paper. FJ is defined in figures 13 – 18. The main differences between these definitions and the definition in [15] is that we factor out a subsumption rule in the type rules, use an *fType* function, and use slightly different notation throughout; for example, we use $\overline{\mathtt{T}\,\mathtt{f};}$, rather than $\overline{\mathtt{T}}\,\overline{\mathtt{f}};$. We also elide casts.

$\mathtt{e}$	::= $\mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}})$	*expressions*
$\mathtt{Q}$	::= $\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\}$	*class declarations*
$\mathtt{M}$	::= $\mathtt{T\,m}(\overline{\mathtt{T}\,\mathtt{x}})\ \mathtt{\{return\ e;\}}$	*method declarations*
$\mathtt{v}$	::= $\mathtt{new\ C}(\overline{\mathtt{v}})$	*values*
$\mathtt{N}$	::= $\mathtt{C}$	*class names*
$\mathtt{T}$	::= $\mathtt{N}$	*types*
Γ	::= $\overline{\mathtt{x}:\mathtt{T}}$	*environments*
$\mathtt{x}$		*variables*
$\mathtt{C}, \mathtt{D}$		*classes*

Figure 13: Syntax of FJ.

$$\frac{}{fields(\mathtt{Object}) = \emptyset} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\} \quad fields(\mathtt{D}) = \overline{\mathtt{g}}}{fields(\mathtt{C}) = \overline{\mathtt{g}}, \overline{\mathtt{f}}}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\}}{fType(\mathtt{f}_i, \mathtt{C}) = \mathtt{T}_i} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\} \quad \mathtt{f} \notin \overline{\mathtt{f}}}{fType(\mathtt{f}, \mathtt{C}) = fType(\mathtt{f}, \mathtt{N})}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\} \quad \mathtt{m} \notin \overline{\mathtt{M}}}{mBody(\mathtt{m}, \mathtt{C}) = mBody(\mathtt{m}, \mathtt{N})} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\} \quad \mathtt{U\,m}(\overline{\mathtt{U}\,\mathtt{x}})\ \mathtt{\{return\ e_0;\}} \in \overline{\mathtt{M}}}{mBody(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}; \mathtt{e}_0)}$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\} \quad \mathtt{m} \notin \overline{\mathtt{M}}}{mType(\mathtt{m}, \mathtt{C}) = mType(\mathtt{m}, \mathtt{N})} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\} \quad \mathtt{U\,m}(\overline{\mathtt{U}\,\mathtt{x}})\ \mathtt{\{return\ e_0;\}} \in \overline{\mathtt{M}}}{mType(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{U}} \to \mathtt{U}}$$

Figure 14: Auxillary functions for FJ.

$$\frac{\Delta \vdash \mathtt{T} <: \mathtt{T}'' \qquad \Delta \vdash \mathtt{T}'' <: \mathtt{T}'}{\Delta \vdash \mathtt{T} <: \mathtt{T}'} \text{ (S-Trans)}$$

$$\frac{}{\Delta \vdash \mathtt{T} <: \mathtt{T}} \text{ (S-Reflex)} \qquad \frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\ldots\}}{\Delta \vdash \mathtt{C} <: \mathtt{N}} \text{ (S-Sub-Class)}$$

Figure 15: FJ subtyping.

$$\frac{\mathtt{class\ C} \lhd \mathtt{D}\ \{\ldots\}}{\Delta \vdash \mathtt{C}\ \text{OK}} \text{ (F-Class)}$$

Figure 16: FJ well-formed types.

$$\frac{\Delta \vdash \mathtt{C}\ \text{OK} \qquad fields(\mathtt{C}) = \overline{\mathtt{f}} \quad fType(\mathtt{f}, \mathtt{C}) = \mathtt{D} \qquad \Delta; \Gamma \vdash \overline{\mathtt{e} : \mathtt{D}}}{\Delta; \Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}} \text{ (T-New)}$$

$$\frac{}{\Delta; \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})} \text{ (T-Var)} \qquad \frac{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{C} \quad fType(\mathtt{f}, \mathtt{C}) = \mathtt{D}}{\Delta; \Gamma \vdash \mathtt{e.f} : \mathtt{D}} \text{ (T-Field)}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{C} \quad mType(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \to \mathtt{D} \quad \Delta; \Gamma \vdash \overline{\mathtt{e} : \mathtt{D}}}{\Delta; \Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{D}} \text{ (T-Invk)} \qquad \frac{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{C}' \quad \Delta \vdash \mathtt{C}' <: \mathtt{C} \quad \Delta \vdash \mathtt{C}\ \text{OK}}{\Delta; \Gamma \vdash \mathtt{e} : \mathtt{C}} \text{ (T-Subs)}$$

Figure 17: FJ typing rules.

$$\frac{\mathtt{class\ C} \lhd \mathtt{N}\ \{\ldots\} \quad \emptyset \vdash \mathtt{T}, \overline{\mathtt{T}}\ \text{OK} \qquad \overline{\mathtt{x}:\mathtt{T}}, \mathtt{this}:\mathtt{C} \vdash \mathtt{e}:\mathtt{T} \quad override(\mathtt{m}, \mathtt{N}, \overline{\mathtt{T}} \to \mathtt{T})}{\vdash \mathtt{T\ m}(\overline{\mathtt{T}\,\mathtt{x}})\ \mathtt{\{return\ e\}}\ \text{OK in C}} \text{ (T-Method)}$$

$$\frac{mType(\mathtt{m}, \mathtt{N}) = \overline{\mathtt{T}} \to \mathtt{T}}{override(\mathtt{m}, \mathtt{N}, \overline{\mathtt{T}} \to \mathtt{T})} \text{ (T-Override)} \qquad \frac{mType(\mathtt{m}, \mathtt{N}) \quad undefined}{override(\mathtt{m}, \mathtt{N}, \overline{\mathtt{T}} \to \mathtt{T})} \text{ (T-OverrideUndef)}$$

$$\frac{\emptyset \vdash \mathtt{N}, \overline{\mathtt{T}}\ \text{OK} \qquad \vdash \overline{\mathtt{M}}\ \text{OK in C}}{\mathtt{class\ C} \lhd \mathtt{N}\ \{\overline{\mathtt{T}\,\mathtt{f};}\ \overline{\mathtt{M}}\}} \text{ (T-Class)}$$

Figure 18: FJ method and class typing rules.

C. TRANSLATION TO FJ

We prove that ∃FJ and FJ [15] are equivalent, that is:

Theorem For all $\overline{\mathtt{x}}$, $\overline{\mathtt{C}}$, $\mathtt{e}$, and $\mathtt{D}$, $\emptyset; \overline{\mathtt{x}:\exists\mathcal{X}<:\mathtt{C}.\mathcal{X}} \vdash \mathtt{e} : \exists\mathcal{X}<:\mathtt{D}.\mathcal{X}$ if and only if $\overline{\mathtt{x}:\mathtt{C}} \vdash \mathtt{e} : \mathtt{D}$.

$$\frac{}{[\![\mathtt{C}]\!]_\Delta = \mathtt{C}}$$

$$\frac{}{[\![\mathtt{X}]\!]_\Delta = [\![\Delta(\mathtt{X})]\!]_\Delta}$$

$$\frac{}{[\![\exists\mathcal{X}<:\mathtt{C}.\mathcal{X}]\!]_\Delta = \mathtt{C}}$$

Figure 19: Translation from ∃FJ to FJ.

$$\frac{}{\Delta \vdash \mathtt{A} <: \mathtt{A}}\ (\text{S-Reflex}) \qquad \frac{\Delta \vdash \Delta(\mathtt{X}) <: \mathtt{A}}{\Delta \vdash \mathtt{X} <: \mathtt{A}}\ (\text{S-Bound})$$

$$\frac{\mathtt{class\ C} \lhd \mathtt{D\ \{...\}} \quad \Delta \vdash \mathtt{D} <: \mathtt{A}}{\Delta \vdash \mathtt{C} <: \mathtt{A}}\ (\text{S-Sub-Class}) \qquad \frac{\Delta, \mathtt{X}<:\mathtt{C} \vdash \mathtt{X} <: \mathtt{A}' \quad \mathtt{X} \notin fv(\mathtt{A}') \quad \Delta \vdash \mathtt{A}' <: \mathtt{A}}{\Delta \vdash \exists\mathcal{X}<:\mathtt{C}.\mathcal{X} <: \mathtt{A}}\ (\text{S-Open})$$

$$\frac{\Delta \vdash \mathtt{A} <: \mathtt{C} \quad fv(\mathtt{A}) \subseteq dom(\Delta) \quad \Delta \vdash \exists\mathcal{X}<:\mathtt{C}.\mathcal{X} <: \mathtt{A}'}{\Delta \vdash \mathtt{A} <: \mathtt{A}'}\ (\text{S-Close})$$

Figure 20: ∃FJ subtyping without transitivity.

The simple translation of types used in the above definition applies only to existential types (the only types that may be written in a ∃FJ program). We generalise this translation to all types ($\mathtt{A}$) in Fig. 19. We also define transitivity-free subtyping for ∃FJ in Fig. 20. The above theorem is then proved by the following lemmas (complete proofs can be downloaded from http://www.doc.ic.ac.uk/~ncameron/papers/cameron_ftfjp09_proofs.pdf):

Lemma For all Δ, Γ, $\mathtt{e}$, and $\mathtt{A}$, if $\Delta;\Gamma \vdash \mathtt{e} : \mathtt{A}$ and $\emptyset \vdash \Delta$ OK then $[\![\Gamma]\!]_\Delta \vdash \mathtt{e} : [\![\mathtt{A}]\!]_\Delta$.

Lemma For all $\mathtt{C}$, if $\vdash \mathtt{C}$ OK in FJ, then $\emptyset \vdash \mathtt{C}$ OK and $\emptyset \vdash \exists\mathcal{X}<:\mathtt{C}.\mathcal{X}$ OK in ∃WFJ.

Lemma For all $\mathtt{C}$, if $\emptyset \vdash \exists\mathcal{X}<:\mathtt{C}.\mathcal{X}$ OK in ∃WFJ then $\vdash \mathtt{C}$ OK in FJ.

Lemma For all $\mathtt{C}$, if $\Delta \vdash \mathtt{A}$ OK and $\emptyset \vdash \Delta$ OKin ∃WFJ then $\vdash [\![\mathtt{A}]\!]_\Delta$ OK in FJ.

Lemma For all $\overline{\mathtt{C}}$ and $\mathtt{D}$, if $\vdash \mathtt{C} <: \mathtt{D}$ in FJ, then $\emptyset \vdash \exists\mathcal{X}<:\mathtt{C}.\mathcal{X} <: \exists\mathcal{X}<:\mathtt{D}.\mathcal{X}$ in ∃WFJ.

Lemma For all $\overline{\mathtt{C}}$ and $\mathtt{D}$, if $\emptyset \vdash \exists\mathcal{X}<:\mathtt{C}.\mathcal{X} <: \exists\mathcal{X}<:\mathtt{D}.\mathcal{X}$ in ∃WFJ, then $\vdash \mathtt{C} <: \mathtt{D}$ in FJ.

Lemma For all Δ, $\mathtt{A}_1$, $\mathtt{A}_2$, $\mathtt{A}_3$, if $\Delta \vdash \mathtt{A}_1 <: \mathtt{A}_2$ and $\Delta \vdash \mathtt{A}_2 <: \mathtt{A}_3$ then $\Delta \vdash \mathtt{A}_1 <: \mathtt{A}_3$ in ∃FJ without transitivity.

Lemma For all Δ, $\mathtt{A}_1$, $\mathtt{A}_2$, if $\Delta \vdash \mathtt{A}_1 <: \mathtt{A}_2$ in ∃FJ then $\Delta \vdash \mathtt{A}_1 <: \mathtt{A}_2$ in ∃FJ without transitivity.

Lemma For all Δ, $\mathtt{A}_1$, $\mathtt{A}_2$, if $\Delta \vdash \mathtt{A}_1 <: \mathtt{A}_2$ in ∃FJ without transitivity then $\vdash [\![\mathtt{A}_1]\!]_\Delta <: [\![\mathtt{A}_2]\!]_\Delta$ in FJ.

D. TYPE PARAMETER INFERENCE AND WILDCARD CAPTURE

As discussed in Sect. 5.2, we do not support type parameter inference and wildcard capture of parameters in ∃WFJ. A Java program which uses this feature can be encoded in ∃WFJ since ∃WFJ supports wildcard capture of receivers. For example, consider the following methods:

```
class C<X> {...}

class D<> {
    <X>C<X> m1(C<X> x) {...}

    C<?> m2(C<?> y) {
        return this.m1(y);
    }
}
```

This can be encoded as

```
class C<Y> {
    ...
    C<Y> m3(D<> r) {
        return r.m3(this);
    }
}

class D<> {
    <X>C<X> m1(C<X> x) {...}

    C<Y> m2(C<?> y) {
        return y.m3(this);
    }
}
```

Multiple parameters can be encoded by adding multiple methods to classes, each method encodes the capture conversion of one parameter.