Type Inference for Java 5
Wildcards, F-Bounds, and Undecidability

Karl Mazurak and Steve Zdancewic

University of Pennsylvania
{mazurak,stevez}@cis.upenn.edu

Abstract. We consider the problem of type checking for Java 5 with generics and wildcard types. Java supports type inference for methods with type parameters, but its presentation in the Java Language Specification lacks both clarity and a sense for how inference fits into the larger picture of Java's type system. We show, in contrast, how this type inference can be cleanly integrated with subtyping of wildcard types, justifying the intuitions of programmers and logicians alike. In addition, we examine the decidability of Java typechecking and identify key points in the subtyping relation that both lead to undecidability and suggest a decidable conservative approximation.

1 Introduction

Java 5 improves on previous versions of Java by introducing many advanced type-system features including *generic types* (sometimes called parametric polymorphic types) and *wildcard types*. Generics are useful for defining parameterized classes; for example, given the generic definition `class List<X> {...}`, the program can create instances `List<String>` and `List<Integer>`. The type variable `X` can be mentioned in the signatures of `List<X>` methods, which is useful for ensuring that all the elements of a `List<String>` object are `String`s. In contrast, a field (or method argument) with wildcard type `List<?>` is known to store an object of type `List<C>` for some unknown class `C`. While subtyping of generic classes is naturally *invariant* with respect to type parameters—`List<Object>` is not a supertype of `List<String>`—the type `List<?>` *is* a supertype of `List<C>` for all `C`. Wildcard types thus help smooth the integration of subtype polymorphism with generics.

Unfortunately, the addition of generics and wildcards to Java significantly complicates the typechecking process. Java supports F-bounded subtyping, which means that type parameters to generic classes may be given bounds that mention the parameter itself: for example, `class C<X extends D<X>> {...}` is a legal Java 5 class declaration. Wildcard types complicate matters even further, because implementing them requires inference to determine what concrete types should instantiate occurrences of the wildcard '`?`'. For example, when checking a method call `o.m(e)` where the method `m` has type signature `void m(List<?> l)` and the expression `e` has time `List<T>`, the typechecker must infer that the

wildcard should be instantiated with the type `T`. In this simple example, such inference is easy, but in the presence of parameterized types with bounds, the problem quickly becomes more difficult. As we show, both the type inference and type checking problems for Java 5 are undecidable in general.

Given this result, complete Java 5 typechecking can be at best implemented as a semi-algorithm (which may not terminate on all programs). Pragmatically, it seems best to forgo completeness and instead implement a sound algorithm that conservatively rejects some well-typed programs. It is not clear how best to make this compromise. Sun's current stable release implementation of Java 5 and the Java 6 snapshots diverge (or exhaust stack space) on both well-typed and ill-typed inputs, in addition to conservatively rejecting some well-typed inputs. However, exactly how their implementation is intended to behave is difficult to determine—the definitive resource for understanding Java 5 typechecking, the Java Language Specification (JLS) [1], is complex, very long, and, especially with respect to the advanced type features of Java, rather reader unfriendly.

This situation is bad for programmers, because in order to use these new features one often has to reason about their interaction with the typechecker. It is also bad for researchers and compiler writers, because implementing a Java 5-compliant typechecker is very difficult—in fact, the results in this paper came about from our own attempts to implement a Java 5-compliant frontend for research purposes and our inability to fully understand the type inference and typechecking "algorithms" specified in the JLS.

This paper aims to improve this situation by explaining Java 5 type inference and typechecking. Specifically, we make the following contributions:

- We introduce ∃FJ, a clean, declarative formalism for reasoning about Java 5 generics and wildcards. We give two variants: a "kernel" version that has restricted inference for F-bounded subtyping, and a "full" version that corresponds more closely to Java 5.
- We prove type soundness for the inference process by (1) showing soundness of ∃FJ without type inference and (2) showing that inferred type arguments always correspond to well-typed explicit arguments.
- We argue that inference and subtyping—closely linked in our presentation—are decidable for the simpler kernel ∃FJ, and that the same are undecidable for full ∃FJ, and hence also for Java 5.
- In light of these results, we examine algorithmic issues with respect to implementing both inference and typechecking, and suggest a way to easily specify a nice class of incomplete (but decidable) typechecking algorithms for full Java 5.

Our formalism is based on prior work described by Torgersen et al. in their "Wild FJ" paper [2]. Here we adopt their use of existential types as a convenient intermediate language for use in typechecking. We diverge from their approach in our more rigorous approach to type inference; out intent is to provide a clean formalism that gives insight into the connections, implied by both programming practice and logical interpretations, among type argument inference, subtyping,

and wildcards. In addition, we provide a proof of type soundness and investigate the question of decidability; to our knowledge neither of these has been previously attempted in this context.

Our language (∃FJ) is described in Section 2. That section also describes our approach to inference for the kernel and full versions of the language and establishes a soundness result. Section 3 considers the algorithmic issues of typechecking for ∃FJ and shows that in kernel ∃FJ typechecking is decidable while in the full version it is not. Section 4 discusses the practical ramifications of these results, illustrates the (potentially undesirable) behavior of Sun's Java 5 implementation, and suggests a uniform way of cutting back the full ∃FJ type inference and checking rules to obtain an algorithm suitable for use in practice. This section also discusses possible refinements to the type system that make more precise inference possible. Section 5 concludes.

2 Existential Featherweight Java

To explain Java's type inference and demonstrate its connection with wildcard types, we present Existential Featherweight Java, a Java core calculus and direct extension of Featherweight Java and Featherweight Generic Java [3]. We present two prospective definitions of subtyping, called *kernel* and *full* subtyping after similar—in spirit and in consequence—variations on the language $F_{<:}$ [4].

Inspired by the Wild FJ formalism [2], we treat wildcards in Java 5 as a constrained form of existential types where each wildcard placeholder `?` is replaced by a distinct existentially bound type variable. This allows us to express strictly more types than would be possible using only wildcards; for example, the Java 5 type `List<?>` can instead be written as the existential type $\exists \bot \leqslant \texttt{X} \leqslant \texttt{Object}.\ \texttt{List<X>}$, while the slightly more complex existential type $\exists \bot \leqslant \texttt{X} \leqslant \texttt{Object}.\ \texttt{Pair<X,X>}$ cannot be written with wildcards, as Java provides no means of referring to a previous wildcard parameter. The Java 5 type `Pair<?,?>` may look similar, but this type means and is equivalent in our calculus to $\exists \bot \leqslant \texttt{X} \leqslant \texttt{Object}, \bot \leqslant \texttt{Y} \leqslant \texttt{Object}.\ \texttt{Pair<X,Y>}$. These new types, however, do not adversely affect our results; they lead, in fact, to simpler typing and subtyping rules, and our conclusions on decidability in Section 3 hold even considering only those types corresponding to legal Java 5 code.

∃FJ is defined in Figure 1, where the symbol $\lhd$ is shorthand for the word `extends`. Other than the type system, described in the next section, it is identical to Featherweight Generic Java [3], which added generics (but neither wildcards nor type inference) to Featherweight Java, intended to be the smallest possible subset of Java to which we can associate an interesting type system. It features only classes, not interfaces, and each method has as its body a single return statement. Expressions e include only method argument variables, object creation, field access, method invocation, and coercion via typecast. Additionally, such complicating features as exceptions, field shadowing, and method overloading are not present. (Method override, however, *is* allowed.)

Syntax

ρ, σ, τ	::=	$\exists \Delta.T$	*existential types*
S, T, U, V	::=	$N \mid X$	*bare types*
L, M, N	::=	$\texttt{Object} \mid C\texttt{<}\overline{T}\texttt{>} \mid \bot$	*class types*
cd	::=	$\texttt{class}\ C\texttt{<}\overline{X} \lhd \overline{\exists \Delta^{\exists}.N}\texttt{>} \lhd \exists \Delta^{\exists}.N\ \{\ \overline{\tau}\ \overline{f}\texttt{;}\ \overline{md}\ \}$	*class declarations*
md	::=	$\texttt{<}\overline{X} \lhd \overline{\exists \Delta^{\exists}.N}\texttt{>}\ \tau\ m\texttt{(}\overline{\tau}\ \overline{x}\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\ \}$	*method declarations*
e	::=	$x \mid \texttt{new}\ N\texttt{(}\overline{e}\texttt{)} \mid e\texttt{.}f \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid e\texttt{.<}\overline{\tau}\texttt{>}m\texttt{(}\overline{e}\texttt{)} \mid \texttt{(}\tau\texttt{)}e$	*expressions*
Γ	::=	$\emptyset \mid \Gamma, x : \tau$	*typing contexts*
Δ	::=	$\emptyset \mid \Delta, L \leqslant X \leqslant M$	*type environments*
θ	::=	$\emptyset \mid \theta, X \mapsto T$	*type substitutions*

Metavariables

CT	*class tables*	X, Y, Z	*type variables*	f	*field names*
C, D	*class names*	x, y, z	*expression variables*	m	*method names*

Fig. 1. Existential Featherweight Java

As we are concerned primarily with typing and not with operational semantics, we make the additional simplification of eliding constructors; each class is assumed to have one constructor, taking arguments appropriate for its superclass followed by as many arguments as it has fields, and assigning the latter in order to its own fields after passing the former to the constructor of its superclass. We also make liberal use of overbar notation whenever it introduces no ambiguity or loss of information, so that our rules may be kept concise; for example, a class type $C\texttt{<}\overline{T}\texttt{>}$ should be read as $C\texttt{<}T_1, \ldots, T_k\texttt{>}$, while the field declarations $\overline{\tau}\ \overline{f}\texttt{;}$ denote $\tau_1\ f_1\texttt{;}\ \ \ldots\ \tau_k\ f_k\texttt{;}$. Finally, we make the common assumption that variables are either distinct or can be silently renamed to avoid conflicts, and thus we do not worry about variable capture and similar issues.

An ∃FJ program is defined as a class table CT together with a single expression analogous to the body of a "main" method. Values or normal forms of the calculus are defined to be `new` expressions whose arguments are also values; this is as close as our small-step semantics comes to anything resembling objects on the heap. Evaluation proceeds in a call-by-value fashion; the receiver, if any, is evaluated to a normal form, followed by the arguments. Field accesses evaluate to the appropriate field of the receiver, while method invocations evaluate to the method body with formal parameters replaced by actual parameters. Valid typecasts have no runtime behavior; invalid typecasts cause the entire program to evaluate to an error state, which is the only terminal configuration apart from a value. Formal evaluation rules for ∃FJ can be found in our forthcoming technical report.

2.1 Type system

As seen in Figure 1, types τ in $\exists$FJ take the form $\exists\Delta.\ T$, where Δ is a type environment—a list of type variables with both lower and upper bounds. We write $T_1 \leqslant \texttt{X} \leqslant T_2$ to indicate that type variable `X` has lower bound T_1 and upper bound T_2. Like full Java, we allow the type variables of classes and methods to appear in their own upper bounds and in the bounds of simultaneously defined variables, and, since type environments also arise from the upper bounds of those variables, we allow this sort of self-reference (traditionally called F-boundedness [5]) in existentially bound variables as well. We do not, however, allow variables to appear in their own lower bounds; not only is this impossible in full Java, as only upper bounds can be specified for named type variables, but it would needlessly complicate many of our rules.

Bare types T are either type variables X or class types N, where the types created by classes defined in the class table CT are augmented by both `Object`—which, unlike full Java's `Object`, has no fields or methods—and the null type $\bot$. We may occasionally write C instead of $C\texttt{<>}$, just as we may write T in place of $\exists\emptyset.\ T$ or $L \leqslant X \leqslant M$ in place of $\emptyset, L \leqslant X \leqslant M$, as long as it is unambiguous to do so; we write the concatenation of two contexts Δ and Δ' as $\Delta\Delta'$. $\bot$ is a subtype of all other types and, although it cannot be written, is the type of `null` in Java 5; here it allows us to write lower bounds for all type variables. Of course, to correspond to a legal Java program, $\bot$ must appear *only* in lower bounds.

In Wild FJ [2], Torgersen, Ernst, and Plesner Hansen demonstrated that the transformation from wildcards to existential types is a relatively simple syntactic operation that could conceivably be done as a preprocessing step; Kim Bruce has independently been developing a similar translation [6]. As we are interested less in wildcards themselves and more in clarifying the type inference process, we adopt existential types as our primitive construct. The additional types allowed by both existential variables and the ability to specify upper and lower bounds simultaneously do not add any complexity to our rules, and, since type inference is essentially the same problem as determining an existential witnesses, it is more natural to start with existential types rather than to introduce them later on.

Existential types in other contexts [7, 8] generally involve a *pack* operation, which produces a value of type $\exists\alpha.\ \tau$ by abstracting some part of τ by the type variable α, as well as an *unpack* operation allowing α to be used within some scope with no assumptions about the type originally abstracted by α. With bounded existentials, which constrain type variables in the same manner as in bounded universal quantification [9], these hidden types are partially revealed by exposing their upper bounds—and, in the case of Java, their lower bounds. When used in the style of Java wildcards, however, neither pack nor unpack is present. Instead, the existentially bound variables implied by a wildcard can be inferred as type arguments to polymorphic methods—the JLS refers to this as *capture conversion*—while types can be implicitly packed to existentials via the subtyping relation. It is our goal to show that the latter process, this discovery

[T-App]

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : \exists\Delta_0.\ T_0 \qquad \Delta;\Gamma \vdash e_1 : \sigma_1 \quad \ldots \quad \Delta;\Gamma \vdash e_n : \sigma_n \\ mtype_{\Delta\Delta_0}(m, T_0) = \texttt{<}\overline{X} \lhd \exists\overline{\Delta'}.\ \overline{N}\texttt{>}\ \exists\Delta_1''.\ T_1'', \ldots, \exists\Delta_n''.T_n \to \exists\Delta'.\ T' \\ \Delta\Delta_0\overline{\Delta'} \vdash \sigma_1 <: \exists\Delta_1''\overline{\Delta}.\ [\overline{X} \mapsto \overline{T}]T_1'' \quad \ldots \quad \Delta\Delta_0\overline{\Delta'} \vdash \sigma_n <: \exists\Delta_n''\overline{\Delta}.\ [\overline{X} \mapsto \overline{T}]T_n'' \\ \Delta\Delta_0 \vdash \exists\overline{\Delta}.\ \overline{T} <: \exists\overline{\Delta'}.\ \overline{N}\end{array}}{\Delta;\Gamma \vdash e\texttt{.<}\exists\overline{\Delta}.\ \overline{T}\texttt{>}m\texttt{(}e_1, \ldots, e_n\texttt{)} : \exists\Delta'\overline{\Delta}[\overline{X} \mapsto \overline{T}]T'}$$

[T-InferApp]

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : \exists\Delta_0.\ T_0 \qquad \Delta;\Gamma \vdash e_1 : \sigma_1 \quad \ldots \quad \Delta;\Gamma \vdash e_n : \sigma_n \\ mtype_{\Delta\Delta_0}(m, T_0) = \texttt{<}\overline{X} \lhd \exists\overline{\Delta'}.\overline{N}\texttt{>}\ \tau_1, \ldots, \tau_n \to \exists\Delta'.\ T' \qquad \Delta^{\forall}{}_0 = \bot \leqslant \overline{X} \leqslant \overline{N} \\ \Delta^{\forall}{}_0; \Delta\Delta_0\overline{\Delta'}; \emptyset \vdash \sigma_1 <: \tau_1; \Delta^{\forall}{}_1; \theta_1 \quad \ldots \quad \Delta^{\forall}{}_0; \Delta\Delta_0\overline{\Delta'}; \theta_{n-1} \vdash \sigma_n <: \tau_n; \Delta^{\forall}{}_n; \theta_n \\ (\overline{\Delta}, \overline{T}) = infer_{\Delta\Delta_0\overline{\Delta'}}(\theta_n(\overline{X}), \Delta^{\forall}{}_1 \ldots \Delta^{\forall}{}_n)\end{array}}{\Delta;\Gamma \vdash e\texttt{.}m\texttt{(}e_1, \ldots, e_n\texttt{)} : \exists\Delta'\overline{\Delta}[\overline{X} \mapsto \overline{T}]T'}$$

Fig. 2. Typing: $\Delta;\Gamma \vdash e : \tau$

of witnesses for existential variables, is in fact equivalent to the inference of type arguments to methods.

Most of the typing rules for ∃FJ are taken directly from FJ. To reduce clutter, our rules omit well-formedness checks on type environments, defined for later reference in Figure 3. The complete set of rules for both typing and well-formedness can be found in Appendix A, and the full definitions of such helper functions as *mtype*—denoting the lookup of a method type in the class table, just as in FJ—can be found in our forthcoming technical report. The complexity of the system, however, shows itself in the rules for method invocation, which can be seen in Figure 2. Unlike Wild FJ [2], we limit ourselves to method invocation as allowed by Java 5; either all type arguments are specified or all type arguments are omitted. In the latter case the inferred type arguments may be instantiated to existential type variables that would otherwise be considered out of scope; this is what the JLS calls capture conversion.

Both method invocation rules, T-App and T-InferApp, begin by checking the types of the method receiver and method arguments, then looking up the method signature. After this, however, T-App's task is relatively simple; the types of the actual parameters are checked against the types of the formal parameters after substituting the supplied type arguments, while said type arguments are checked against their bounds.

T-InferApp, by contrast, begins with this second step, but approaches the subtype relation in a more nuanced way; the subtyping judgments begin with an additional type environment $\Delta^{\forall}{}_0$, so named because we intend to infer bindings for the method's universally quantified type variables. Although we leave precise definitions of this more involved form of subtyping for the next section, the general intuition is that additional constraints on type variables are accumulated in the $\Delta^{\forall}{}_i$ environments on the right-hand side of the relation while substitutions

$$[\text{WF-Env}]\ \frac{\begin{array}{c}\Delta' = L_1 \leqslant X_1 \leqslant M_1, \ldots, L_n \leqslant X_n \leqslant M_n \\ \Delta \vdash L_1\ \mathsf{ok} \quad \ldots \quad \Delta \vdash L_n\ \mathsf{ok} \qquad \Delta\Delta' \vdash M_1\ \mathsf{ok} \quad \ldots \quad \Delta\Delta' \vdash M_n\ \mathsf{ok} \\ \Delta'' = \mathit{floorless}(\Delta') \qquad \Delta\Delta'' \vdash L_1 <: M_1 \quad \ldots \quad \Delta\Delta'' \vdash L_n <: M_n\end{array}}{\Delta \vdash \Delta'\ \mathsf{ok}}$$

$$[\text{WF-SubstAll}]\ \frac{\Delta \vdash X_1 \mapsto T_1\ \mathsf{ok} \quad \ldots \quad \Delta \vdash X_n \mapsto T_n\ \mathsf{ok}}{\Delta \vdash X_1 \mapsto T_1, \ldots, X_n \mapsto T_n\ \mathsf{ok}}$$

$$[\text{WF-SubstOne}]\ \frac{\Delta = \Delta', L \leqslant X \leqslant M, \Delta'' \qquad \Delta \vdash L <: T \qquad \Delta \vdash T <: M}{\Delta \vdash X \mapsto T\ \mathsf{ok}}$$

Fig. 3. Well-formedness

Lower bound erasure

$$[\text{F-Empty}]\ \frac{}{\mathit{floorless}(\emptyset) = \emptyset}$$

$$[\text{F-Env}]\ \frac{\mathit{floorless}(\Delta) = \Delta'}{\mathit{floorless}(\Delta, L \leqslant X \leqslant M) = \Delta', \bot \leqslant X \leqslant M}$$

Substitution extension

$$[\text{SE-Id}]\ \frac{}{\{X \mapsto T\}\emptyset = X \mapsto T}$$

$$[\text{SE-Disjoint}]\ \frac{X \neq X' \qquad \theta' = \{X \mapsto T\}\theta}{\{X \mapsto T\}\theta, X' \mapsto T' = \theta', X' \mapsto [X \mapsto T]T'}$$

Environment restriction

$$[\text{R-Env}]\ \frac{\Delta' = \overline{L} \leqslant \overline{X} \leqslant \overline{M}}{\Delta\lfloor_{\Delta'} = \Delta\lfloor_{\overline{X}}} \qquad [\text{R-Empty}]\ \frac{}{\emptyset\lfloor_X = \emptyset}$$

$$[\text{R-No}]\ \frac{X \neq Y}{\Delta, L \leqslant Y \leqslant M\lfloor_X = \Delta\lfloor_X} \qquad [\text{R-Yes}]\ \frac{L \leqslant X \leqslant M \notin \Delta\lfloor_X}{\Delta, L \leqslant X \leqslant M\lfloor_X = \Delta\lfloor_X, L \leqslant X \leqslant M}$$

Fig. 4. Helper functions

$$\text{[I-Exact]}\ \frac{}{\mathit{infer}_\Delta(N, \Delta^\forall) = (\emptyset, N)}$$

$$\text{[I-Default]}\ \frac{\Delta^\forall\lfloor_X = \bot \leqslant X \leqslant M \qquad X \notin FV(M)}{\mathit{infer}_\Delta(X, \Delta^\forall) = (\emptyset, M)}$$

$$\text{[I-Bounded]}\ \frac{\Delta^\forall\lfloor_X = \overline{L} \leqslant X \leqslant M \qquad \Delta\Delta^\forall \vdash \bigsqcup \overline{L} = \exists\Delta.T \qquad T \neq \bot}{\mathit{infer}_\Delta(X, \Delta^\forall) = (\Delta, T)}$$

$$\text{[I-All]}\ \frac{\mathit{infer}_\Delta(X_1, \Delta^\forall) = (\Delta^\forall{}_1, T_1) \qquad \ldots \qquad \mathit{infer}_\Delta(X_n, \Delta^\forall) = (\Delta^\forall{}_n, T_n)}{\mathit{infer}_\Delta(X_1, \ldots, X_n, \Delta^\forall) = (\Delta^\forall{}_1, \ldots, \Delta^\forall{}_n, T_1, \ldots, T_n)}$$

Fig. 5. Type inference

θ_i capture those type variables that have been inferred to an exact type. Finally, the accumulated substitution and constrains are passed to the helper function *infer*, defined in Figure 5.

Note, however, that *infer*—discussed later in more detail—does little save reorganize its input; the real work of type inference is handled by the subtype relation itself. We explore this in more detail in the next section and defer a detailed discussion of *infer* until then.

2.2 Kernel Subtyping

We begin with the simpler of our two variations of subtyping. The primary feature distinguishing our definitions from those previously suggested for Java [3, 2] is our focus on the discovery of *witnesses* for existential types. That is, if we desire to show $S <: \exists\bot \leqslant X \leqslant \texttt{Object}.\, T$, what part of S proves that it can indeed instantiate the existentially bound variable X? Traditional systems with existential types have relied on explicit pack constructs mentioned in Section 2.1 for promotion to existential types; Java, however, attempts to accomplish this task automatically.

One might think, however, that this is sounds remarkably similar to the process of inferring type arguments to methods. Indeed, we will show that this is the case; type arguments can be inferred in precisely this manner by keeping track of what witnesses are assigned to each variable and selectively allowing this information to propagate outside its scope. This satisfies our intuition from logic—$\forall x.\ (\varphi \Rightarrow \psi)$ is equivalent to $(\exists x.\ \varphi) \Rightarrow \psi$ as long as the latter remains legally scoped—as well as programmer's intuition that wildcards can often be used instead of methods with type parameters. On the other hand, it may be worrisome to the type theorist who recalls that the reconstruction of explicit type instantiations in the polymorphic λ-calculus (also called System F) is undecid-

$$\text{[PS-Refl]}\ \frac{}{\Delta \vdash T \lessdot T} \qquad \text{[PS-Null]}\ \frac{}{\Delta \vdash \bot \lessdot T}$$

$$\text{[PS-VUpper]}\ \frac{\Delta = \Delta', L \leqslant X \leqslant M, \Delta''}{\Delta \vdash X \lessdot M} \qquad \text{[PS-VLower]}\ \frac{\Delta = \Delta', L \leqslant X \leqslant M, \Delta''}{\Delta \vdash L \lessdot X}$$

$$\text{[PS-Trans]}\ \frac{\Delta \vdash S \lessdot T \qquad \Delta \vdash T \lessdot U}{\Delta \vdash S \lessdot U}$$

$$\text{[PS-VMixed]}\ \frac{\Delta = \Delta', L \leqslant X \leqslant N_1, \Delta'', N_1 \leqslant Y \leqslant M, \Delta''' \qquad \Delta \vdash N_1 \lessdot N_2}{\Delta \vdash X \lessdot Y}$$

$$\text{[PS-Class]}\ \frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{X} \lhd \overline{N}\texttt{>} \lhd N}{\Delta \vdash C\texttt{<}\overline{T}\texttt{>} \lessdot [\overline{X} \mapsto \overline{T}]N}$$

Fig. 6. Pointwise Subtyping: $\Delta \vdash S \lessdot T$

able [10]; by reducing the problem of type inference to subtyping, might we have endangered subtyping's decidability? We address these concerns in Section 3.

For ease of presentation, and to foreshadow our analysis in Section 3, our subtype relation is built on two auxiliary relations. The first, written $\Delta \vdash S \lessdot T$ and defined in Figure 6, denotes the standard pointwise subtyping, analogous to that defined for FGJ [3]. It states that subtyping is reflexive and transitive, that $\bot$ is the bottom of the subtype ordering, and that subtyping does indeed follow the bounds of type variables and the class hierarchy. It makes no reference to existential types, nor does it make distinctions based on which context binds a type variable. The pointwise subtype relation also makes no reference to the other relations, and, indeed, without the separation of type environments that they possess, it could not do so. This design simplifies our presentation, but that is not our only reason for taking this approach, as we will discuss in Section 3. The pointwise subtyping premises in rules EP-Witness and S-Var (discussed shortly) derive from similar concerns.

The second auxiliary relation, existential promotion, is a bit more complex; it is written $\Delta^{\exists}; \Delta; \theta \vdash S \preceq T; \Delta^{\exists'}; \theta'$ and defined in Figure 7. The additional type environment $\Delta^{\exists}$ contains those type variables for which it is legal to bind a witness type, or, equivalently, to infer a type instantiation; those variables in Δ are are assumed to be bound at some outer scope—for example, they might be type variables of a class—and are not available for such binding. The type substitution θ is a mapping from type variables not appearing in either Δ or $\Delta^{\exists}$ to their witnesses.

While Δ remains conceptually unchanged by the existential promotion process, $\Delta^{\exists'}$ and θ' can be thought of as representing modifications made to $\Delta^{\exists}$ and θ, respectively. Variables from $\Delta^{\exists}$ that have been witnessed are no longer

$$\text{[EP-Refl]}\ \frac{}{\Delta^{\exists}; \Delta; \theta \vdash T \preceq T; \Delta^{\exists}; \theta}$$

$$\text{[EP-Class]}\ \frac{\Delta^{\exists}; \Delta; \theta \vdash S_1 \preceq T_1; \Delta^{\exists}{}_1; \theta_1 \quad \ldots \quad \Delta^{\exists}{}_{n-1}; \Delta\theta_{n-1} \vdash S_n \preceq T_n; \Delta^{\exists}{}_n; \theta_n}{\Delta^{\exists}; \Delta\theta \vdash C\texttt{<}S_1, \ldots, S_n\texttt{>} \preceq C\texttt{<}T_1, \ldots, T_n\texttt{>}; \Delta^{\exists}{}_n; \theta_n}$$

$$\text{[EP-Subst1]}\ \frac{\Delta^{\exists}; \Delta; \theta \vdash \theta(X) \preceq T; \Delta^{\exists\prime}; \theta'}{\Delta^{\exists}; \Delta; \theta \vdash X \preceq T; \Delta^{\exists\prime}; \theta'}$$

$$\text{[EP-Subst2]}\ \frac{\Delta^{\exists}; \Delta; \theta \vdash T \preceq \theta(X); \Delta^{\exists\prime}; \theta'}{\Delta^{\exists}; \Delta; \theta \vdash T \preceq X; \Delta^{\exists\prime}; \theta'}$$

$$\text{[EP-Witness]}\ \frac{\begin{array}{cc} X \notin dom(\theta) & \Delta^{\exists} = \Delta^{\exists\prime}, L \leqslant X \leqslant M, \Delta^{\exists\prime\prime} \\ \Delta^{\exists}\Delta \vdash \theta(L) \lessdot \theta(T) & \Delta^{\exists}\Delta \vdash \theta(T) \lessdot (\{X \mapsto \theta(T)\}\theta)(M) \end{array}}{\Delta^{\exists}; \Delta; \theta \vdash T \preceq X; \Delta^{\exists\prime}\Delta^{\exists\prime\prime}; \{X \mapsto \theta(T)\}\theta}$$

Fig. 7. Existential Promotion: $\Delta^{\exists}; \Delta; \theta \vdash S \preceq T; \Delta^{\exists\prime}; \theta'$

present in $\Delta^{\exists\prime}$, and θ' is an extension of θ incorporating these new bindings. Rule EP-Witness performs this transfer of a variable while ensuring that the substitution being accumulated remains idempotent. Other rules ensure that type environments and substitutions are propagated correctly. Note, however, that this relation only equates types that are already in roughly the same "shape"; it refers only to the pointwise subtyping relation when checking the validity of a witness.

Working from these two relations, then, our definition of the subtype relation $\Delta^{\forall}; \Delta; \theta \vdash \sigma <: \tau; \Delta^{\forall\prime}; \theta'$, shown in Figure 8, is relatively simple. The separation of contexts mirrors that of the existential promotion relation, although the use of $\Delta^{\forall}$ instead of $\Delta^{\exists}$ serves to indicate that these additional instantiable type variables originate as method type arguments in need of inference, as opposed to those variables existentially bound by the right-hand type, which also serve as potential binders for witnesses. Intuitively, rule S-Ground promotes a type as far as it can using pointwise subtyping, then attempts to reconcile the existential type variables. Rule S-Simple is mere shorthand—or, technically, an overloading of our relation symbol—allowing us to more concisely state common uses of the subtype relation. Rule S-Var is a bit more complicated, acting very similarly to EP-Witness; instead of extending the substitution with a witness, however, it tightens the variable's lower bound.

Why? Consider finding a witness for the type variable `X` in order to show that `C` $<:$ $\exists \bot \leqslant$ `X` $\leqslant$ `Object`. `X`. `C`, of course, is a valid witness, but so is `Object`, and, in fact, so is any supertype of `C`. Such is not the case when looking at, for example, `A<C>` $<:$ $\exists \bot \leqslant$ `X` $\leqslant$ `Object`. `A<X>`; here `C` is the only possible witness for `X`. It is

$$\text{[S-Ground]}\ \frac{\Delta^\forall \Delta^\exists{}_1 \Delta \vdash \theta(T_1) \lessdot T' \qquad \Delta^\forall \Delta^\exists{}_2; \Delta\Delta^\exists{}_1; \theta \vdash T' \preceq N; \Delta^\exists; \theta'}{\Delta^\forall; \Delta; \theta \vdash \exists \Delta^\exists{}_1.\ T_1 <: \exists \Delta^\exists{}_2.\ N; \Delta^\exists; \theta'}$$

$$\text{[S-Var]}\ \frac{\begin{array}{c}\Delta^\forall = \Delta^{\forall\prime}, L \leqslant X \leqslant M, \Delta^{\forall\prime\prime} \qquad \Delta^\forall \Delta \vdash \theta(L) \lessdot \theta(T) \\ \Delta^\forall \Delta \vdash \theta(T) \lessdot \{X \mapsto T\}\theta(M) \qquad \Delta^{\forall\prime}, T \leqslant X \leqslant M, \Delta^{\forall\prime\prime} \Delta \vdash \theta(T) \lessdot \theta(X)\end{array}}{\Delta^\forall; \Delta; \theta \vdash \exists \Delta^\exists{}_1.\ \theta(T) <: \exists \Delta^\exists{}_2.\ X; \Delta^{\forall\prime}, T \leqslant X \leqslant M, \Delta^{\forall\prime\prime}; \theta}$$

$$\text{[S-Simple]}\ \frac{\emptyset; \Delta; \emptyset \vdash \sigma <: \tau; \Delta^\forall; \theta}{\Delta \vdash \sigma <: \tau}$$

Fig. 8. Kernel Subtyping: $\Delta^\forall; \Delta; \theta \vdash \sigma <: \tau; \Delta^{\forall\prime}; \theta'$

exactly this invariance that motivated the introduction of Java's wildcards, and hence our existential types. All this may not seem important in the context of the examples just given—indeed, the first example cannot even arise in standard Java—but the importance of this distinction becomes clear in the context of type inference, where our goal is to allow as much freedom as possible while still keeping with the constraints imposed by the actual parameters.

We can now begin to understand more precisely the typing rule T-InferApp. Its subtyping judgments take as their additional instantiable variables ($\Delta^\forall$) the method's formal type parameters, and in the process of subtype checking these variables become mapped by the accumulated substitution or are given lower bounds by the refined contexts. Should a variable `X` be mapped to a non-variable type by θ_n, it will be inferred to exactly $\theta(\texttt{X})$ (rule I-Exact). If not, the *infer* function defined in Figure 4 will examine the combined context looking only at bounds on the variable in question—we write such a narrowed context, which may not be well-formed, as $\Delta^\forall \lfloor_X$—and return the least upper bound of the various lower bounds; this must, by definition, be a valid instantiation (rule I-Bounded). Finally, if the lower bound in every case is $\bot$, the upper bound—which must be consistent across occurrences of a single variable, as no subtyping rule modifies upper bounds—will be returned, exactly as specified in the JLS for cases where insufficient type information is provided (rule I-Default). Section 4.2, however, discusses how these rules can easily be modified to allow for more precise type inference.

2.3 Full Subtyping

The rules for kernel subtyping use only the pointwise subtype relation when comparing bounds in the rules PS-VMixed, EP-Witness, and S-Var. Although we will see in Section 3 the utility of this restriction, it still seems unsatisfying, as the use of only a fragment of subtyping in these situations prohibits certain perfectly innocent terms from typechecking.

[PS-VMixed]

$$\frac{\Delta^{\exists}\Delta = \Delta', L \leqslant X \leqslant N_1, \Delta'', N_1 \leqslant Y \leqslant M, \Delta''' \qquad \Delta^{\exists}; \Delta; \theta \vdash N_1 <: N_2; \Delta^{\exists'}; \theta'}{\Delta^{\exists}; \Delta; \theta \vdash X \lessdot Y; \Delta^{\exists'}; \theta'}$$

[EP-Witness]

$$\frac{\begin{array}{cc} X \notin dom(\theta) & \Delta^{\exists} = \Delta^{\exists'}, L \leqslant X \leqslant M, \Delta^{\exists''} \\ \Delta^{\exists}\Delta \vdash \theta(L) <: \theta(T) & \Delta^{\exists}\Delta \vdash \theta(T) \lessdot (\{X \mapsto \theta(T)\}\theta)(M) \end{array}}{\Delta^{\exists}; \Delta; \theta \vdash T \preceq X; \Delta^{\exists'}\Delta^{\exists''}; \{X \mapsto \theta(T)\}\theta}$$

[S-Var]

$$\frac{\begin{array}{cc} \Delta^{\forall} = \Delta^{\forall'}, L \leqslant X \leqslant M, \Delta^{\forall''} & \Delta^{\forall}\Delta \vdash \theta(L) \lessdot \theta(T) \\ \Delta^{\forall}\Delta \vdash \theta(T) \lessdot \{X \mapsto T\}\theta(M) & \Delta^{\forall'}, T \leqslant X \leqslant M, \Delta^{\forall''}\Delta \vdash \theta(T) \lessdot \theta(X) \end{array}}{\Delta^{\forall}; \Delta; \theta \vdash \exists\Delta^{\exists}{}_1.\ \theta(T) <: \exists\Delta^{\exists}{}_2.\ X; \Delta^{\forall'}, T \leqslant X \leqslant M, \Delta^{\forall''}; \theta}$$

Fig. 9. Changes for full subtyping

Consider, given generic classes `C<X>` and `D<X>`, a method `m` with the signature `<X> Object m(C<? extends D<X>> x)`; the type of the method argument would be written in our calculus as $\exists\bot \leqslant \texttt{Z} \leqslant \texttt{D<X>}.\ \texttt{C<Z>}$. When applied to an argument of type `C<T>`, we expect the application to typecheck whenever `T` is a subtype of `D<U>` for some `U`. Our rules attempt to derive the existential promotion $\texttt{T} \preceq \texttt{Z}$, and, in doing so, test whether `T` is a subtype of `Z`'s upper bound. Yet this condition is only tested using the pointwise subtyping relation, so while the type variables `X` and `Z` had previously been available for inference, at no point in the pointwise sub-derivation may the method's type argument `X` be bound to `U`, which is precisely what is needed to typecheck this application. Thus the application will be rejected and the programmer will be forced to write explicit type arguments that, on the surface, seem quite unnecessary.

The simplest solution, of course, is to strengthen our rules by lifting the restriction to pointwise subtyping in these instances. We call this variant full subtyping; the key altered rules can be seen in Figure 9. Note that the pointwise subtyping relation must now, like the other relations, distinguish between type variables that can and cannot be instantiated, although in all rules but PS-Class it need not make use of this information. Another equivalent solution would involve losing the distinction between $\lessdot$, $\preceq$, and $<:$ altogether.

The full subtyping relation is intended to correspond to Java 5 subtyping as defined by the JLS. Although we offer no formal proof of this, the correspondence between full ∃FJ subtyping and Java 5 subtyping should be clear by inspection, as our rules attempt to follow the JLS rather closely in this regard. We leave a formal analysis of JLS type inference for future work; indeed, our search for a comprehensible type inference formalism came about largely due to the difficulty of working with the presentation in the JLS.

2.4 Type Soundness

We show type soundness first by showing the soundness of the system without type inference—that is, the case where every method invocation explicitly supplies its type arguments—and then proceed to show that any instantiation produced by inference leads to valid explicit type parameters if existential type variables can be used outside of their scope. This last condition is required to account for the implicit unpacks or capture conversion performed by the inference process. Analogous proofs hold for both the full and kernel subtyping variants; we give a sketch that can apply to either.

Theorem 1. *The $\exists FJ$ type system is sound whenever all method calls are explicitly given type parameters. That is, well-type terms are either values or can take a step by our operational semantics. This step will yield either another well-typed term or the special result indicating an illegal typecast.*

Proof. Igarashi, Pierce, and Wadler have shown type soundness for Featherweight GJ [3]; their proof adapts readily to our system given a few modifications. For example, one cannot conclude that `C<T>` $<:$ `D<U>` and `C<T'>` $<:$ `D<U>` if and only if `T` $=$ `T'`, a property invoked in the proof that type substitution preserves subtyping. Still, the changes to be made are comparatively minor; a complete proof can be found in our forthcoming technical report. □

We now wish to show that, assuming existential variables can be referenced outside of their scope, one can always replace a method invocation with inferred type parameters by one with explicit type parameters. We begin by introducing a few lemmas, the first two of which state that well-formed type environments and substitutions on the right-hand side of the subtyping relation imply well-formed type environments and substitutions on the left-hand side. The type environment $\hat{\Delta}$ in Lemma 2 is meant to represent appropriate bounds for all variables involved in the judgment.

Lemma 1. *For any type environments $\Delta^\forall$ and Δ such that $\Delta\Delta^\forall$ is well-formed and any well-formed θ, σ, and τ, if $\Delta^\forall; \Delta; \theta \vdash \sigma <: \tau; \Delta^{\forall\prime}; \theta'$ for some $\Delta^{\forall\prime}$ and θ', then $\Delta^{\forall\prime}$ is also well-formed.*

Lemma 2. *Given a well-formed initial typing environment $\hat{\Delta}$ and substitution θ such that $\hat{\Delta} \vdash \theta$ ok, if $\Delta^\exists; \Delta; \theta \vdash S \preceq T; \Delta^{\exists\prime}; \theta'$ for any S and T and any $\Delta^\exists$, $\Delta^{\exists\prime}$, and Δ contained in $\hat{\Delta}$, then we also have that $\hat{\Delta} \vdash \theta'$ ok.*

The proofs are straightforward. Two more lemmas, which form the crux of our proof, establish that, whenever subtyping is allowed because of the instantiation of some variables, it would also be possible for that subtyping to go through by substituting for those variables "in advance," assuming—to account for capture conversion—that existential variables can be accessed from outside of their scope. Lemma 4 follows from Lemma 3; the two are are identical in every respect other than the relation to which they refer.

Lemma 3. *Whenever* $\Delta^{\exists}; \Delta; \theta \vdash S \preceq T; \Delta^{\exists'}; \theta'$*, and (for well-formed environments, types, and substitutions), it is also the case that* $\emptyset; \Delta; \emptyset \vdash S \preceq \hat{\theta}(T); \Delta^{\exists''}; \theta''$ *for some* $\Delta^{\exists''}$ *and* θ'' *and some* $\hat{\theta}$ *such that* $dom(\hat{\theta}) \subseteq dom(\Delta^{\forall})$*.*

Lemma 4. *Whenever* $\Delta^{\forall}; \Delta; \theta \vdash \sigma <: \tau; \Delta^{\forall'}\theta'$*, and* $dom(\Delta^{\forall}) \cap FV(\sigma) = \emptyset$ *(for well-formed environments, types, and substitutions), it is also the case that* $\Delta \vdash \sigma <: \hat{\theta}(\tau)$ *for some* $\hat{\theta}$ *such that* $dom(\hat{\theta}) \subseteq dom(\Delta^{\forall})$*.*

The proofs of both are again by straightforward induction over their respective relations. We now proceed with our main theorem:

Theorem 2. *Whenever* $\Delta; \Gamma \vdash e.m(e_1, \ldots, e_n) : \tau$*, there exist* $\rho_1, \ldots, \rho_k$ *such that* $\Delta; \Gamma \vdash e.\texttt{<}\rho_1, \ldots, \rho_k\texttt{>}m(e_1, \ldots, e_n) : \tau$*.*

Proof. By inversion of our typing rules, we know that $e.m(e_1, \ldots, e_n)$ must be typed by rule T-InferApp, and that $e.\texttt{<}\rho_1, \ldots, \rho_k\texttt{>}m(e_1, \ldots, e_n)$ must be typed by rule T-App. Our problem is then, given a derivation for $\Delta; \Gamma \vdash e.m(e_1, \ldots, e_n) : \exists\Delta'\overline{\Delta}[\overline{X} \mapsto \overline{T}]T'$ by rule T-InferApp, to construct a derivation for $\Delta; \Gamma \vdash e.\texttt{<}\exists\overline{\Delta}.\overline{T}\texttt{>}m(e_1, \ldots, e_n) : \exists\Delta'\overline{\Delta}[\overline{X} \mapsto \overline{T}]T'$ via T-App. The first three premises of T-App match exactly with those of T-InferApp; it remains to be shown, then, first that the type arguments inferred are within their respective bounds, and second that the types of the actual parameters remain legal after type variable substitution.

Examining these proof obligations in that order, we first observe that the *infer* operation, defined in Figure 5, serves only to take either the least upper bound of several lower bounds on the same type variable or the single declared upper bound of that variable. If such a least upper bound exists—which is always the case in ∃FJ, but is sometimes not the case in full Java—it must by definition be less than the variable's declared upper bound, given that contexts are well-formed. It is sufficient, then, to show two things; first that all substitutions θ resulting from the calculation of subtypes are legal, and second that all newly introduced lower bounds are less than their associated upper bounds. The first of these follows from Lemma 2, as only the existential promotion relation actively extends the substitution, while the second follows from Lemma 1.

We must next establish that the subtype relation between the actual and formal parameter types is preserved if type arguments, instead of being inferred, are explicitly substituted. This follows directly from Lemma 4. With that we have all the components we need to construct a use the rule T-App. □

Since we know from Theorem 1 that the system without the T-InferApp rule is sound, we have:

Corollary 1. *The ∃FJ type system is sound; that is, well typed terms can only go wrong as the result of an illegal typecast.*

This definition of type soundness is the same as stated for FJ [3].

3 Decidability

Having shown our type system to be sound, we now desire to determine whether it is decidable. It is clear by examining our type system that this is the case as long as we have a decidable subtyping algorithm, so that is where we will focus our attention. We begin by presenting such an algorithm for the kernel variant of subtyping.

3.1 Algorithmic Subtyping of Kernel ∃FJ

A set of inference rules does not necessarily give a decision procedure. If the rules are not syntax directed—that is, if the shape of the conclusion does not uniquely determine a single applicable rule—then some work must be done to disambiguate the rule system in order to arrive at a deterministic algorithm. In addition, some termination measure must be established to guarantee that the procedure suggested by the rules always halts.

Lemma 5. *The rules given in Section 2.2 for ∃FJ can be converted to algorithmic form; that is, the applicable rule and instantiations of its premises can be made obvious from the form of the conclusion.*

Proof. Of the rules given in Section 2.2, PS-Refl, PS-Trans, EP-Refl, EP-Subst1, EP-Subst2, and S-Ground are not syntax directed. The reflexivity rules are easily taken care of; even without removing the rules and taking steps to ensure that reflexivity continues to hold in the system, a check for equality between two arguments before applying other rules cannot be seen as an obstacle to a decidable algorithm. The substitution rules are likewise not problematic; one can envision the substitution θ being eagerly applied whenever a variable in its domain is encountered. In fact, since EP-Witness removes X from $\Delta^{\exists}$ whenever X is added to θ, little else can be done with X after this point.

In systems with structural subtyping—for example, $F_{<:}$ [4]—transitivity can present some difficulty, as it essentially allows us to guess a T whenever presented with $S \lessdot U$. However, Java's nominal subtyping gives us an obvious "guess"; if it is indeed the case that $S \lessdot U$ holds, then we should be able to instantiate T with either the declared bound or the immediate superclass of S, as appropriate. We omit the details of how to express this in our rules, but it should be clear that it is not too difficult of a task to replace PS-Class and PS-Trans with a rule that explicitly walks up the class hierarchy. From there, the other rules need only slight modification before transitivity can become a derived property of the system.

The situation with S-Ground is similar to PS-Trans, although it is easy to tell when the rule should be applied; the problem is the new type T' that must be "guessed" whenever the rule is applied. However, the intent of the rule is that T' be convertible to N via existential promotion. Thus, taking a cue from Odersky's earlier discussion of type inference for GJ [11], we can augment our pointwise subtype relation with a "goal" type and consider climbing the class hierarchy until we match—temporarily ignoring type arguments—this goal. □

Once we have an appropriately algorithmic set of rules, the more interesting decidability result follows easily.

Theorem 3. *Subtyping in kernel ∃FJ is decidable.*

Proof. Lemma 5 gives us that the declarative system can be made everywhere syntax-directed. We observe both that the subtyping relation (<:) never refers back to itself, and that the existential promotion relation—once non-syntax-directed rules are removed—refers to itself only on smaller terms, through EP-Class. We are left then with pointwise subtyping (≤), but given our discussion of transitivity above, application of these rules is also clearly terminating. Comparisons involving type variables become comparisons of class types, which terminate in a finite traversal of the class hierarchy.

One further complication involves the well-formedness constraints on types, type environments, and type substitutions, which are not explicitly stated in our rules. Some of them, however—in particular the constraints listed in Figure 3—make reference to the subtyping relation. Note, however, that well-formedness of type environments need never be explicitly tested by an algorithm; Lemma 1 gives us this for free, given that a well-formed class always generates a well-formed initial environment. Similarly with type substitutions and Lemma 2. We can thus conclude that subtyping for kernel ∃FJ is decidable. □

3.2 Undecidability of Full ∃FJ

When we move from the kernel to the full variant of subtyping, however, we encounter some difficulties. While the process of converting the rules to syntax-directed form is the same as given above, the argument for termination makes direct use of the fact, mentioned in Section 2.2, that the comparison of bounds in such rules as PS-VMixed, EP-Witness, and S-Var checks only pointwise and not full subtyping. This is precisely the restriction done away with in full ∃FJ, and indeed, both full ∃FJ and Java 5 suffer from undecidable subtyping.

Syntax

$$
\begin{array}{lcll}
\rho, \sigma, \tau & ::= & \alpha \mid \top \mid \tau \to \tau \mid \forall \alpha <: \tau.\ \tau & \textit{types} \\
e & ::= & x \mid \lambda x : \tau.\ e \mid e\ e \mid \Lambda \alpha <: \tau.\ e \mid e[\tau] & \textit{expressions} \\
\Gamma & ::= & \emptyset \mid \Gamma, x : \tau & \textit{typing contexts} \\
\Delta & ::= & \emptyset \mid \Delta, \alpha <: \tau & \textit{type environments}
\end{array}
$$

Subtyping: $\Delta \vdash \sigma <: \tau$

$$[\text{FS-Refl}]\ \frac{}{\Delta \vdash \tau <: \tau} \qquad [\text{FS-Trans}]\ \frac{\Delta \vdash \rho <: \sigma \qquad \Delta \vdash \sigma <: \tau}{\Delta \vdash \rho <: \tau}$$

$$[\text{FS-Top}]\ \frac{}{\Delta \vdash \tau <: \top} \qquad [\text{FS-TVar}]\ \frac{\Delta = \Delta', \alpha <: \tau, \Delta''}{\Delta \vdash \alpha <: \tau}$$

$$[\text{FS-Arrow}]\ \frac{\Delta \vdash \tau_1 <: \sigma_1 \qquad \Delta \vdash \sigma_2 <: \tau_2}{\Delta \vdash \sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2}$$

$$[\text{FS-All}]\ \frac{\Delta \vdash \tau_1 <: \sigma_1 \qquad \Delta, \alpha <: \tau_1 \vdash \sigma_2 <: \tau_2}{\Delta \vdash \forall \alpha <: \sigma_1.\ \sigma_2 <: \forall \alpha <: \tau_1.\ \tau_2}$$

Fig. 10. $F_{<:}$ - the polymorphic λ-calculus with subtyping

Evidence for this was first noticed by Odersky [12], who provided the following example, which causes even the latest Java 5 compiler from Sun (in addition to their Java 6 snapshots, at the time of this writing) to exhaust their stack space:

```
class F<T> {}

class C<X extends F<F<? super X>>> {
  C(X x) {
    F<? super X> f = x;
  }
}
```

As Odersky observes, in order to verify `X <: F<? super X>`, one must show that `F<F<? super X>> <: F<? super X>`, but this itself quickly reduces back to `X <: F<? super X>`. Our formalism, if extended in the naive way suggested above, would be subject to similar problems. In fact, it turns out that these problems cannot be avoided; full ∃FJ (and, hence Java 5) subtyping is undecidable.

We prove this undecidability result by reduction from subtyping in $F_{<:}$, which is known to be undecidable [13]. $F_{<:}$ is based on the polymorphic λ-calculus (also called System F [9]) with the addition of structural subtyping; like Java 5, a re-

$$
\begin{array}{rcl}
[\![\alpha]\!] & = & (\mathtt{X}_\alpha;\ \cdot) \\
[\![\top]\!] & = & (\mathtt{Object};\ \cdot) \\
[\![\tau_1 \to \tau_2]\!] & = & (\mathtt{Arr<?\ super}\ \mathit{fst}\ [\![\tau_1]\!]\mathtt{,\ ?\ extends}\ \mathit{fst}\ [\![\tau_2]\!]\mathtt{>}; \\
 & & \ \mathit{snd}\ [\![\tau_1]\!], \mathit{snd}\ [\![\tau_2]\!]) \\
[\![\forall\alpha <: \tau_1.\ \tau_2]\!] & = & (\mathtt{All<?\ super}\ \mathit{fst}\ [\![\tau_1]\!]\mathtt{,\ ?\ extends}\ \mathit{fst}\ [\![\tau_2]\!]\mathtt{>}; \\
 & & \ \mathtt{X}_\alpha \lhd \mathit{fst}\ [\![\tau_1]\!], \mathit{snd}\ [\![\tau_1]\!], \mathit{snd}\ [\![\tau_2]\!]) \\
[\![\emptyset]\!] & = & \cdot \\
[\![\Delta, \alpha <: \tau]\!] & = & [\![\Delta]\!], \mathtt{X}_\alpha \lhd \mathit{fst}\ [\![\tau]\!], \mathit{snd}\ [\![\tau]\!] \\
[\![\Delta \vdash \sigma <: \tau]\!] & = & \mathtt{class\ Arr<X,\ Y>\ \{\}} \\
 & & \mathtt{class\ All<X,\ Y>\ \{\}} \\
 & & \\
 & & \mathtt{class\ Fsub<}[\![\Delta]\!]\mathtt{,}\ \mathit{snd}\ [\![\sigma]\!]\mathtt{,}\ \mathit{snd}\ [\![\tau]\!]\mathtt{>\ \{} \\
 & & \quad \mathit{fst}\ [\![\tau]\!]\ \mathtt{decide(}\mathit{fst}\ [\![\sigma]\!]\ \mathtt{x)\ \{} \\
 & & \quad\quad \mathtt{return\ x;} \\
 & & \quad \mathtt{\}} \\
 & & \mathtt{\}}
\end{array}
$$

Fig. 11. From $F_{<:}$ to Java 5

stricted variant of $F_{<:}$ (called kernel $F_{<:}$) can be shown to be decidable. The syntax and subtyping rules for the undecidable full $F_{<:}$ can be seen in Figure 10; its typing and evaluation rules are standard. We abuse notation slightly by employing many of the same metavariables used for ∃FJ; it should be clear from context which system is being referenced.

Interestingly enough, replacing the premise $\Delta \vdash \tau_1 <: \sigma_1$ in FS-All with $\tau_1 = \sigma_1$ yields the decidable system referred to as kernel $F_{<:}$. Thus, as in Java, it is the ability to fully compare the bounds of type variables that results in undecidability.

We define a translation $[\![\ \text{-}\ ]\!]$ from subtyping judgments in $F_{<:}$ to a subset of Java 5 that falls within the scope of ∃FJ. Because Java allows type variables to be introduced only in class definitions, we map each $F_{<:}$ type to a pair consisting of a Java type and a list of type variables introduced by the $F_{<:}$ type; for lack of better notation, we denote the type component by $\mathit{fst}\ [\![\tau]\!]$ and the list component by $\mathit{snd}\ [\![\tau]\!]$. We then generate from an $F_{<:}$ subtyping judgment a Java class that typechecks—according to full $F_{<:}$subtyping, and according to Java subtyping as defined by the JLS—if and only if the subtyping judgment holds.

The details of our encoding can be seen in Figure 11; as usual, we assume uniqueness (or implicit renaming) of bound variables. $F_{<:}$ type variables map to similarly bounded Java type variables and $\top$ maps to `Object`, as one would expect. Function and universal types are both associated with a parameterized Java class, and the contravariance and covariance of their component types are reflected in their `super` and `extends` wildcard arguments. In addition, the bound

variable of a universal type is propagated upwards and becomes a type variable bound by the class, allowing uses of the Java type variable to properly stand for uses of the $F_{<:}$ type variable. Haskell source code for this encoding can be found in our forthcoming technical report.

Lemma 6. *The* $F_{<:}$ *judgment* $\Gamma \vdash \sigma <: \tau$ *holds if and only if* $[\![\Gamma \vdash \sigma <: \tau]\!]$ *is a valid class.*

Proof. It is easy to see, by induction over the structure of $F_{<:}$ types, that any true subtyping judgment will yield a Java class that typechecks successfully—after transforming wildcards to existential types—in full ∃FJ. If we restrict our attention to the subset of Java classes generated by our translation, we can similarly see that the implication holds in the other direction. □

Theorem 4. *Subtyping in full ∃FJ is undecidable.*

Proof. Follows directly from Lemma 6 and the undecidability of full $F_{<:}$. □

Note that our reduction was made possible because of two features of full ∃FJ. First, we are able to capture the covariance and contravariance inherent in $F_{<:}$ subtyping using wildcards, which implies that such a reduction would also be possible in Java with use-site variance annotations as proposed by Igarashi and Viroli [14]. Second, although our translation does not technically produce F-bounded type variables—no variable appears directly in its own bound—it does create many complex dependencies among variable bounds, which could not be captured by the rules of kernel ∃FJ. And since we never rely on any types that are not expressible with wildcards, our undecidability result extends to Java 5.

Having implemented our translation as a small Haskell program, it is interesting to note that the original counter-example to a proposed algorithm for subtyping in $F_{<:}$ [15]—that is, a subtyping judgment that causes this procedure not to terminate—is in fact rejected by the current Java compiler. Of course, the encoded example is much more complex than the example given in Section 3.2, and it relies on some rather elaborate recursive bounds. It seems likely, as will be further discussed in the next section, that the Java 5 compiler (and likewise the Java 6 snapshot) contains ad-hoc restrictions that rule out such types, although they appear to be doing so at the expense of certain valid and desirable programs.

4 Discussion

Having shown the undecidability of subtyping in Java 5—and hence the undecidability of type inference, which, as we established in Section 2, is a direct extension of subtyping—we must now determine what this means for Java in practice. Undecidability does not necessarily make a system unusable; in fact, it has been claimed that the undecidability of $F_{<:}$ is less of a problem in practice than its lack of greatest lower bounds and least upper bounds [15, 8]. At present,

```
class Group<E extends Comparable<? super E>>
  extends ArrayList<E>
  implements Comparable<Group<? extends E>> {}

class Sequence<E extends Comparable<? super E>>
  extends TreeSet<E>
  implements Comparable<Sequence<? extends E>> {}

class Test<T extends Comparable<? super T>> {
  <C extends Collection<T>> void foo(SortedSet<? extends C> setToCheck,
                                     SortedSet<? extends C> validSet) {}

  public void containsCombination(SortedSet<Group<T>> groups,
                                  SortedSet<Sequence<T>> sequences) {
    foo(groups, sequences);
  }
}
```

Fig. 12. Example on which type inference diverges

however, Java does suffer from several real-world problems related to its undecidability; we explore these and suggest possible solutions in Section 4.1. Another question, discussed in Section 4.2, is how we might extend type inference to account for additional information.

4.1 Decidability

The undecidable example given in Section 3.2 does not look like something that would come up in practice, so one might hope that Java's undecidability would thus become a non-issue. Unfortunately, the latest implementations of Java from Sun—both the stable Java 5 release[1] and the Java 6 snapshots[2]—do fail to terminate (or terminate with a stack overflow) on real world code. Figure 12 shows an example from Sun's bug database [16] for which type inference diverges. The call to `foo` can be made to typecheck successfully by inserting an explicit type parameter `Collection<T>`.[3]

In addition, it would appear that Sun's current Java implementation is taking steps to catch many cases that might otherwise lead to infinite recursion of its subtyping and type inference algorithms; for instance, as mentioned in Section 3.2, the example subtyping judgment that yields non-termination in that system translates to a class that Java rejects, most likely because of the highly complicated dependencies among type variable bounds. However, such vigilance

[1] Last tested using `javac` version 1.5.0_06.

[2] Last tested using `javac` version 1.6.0 build for December 15th, 2005.

[3] At the time of this writing, the Java 6 snapshot's behavior on this example is even worse than that of the Java 5 compiler, as it no longer runs out of stack space; users must infer for themselves that the compiler will not terminate.

```
class A<T extends A<T>> {}
class B extends A<B> {}
class C extends B {}
class D<T> {}

class Simple {
  <T extends A<T>, S extends T> D<T> m(S s) {
    C c = null;
    D<B> d = m(c);
  }
}
```

Fig. 13. Example erroneously rejected by typechecker

on the part of the Java compiler—which does not appear to correspond to any restrictions mentioned in the JLS—sometimes goes too far. Figure 13, also taken from Sun's bug database [17], gives an example of a wrongfully rejected program that attempts to take advantage of complex type variable bounds. Torgersen, Ernst, and Plesner Hansen [2] give an example of a graph class built from mutually-dependent nodes and edges that leads to similar errors, despite the JLS giving every indication that it should typecheck.

The current implementation thus demonstrates three sorts of error cases: terms that should not typecheck which lead to infinite recursion, terms that *should* typecheck but instead lead to infinite recursion, and terms that should typecheck but are rejected. We look to our formalism to address these problems. As seen in Section 3, full ∃FJ accepts every term we would like to typecheck but gives no guarantee of termination, while kernel ∃FJ always terminates yet excludes many terms we might like to typecheck. Observe, however, that in comparing the rules of the two systems we have identified exactly three points—the three rules in Figure 9, which are strictly more powerful than their equivalents in kernel ∃FJ—where the tradeoff must be made between decidability and expressivity. We can thus put a bound on the number of times that it is permissible to use the complete subtype relation—as opposed to the pointwise fragment—at those points. By contrast, it is difficult to locate analogous tradeoff points in the presentation of subtyping and type inference found in the JLS.

In other words, we propose that a reasonable algorithm for subtyping might, for some constant k, permit the full subtype relation to be checked instead of the pointwise subtyping relation in these three rules a fixed number of times for each subtyping derivation. This avoids the problem of ad-hoc approaches—the algorithm is guaranteed to terminate, and the value of k can always be increased if it is found to exclude useful programs—while being more permissive and requiring less fine-tuning to ensure usability than a restriction based solely off the height of the derivation, the time spent calculating subtypes, or some similar measure. To improve performance, one might also keep track of subtype queries currently being checked and terminate early if one of these queries is repeated, but such optimizations are not strictly necessary.

```
interface Function<A, B> {
  B apply(A x);
}

class Id<A> implements Function<A, A> {
  public A apply(A x) {
    return x;
  }
}

class Test {
  <A> Id<A> identity() {
    return new Id<A>();
  }

  <B> B applyToString(Function<String, B> f) {
    return f.apply("abc");
  }

  void test() {
    String s = applyToString(identity());
  }
}
```

Fig. 14. Example requiring more powerful type inference

4.2 Extensibility

Another important issue is the amenability of type inference to further refinements. Neither our formalism nor the JLS claims the ability to infer type arguments for every method call, with the most obvious counter-examples being methods that take no parameters—e.g., a class `List` might have a static method `<T> List<T> nil()` returning the empty list at type `T`. In order to infer type arguments in situations like this we must incorporate type information from other sources.

The JLS provides for type inference based on the expected result type only when the method result is on the immediate right hand side of an assignment statement. While this restriction almost certainly derives from the possibility that the method result might be passed as an argument to another method with its own set of type parameters, it means that examples like the one in Figure 14—suggested by Odersky in an earlier formalization of type inference [11]—fail to typecheck.

It is not obvious how type inference as presented in the JLS could be refined to handle such cases; how can one derive constraints from a context that might require its own type inference, which itself relies the complete type currently being inferred? In our formalism, however, the solution is simple, if a bit less elegant than our current presentation. Instead of promoting a type variable to its upper bound when an instantiation cannot be found—as specified by I-Default

in Figure 5—we allow variables to remain unspecified and propagate outwards whenever the expression is to be passed to another method. In fact, all type variables not bound by a substitution θ can be given this treatment, with values being fixed only when the expression nesting terminates in some context not amenable to further inference.

5 Conclusion

The addition of generic and wildcard types has made typechecking Java 5 programs complicated. Indeed, as we have shown, the presence of full F-bounded subtyping makes the problem undecidable for full $\exists$FJ, a result that extends to Java 5 as well. The existing description of Java 5 typechecking, as given in the Java Language Specification, is verbose and difficult for programmers and compiler writers alike to understand. We have tried to clarify the problem by studying type inference and typechecking in a simpler core language that highlights their difficulties. This paper provides a declarative specification of $\exists$FJ inference and typechecking that we have proven sound. $\exists$FJ itself is useful as a vehicle for studying further refinements and extensions to the Java 5 type system, and we expect that it will readily scale to full Java implementations. Our formalism also suggests a natural way to cut back on the inherent undecidability of typechecking by giving up on completeness, but it remains to be seen exactly how this approach will fare in practice. We speculate that, besides being a more principled way of determining the compiler's behavior, our approach will be at least as complete as existing Java 5 implementations. We are in the process of verifying this claim by building a full-scale Java 5 compliant compiler whose typechecking algorithm is based on the one described here.

Acknowledgments We would like to thank Kim Bruce, Atsushi Igarashi, Martin Odersky and Stephen Tse who provided helpful and insightful discussion about Java 5 typechecking. We also thank Brian Aydemir and Geoffrey Washburn who gave us useful comments on earlier drafts of this paper.

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition. Addison-Wesley (2005) ISBN 0-321-24678-0.
2. Torgersen, M., Ernst, E., Plesner Hansen, C.: Wild FJ. In Wadler, P., ed.: Proceedings of FOOL 12, Long Beach, California, USA, ACM, School of Informatics, University of Edinburgh (2005) Electronic publication, at the URL given below.
3. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java. In: Conference of Object-Oriented Programming, Systems, Languages and Applications. Volume 34 of ACM SIGPLAN Notices., ACM Press (1999)
4. Cardelli, L., Mitchell, J.C., Martini, S., Scedrov, A.: An extension of system F with subtyping. In Ito, T., Meyer, A.R., eds.: Proc. of 1st Int. Symp. on Theor. Aspects of Computer Software, TACS'91, Sendai, Japan, 24–27 Sept 1991. Volume 526. Springer-Verlag, Berlin (1991) 750–770

5. Canning, P., Cook, W., Hill, W., Mitchell, J., Olthoff, W.: F-bounded polymorphism for object-oriented programming. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture. (1989) 273–280
6. Bruce, K.: Slides for NEPLS talk "Sneaking existentials into Java 5". Personal Communication (2005)
7. Mitchell, J.C.: Foundations for Programming Languages. Foundations of Computing Series. The MIT Press (1996)
8. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
9. Reynolds, J.C.: Introduction to part II, polymorphic lambda calculus. In Huet, G., ed.: Logical Foundations of Functional Programming. Addison-Wesley, Reading, Massachusetts (1990) 77–86
10. Pfenning, F.: On the undecidability of partial polymorphic type reconstruction. Fundamenta Informaticae **19**(1/2) (1993) 185–199
11. Odersky, M.: Inferred type instantiation for GJ. (2002)
12. Odersky, M.: Is Java 1.5 subtyping decidable? Personal Communication (2005)
13. Pierce, B.C.: Bounded quantification is undecidable. Information and Computation **112**(1) (1994) 131–165 Also in C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, MIT Press, 1994. Summary in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Albuquerque, New Mexico.*
14. Igarashi, A., Viroli, M.: On variance-based subtyping for parametric types. In: Proceedings of the European Conference on Object-oriented Programming (ECOOP'02). Lecture Notes in Computer Science, Malaga, Spain (2002) 441–469
15. Ghelli, G.: Divergence of $F_{\leq}$ type-checking. Theoretical Computer Science **139**(1,2) (1995) 131–162
16. Sun Bug Database: Bug ID: 6273455. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6273455` (2005)
17. Sun Bug Database: Bug ID: 6278587. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6278587` (2005)

A Type system

$$[\text{WF-Object}]\ \frac{}{\Delta \vdash \texttt{Object ok}} \qquad [\text{WF-Object}]\ \frac{}{\Delta \vdash \bot\ \mathsf{ok}}$$

$$[\text{WF-TVar}]\ \frac{L \leqslant X \leqslant M \in \Delta}{\Delta \vdash X\ \mathsf{ok}} \qquad [\text{WF-Ex}]\ \frac{\Delta \vdash \Delta'\ \mathsf{ok} \qquad \Delta\Delta' \vdash T\ \mathsf{ok}}{\Delta \vdash \exists \Delta'.T\ \mathsf{ok}}$$

$$[\text{WF-Class}]\ \frac{CT(C) = \texttt{<}\overline{X} \lhd \overline{\exists \Delta'.\ N}\texttt{>} \lhd N \qquad \Delta \vdash \overline{T}\ \mathsf{ok} \qquad \emptyset \vdash \overline{\exists \Delta.\ T} <: \overline{\exists \Delta'.\ N}}{\Delta \vdash C\texttt{<}\overline{T}\texttt{>}\ \mathsf{ok}}$$

Fig. 15. Type well-formedness constraints

$$[\text{T-Var}]\ \frac{\Gamma = \Gamma', x : \tau, \Gamma'}{\Delta; \Gamma \vdash x : \tau} \qquad [\text{T-Field}]\ \frac{\Delta; \Gamma \vdash e_0 : \tau_0 \qquad \mathit{fields}(\tau_0) = \overline{\tau}\ \overline{f}}{\Delta; \Gamma \vdash e_0 \,.\, f_i : \tau_i}$$

$$[\text{T-New}]\ \frac{\mathit{fields}(\exists \emptyset.\ N) = \overline{\tau}\ \overline{f} \qquad \Delta; \Gamma \vdash \overline{e} : \overline{\sigma} \qquad \Delta; \vdash \overline{\sigma} <: \overline{\tau}}{\Delta; \Gamma \vdash \texttt{new}\ \ N(\overline{e}) : \exists \emptyset.\ N}$$

$$[\text{T-UCast}]\ \frac{\Delta; \Gamma \vdash e_0 : \sigma \qquad \Delta \vdash \sigma <: \tau}{\Delta; \Gamma \vdash (\tau) e_0 : \tau}$$

$$[\text{T-DCast}]\ \frac{\Delta; \Gamma \vdash e_0 : \sigma \qquad \Delta \vdash \tau <: \sigma \qquad \tau \neq \sigma}{\Delta; \Gamma \vdash (\tau) e_0 : \tau}$$

$$[\text{T-SCast}]\ \frac{\Delta; \Gamma \vdash e_0 : \sigma \qquad \Delta \vdash \sigma \not<: \tau \qquad \Delta \vdash \tau \not<: \sigma \qquad \mathit{stupid\ warning}}{\Delta; \Gamma \vdash (\tau) e_0 : \tau}$$

Fig. 16. Additional ∃FJ expression typing rules: $\Delta; \Gamma \vdash e : \tau$

$$[\text{T-NewMeth}]\ \frac{\begin{array}{c} CT(C) = \texttt{<}\overline{X} \lhd \overline{\exists \Delta'.\ M}\texttt{>} \lhd M \qquad (m, \exists \emptyset.\ M) \notin \mathit{mtype} \\ \overline{X} : \overline{\exists \Delta.\ N}, \overline{Y} : \overline{\exists \Delta'.\ M}; \overline{x} : \overline{\tau}, \texttt{this} : C\texttt{<}\overline{Y}\texttt{>} \vdash e : \tau \end{array}}{\texttt{<}\overline{X} \lhd \overline{\exists \Delta.\ N}\texttt{>}\ \tau\ m(\overline{\tau}\ \overline{x})\ \ \{\ \texttt{return}\ e;\ \}\ \ \mathsf{ok\ in}\ C}$$

$$[\text{T-OverMeth}]\ \frac{\begin{array}{c} CT(C) = \texttt{<}\overline{X} \lhd \overline{\exists \Delta'.\ M}\texttt{>} \lhd M \\ \overline{X} : \overline{\exists \Delta.\ N}, \overline{Y} : \overline{\exists \Delta'.\ M}; \overline{x} : \overline{\tau}, \texttt{this} : C\texttt{<}\overline{Y}\texttt{>} \vdash e : \tau \\ \mathit{mtype}(m, \exists \emptyset.\ M) = \texttt{<}\overline{X} \lhd \overline{\exists \Delta.\ N}\texttt{>}\ \overline{\tau} \to \tau \end{array}}{\texttt{<}\overline{X} \lhd \overline{\exists \Delta.\ N}\texttt{>}\ \tau\ m(\overline{\tau}\ \overline{x})\ \ \{\ \texttt{return}\ e;\ \}\ \ \mathsf{ok\ in}\ C}$$

$$[\text{T-Class}]\ \frac{\begin{array}{c} \mathit{fields}(\exists \emptyset.\ M) = \overline{\rho}\ \overline{g} \qquad \overline{f} \cap \overline{g} = \emptyset \\ \overline{md}\ \mathsf{ok\ in}\ C \qquad \nexists \overline{T}\ \text{s.t.}\ \exists \emptyset.\ M <: \exists \emptyset.\ C\texttt{<}\overline{T}\texttt{>} \end{array}}{\texttt{class}\ \ C\texttt{<}\overline{X} \lhd \overline{\exists \Delta'.\ M}\texttt{>} \lhd M\ \ \{\ \overline{\tau}\ \overline{f};\ \ \overline{md}\ \}}$$

Fig. 17. Method and class declaration typing rules