Towards a Semantic Model for Java Wildcards

Towards a Semantic Model for Java Wildcards

Alexander J. Summers
ETH Zurich
Switzerland
alexander.summers@inf.ethz.ch

Nicholas Cameron
Victoria Univ. of Wellington
New Zealand
ncameron@ecs.vuw.ac.nz

Mariangiola Dezani-Ciancaglini
Università di Torino
Italy
dezani@di.unito.it

Sophia Drossopoulou
Imperial College London
United Kingdom
s.drossopoulou@imperial.ac.uk

ABSTRACT

Wildcard types enrich the types expressible in Java, and extend the set of typeable Java programs. Syntactic models and proofs of soundness for type systems related to Java wildcards have been suggested in the past, however, the *semantics* of wildcards has not yet been studied.

In this paper we propose a semantic model for Java wildcards, inspired by work on semantic subtyping, which traditionally interprets types as sets of possible values. To easily reflect the nominal type system of Java, our model is defined in terms of runtime types (closed class types) rather than the structure of the runtime values themselves. To reflect the variance introduced by wildcards, our model interprets types as sets of such closed class types.

Having defined our model, we employ a standard semantic notion of subtyping. We show soundness of syntactic subtyping with respect to the semantic version, and demonstrate that completeness fails in the general case. We identify a restricted (but nonetheless rich) type language for which the syntactic notion of subtyping is both sound and complete.

1. INTRODUCTION AND BACKGROUND

Java wildcards are closely related to bounded existential types, which provide a richer underlying type language than that directly available to the programmer. Soundness and decidability of the Java type system has been studied in the context of existential types [2, 20], but there has been little effort to investigate the *meaning* of wildcards.

A *semantic* model of a type system provides a *meaning* of types. It can abstract from various syntactic details, and may be used to address the question of whether a type system is as powerful as it could be. In this paper we propose such a semantic model for Java types, and investigate the soundness and completeness of Java-like type assignment with respect to our model. We demonstrate that syntactic typing and subtyping are sound with respect to their semantic counterparts, while Java-like subtyping is not complete. We introduce some restrictions under which syntactic subtyping is also complete.

FTfJP '10, June 22, 2010, Maribor, Slovenia

Wildcards in Java.

Wildcards [7, 17, 16, 4] introduce subtype variance into Java. A *wildcard type* is a parameterised type in which the symbol `?` is used as an actual type parameter, for example in a type such as `List<?>`. Such a type can be thought of as a list of *some* type, where the wildcard is *hiding* that type. Wildcards may be given upper or lower bounds using the `extends` and `super` keywords respectively, e.g., `List<? extends Shape>`. Upper bounds give *covariance* and lower bounds *contravariance*: e.g., `List<? extends Circle>` is a subtype of `List<? extends Shape>`.

There have been several formalisations of the Java type system, for purposes including formal definition [17], type soundness [4], decidability of type checking [20, 14], and to gain better understanding of the mechanism [18, 3, 15].

Bounded Existential Types.

Existential types are a form of polymorphic type, based on logical existential quantification. In traditional treatments [5, 10, 13], existential types are introduced and eliminated explicitly in the expression syntax. An existential type is introduced using a *pack* expression which hides a *witness type*; a value with existential type must be *unpack*ed before it can be used. Unbounded existential types hide all information about the witness type. *Bounded* existential types [5, 12] give *partial* information about the witness type.

Generic types with variance annotations can be thought of in terms of existential types [9], and Java wildcards can similarly be understood using existential types [17, 16, 4]. For example, `Box<?>` can be represented as $\exists$`X.Box<X>`. Bounds on wildcards can be thought of as bounds on the existentially quantified type variable, thus `Box<? extends Shape>` corresponds to $\exists$`X:[`$\bot$ `Shape].Box<X>`, in which $\bot$ is the lower bound and `Shape` is the upper. We use $\bot$ as a lower bound when a wildcard is unbounded below and `Object` as a lower bound when a wildcard is unbounded above. We will omit bounds in examples when they do not play an important part.

Therefore, the type `Box<Circle>` can be packed to the type $\exists$`X:[`$\bot$ `Shape].Box<X>` and then unpacked to `Box<Z>` where `Z` is fresh and has bounds `[`$\bot$ `Shape]`. In the first case, the original type parameter `Circle` has been hidden

by X. Subtyping between wildcard and non-wildcard types reflects packing of existential types. For example,

Box<Circle> <: Box<? extends Shape>

corresponds to

Box<Circle> <: ∃X:[⊥ Shape].Box<X>

The main difference between traditional type-checking of existential types and type-checking wildcard types is that, for wildcard types in Java, packing and unpacking are implicit operations. Whenever a field or method lookup is to be performed on an expression whose type includes wildcards these types are temporarily unpacked; a process known as *capture conversion*.

The main motivation for using existential types to model Java wildcards is the presence of *expressible but not denotable types*, such as, e.g., List<∃ X:[⊥ Object].List<X>>. These are types which arise during Java type checking but which cannot be denoted using the Java syntax. This happens because the type checker internally employs a richer type language in the style of existential types. On the other hand, the syntax of existential types allows types which are not expressible in Java, i.e., for which no expression in a program written in the Java syntax has that type. For example, ∃ X:[C<X> Object].List<X> is not expressible in Java.

Semantic Subtyping.

Semantic types [6] represent types as the sets of values inhabiting them. Thus, semantic subtyping is naturally induced by the subset relation. In the setting of functional languages with semantic subtypes, completeness for type constructors which have natural set theoretic interpretations (such as cartesian products) comes for free, and subtyping algorithms can be derived by using boolean algebra laws for decomposing subtyping on simpler types. Semantic subtyping for object-oriented types has not yet been attempted.

Outline.

We begin in Section 2 by formulating a syntactic type system, corresponding to a natural extension of the Java type system to unrestricted F-bounded existential types. In particular, we define well-formedness of types, and the appropriate syntactic notions of subtyping and typing, which generalise those of the Java type system.

In Section 3, we turn to our semantic model. We introduce a semantic interpretation of closed class types as sets of class types, employing the sets to model variance. We define a standard notion of semantic subtyping, and give semantic versions of subtyping and typing for the full type syntax of the paper.

In Section 4, we address the questions of soundness and completeness with respect to our semantic notions. The syntactic notions of subtyping and typing turn out to be sound with respect to our model (4.1), whereas syntactic subtyping is shown to be incomplete (4.2). We then identify a key technical constraint on types which suffices to restore a partial completeness result; we call *rich types* the types fulfilling this constraint (4.2). We prove a weak completeness result for rich types, and then introduce the concept of *weakly independent environments*, designed to guarantee the requirement that types be rich via constraints which are much more easily understandable and checkable.

Finally, we conclude and discuss future work in Section 5.

2. SYNTACTIC TYPE SYSTEM

To make meaningful comparisons between Java and our semantic interpretation, we require a syntactic formalisation of the Java type system. We cannot make a formal comparison with either a Java compiler, nor the Java Language Specification [7], since these have only informal definitions. As with most previous models of Java with wildcards [17, 4], we use existential types to model wildcard types. We include assignment, since the presence of mutable state is relevant for the correct treatment of advanced type system features such as wildcards [15]. Our syntactic notion of subtyping is close to those employed in other existential type systems [20, 19, 13] but with some simplifications in presentation and a closer connection with the logical basis of existential types. Our language is rich enough to denote the types that can be expressed in Java (which is a superset of the types which can be denoted in Java). We can also write some types which cannot be expressed in Java; for example, ∃ X:[C<X> Object].C<X> can be neither denoted nor expressed in Java.

Following this, we present expressions in our language and rules for assigning types to expressions.

2.1 Types

We first present the language of types and the subtyping and well-formedness relations which operate on types. *Types* and *type environments* are defined by the following grammars:

N	::=	$C < \overline{T} >$	*class types*
T	::=	$X \mid \exists\Delta.N$	*types*
B	::=	$T \mid \bot$	*type bounds*
Δ	::=	$\overline{X : [B\ B]}$	*type environments*

Type environments map type variables to pairs of lower and upper bounds. We use N as short for $\exists\emptyset.N$. Wildcard types can be represented by existential types; their bounds are represented in the quantifying environment (Δ in $\exists\Delta.N$); for example, the Java type C<?> can be represented as the type ∃X:[⊥ Object].C<X> and D<? super E> as ∃X:[E Object].D<X>.

Well-formed types and environments.

Fig. 1 defines well-formed types and environments. The rules follow the formulations in FGJ [8] and Tame FJ [4], with the addition of F-Exists. The rule F-Bottom is used only to check bounds (since ⊥ can be used only as a bound). F-Object and F-Class define well-formed class types. In class declarations ◁ is short for "extends" and ... stands for the definition of fields and methods of the class as in [8, 4]. F-Exists defines well-formed existential types (which are used to represent wildcard types); it allows existential types to be checked by moving a set of quantified variables into the environment. For example the type ∃X:[⊥ C<Y>],Y:[C<X> Object].C<Y> is well-formed; notice that such a kind of quantification is not expressible in first order logic.

Subtyping.

We give rules for syntactic subtyping in Fig. 2. All rules other than those for existential types are standard. S-Exists-L corresponds to unpacking an existential type, the premises correspond with the premises of the type rules for an unpack expression [13]. S-Exists-R corresponds to packing

$$\frac{X \in dom(\Delta)}{\Delta \vdash X \text{ OK}}\ (\text{F-VAR}) \quad \frac{}{\Delta \vdash \bot \text{ OK}}\ (\text{F-BOTTOM}) \quad \frac{}{\Delta \vdash Object\!<\!> \text{ OK}}\ (\text{F-OBJECT}) \quad \frac{}{\Delta \vdash \emptyset \text{ OK}}\ (\text{F-ENV-EMPTY})$$

$$\frac{class\ C\!<\!\overline{X \lhd T_u}\!> \lhd N\{\ldots\} \quad \Delta \vdash \overline{T <: T_u[\overline{T/X}]}}{\Delta \vdash C\!<\!\overline{T}\!> \text{ OK}}\ (\text{F-CLASS}) \quad \frac{\Delta \vdash \Delta' \text{ OK} \quad \Delta, \Delta' \vdash N \text{ OK} \quad dom(\Delta') \cap fv(\Delta) = \emptyset}{\Delta \vdash \exists \Delta'.N \text{ OK}}\ (\text{F-EXISTS})$$

$$\frac{\Delta, X : [B_l\ B_u], \Delta' \vdash B_l <: B_u \quad \Delta, X : [B_l\ B_u], \Delta' \vdash \Delta' \text{ OK}}{\Delta \vdash X : [B_l\ B_u], \Delta' \text{ OK}}\ (\text{F-ENV})$$

Figure 1: Syntactic well-formed types and environments.

$$\frac{\Delta \vdash B \text{ OK}}{\Delta \vdash B <: B}\ (\text{S-REFLEX}) \quad \frac{\Delta \vdash B <: B'' \quad \Delta \vdash B'' <: B'}{\Delta \vdash B <: B'}\ (\text{S-TRANS}) \quad \frac{\Delta \vdash B \text{ OK}}{\Delta \vdash \bot <: B}\ (\text{S-BOTTOM})$$

$$\frac{\emptyset \vdash \Delta \text{ OK} \quad \Delta(X) = [B_l\ \ B_u]}{\begin{array}{l}\Delta \vdash X <: B_u \\ \Delta \vdash B_l <: X\end{array}}\ (\text{S-BOUND}) \quad \frac{class\ C\!<\!\overline{X \lhd T_u}\!> \lhd N\{\ldots\} \quad \Delta \vdash \overline{T <: T_u[\overline{T/X}]}}{\Delta \vdash C\!<\!\overline{T}\!> <: N[\overline{T/X}]}\ (\text{S-SUB-CLASS})$$

$$\frac{\Delta, \Delta' \vdash N <: T \quad dom(\Delta') \cap (fv(T) \cup fv(\Delta)) = \emptyset}{\Delta \vdash \exists \Delta'.N <: T}\ (\text{S-EXISTS-L}) \quad \frac{\Delta \vdash \overline{B_l[\overline{T/X}] <: T} \quad \Delta \vdash \overline{T <: B_u[\overline{T/X}]}}{\Delta \vdash N[\overline{T/X}] <: \exists \overline{X : [B_l\ B_u]}.N}\ (\text{S-EXISTS-R})$$

Figure 2: Syntactic subtyping.

in a similar way; $\overline{T}$ are witness types hidden by $\overline{X}$. Both rules can be seen as the natural extension of the existential elimination and introduction rules in first order logic to an arbitrary number of variables[1]. Similar rules are used in some existing models of wildcards [20, 3], while some authors combine them into a single rule (e.g., -ENV in [16, 2]).

Expressions.

We use a standard expression syntax for an imperative featherweight calculus: variables, `null`, field access and assignment, method invocation, and object creation (without parameters, as in [8, 4]). Method invocation is simpler than in other models because we do not support type parameters for methods. The syntax of expressions is as follows:

$$e \quad ::= \quad x \ \mid \texttt{null} \ \mid e.f \ \mid e.f = e \ \mid e.m(\overline{e}) \ \mid \texttt{new}\ N$$

Typing Rules.

We define syntactic typing rules in Fig. 3. *Variable environments*, ranged over by Γ, are defined by:

$$\Gamma ::= \overline{x : T}$$

The most interesting typing rules are those dealing with the introduction and elimination of existential types. Existential types are eliminated by implicit unpacking of the type of the receiver in T-FIELD, T-ASSIGN, and T-INVK. Method and field type lookups *fType* and *mType*[2], are applied to the unpacked (unquantified) types, and the appropriate extra bounds are present in the rule premises. To avoid the unpacked type variables escaping into the conclusion (which would be unsound), they must be quantified over once more in the conclusion.[3]

Our rules follow Tame FJ [4], but we simplify by not modelling capture conversion of method type parameters. We can therefore simplify re-packing of type variables: we do not require guarding environments to keep track of unpacked variables. Our rules reflect a simpler subset of Java than Tame FJ, however, all Tame FJ programs can be encoded in our calculus by encoding capture conversion of parameter types to unpacking of the receiver's type [3].

3. A SEMANTIC MODEL

In semantic types [6], types are interpreted as the sets of values inhabiting them. Since at runtime, Java objects have closed class types, our model interprets closed syntactic types as sets of such closed class types. The sets we use are closed under subclassing, to reflect inheritance-based variance. We use these sets to model the variance introduced by (bounded) existential types — the "hidden" information of the existentially-bound variables is dealt with considering each of the possible instantiations which may be hidden, i.e., by considering all substitutions mapping the variables to closed types satisfying their declared bounds.

We use σ to range over substitutions mapping type variables onto closed types, and say that a substitution σ *agrees*

[1]Subtyping coincides with derivability in a formulae-as-types isomorphism for the common syntax.

[2]Whose definitions are as in FGJ or Tame FJ and are elided here.

[3]In the typing rules T-ASSIGN and T-INVK the variables in the domain of Δ' are bound in the first premise, therefore they cannot occur free in the other premises by the "variable convention" (see [1] 2.1.12). In particular they cannot occur free in the expressions e' and $\overline{e}$.

$$\frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}\ (\text{T-Var}) \qquad \frac{\Delta \vdash N\ \text{ok}}{\Delta;\Gamma \vdash new\ N : \exists\emptyset.N}\ (\text{T-New}) \qquad \frac{\Delta;\Gamma \vdash e : \exists\Delta'.N \quad fType(f,N) = T}{\Delta;\Gamma \vdash e.f : \exists\Delta'.T}\ (\text{T-Field})$$

$$\frac{\Delta;\Gamma \vdash e : \exists\Delta'.N \quad fType(f,N) = T \quad \Delta,\Delta';\Gamma \vdash e' : T}{\Delta;\Gamma \vdash e.f = e' : \exists\Delta'.T}\ (\text{T-Assign}) \qquad \frac{\Delta;\Gamma \vdash e : U \quad \Delta \vdash U <: T}{\Delta;\Gamma \vdash e : T}\ (\text{T-Subs})$$

$$\frac{\Delta;\Gamma \vdash e : \exists\Delta'.N \quad mType(m,N) = \overline{U} : U \quad \Delta,\Delta';\Gamma \vdash \overline{e : U}}{\Delta;\Gamma \vdash e.m(\overline{e}) : \exists\Delta'.U}\ (\text{T-Invk}) \qquad \frac{\Delta \vdash T\ \text{ok}}{\Delta;\Gamma \vdash \texttt{null} : T}\ (\text{T-Null})$$

Figure 3: Syntactic typing rules.

$$\frac{}{\Gamma \models x : \Gamma(x)}\ (\text{S-Var}) \qquad \frac{}{\Gamma \models new\ N : [\![N]\!]}\ (\text{S-New}) \qquad \frac{\Gamma \models e : S \quad \forall N \in S, fType(f,N)\ defined}{\Gamma \models e.f : \bigcup_{N\in S} [\![fType(f,N)]\!]}\ (\text{S-Field})$$

$$\frac{\Gamma \models e : S \quad \Gamma \models e' : S' \quad \forall N \in S, S' \subseteq [\![fType(f,N)]\!]}{\Gamma \models e.f = e' : \bigcup_{N\in S} [\![fType(f,N)]\!]}\ (\text{S-Assign}) \qquad \frac{\Gamma \models e : S \quad S \subseteq S'}{\Gamma \models e : S'}\ (\text{S-Subs})$$

$$\frac{\Gamma \models e : S \quad \overline{\Gamma \models e : S} \quad \forall N \in S, \overline{S \subseteq S'}\ where\ \overline{S'} = [\![mType(m,N)\downarrow_1]\!]}{\Gamma \models e.m(\overline{e}) : \bigcup_{N\in S} [\![mType(m,N)\downarrow_2]\!]}\ (\text{S-Invk}) \qquad \frac{}{\Gamma \models \texttt{null} : S}\ (\text{S-Null})$$

Figure 4: Semantic typing rules.

with an environment Δ, formally $\Delta \vdash \sigma$, if σ respects the bounds declared in Δ:

Definition 1. $\Delta \vdash \sigma$ *if* $dom(\Delta) = dom(\sigma)$ *and for all* $X \in dom(\sigma)$ *we get* $\vdash \sigma(B_l) <: \sigma(X) <: \sigma(B_u)$, *where* $[B_l\ B_u] = \Delta(X)$.

Our model will only provide a semantics for closed types[4]; we deal with class parameters instead when we *use* this semantics, as we will show later. In the case of a non-quantified (i.e., class) type, the only variance comes from the possibility of subsumption - an object of a particular static type may have a runtime type which is a subclass. To define this, we use a *subclassing* judgement, $N' \sqsubseteq N$, defined to be the least preorder such that:

$$\frac{class\ C < \overline{X \lhd T_u} > \lhd N\{\ldots\} \qquad \Delta \vdash \overline{T <: T_u[\overline{T/X}]}}{\Delta \vdash C < \overline{T} > \sqsubseteq N[\overline{T/X}]}$$
(S-Sub-Class′)

Definition 2 (Semantics of Types). Semantic types*, ranged over by* S*, are sets of closed class types. We define a semantics for* closed *types via a mapping* $[\![.]\!]$ *from closed syntactic types to semantic types:*

$$\begin{array}{rcl} [\![N]\!] & = & \{N' \mid N' \sqsubseteq N\} \\ [\![\bot]\!] & = & \emptyset \\ [\![\ \exists\Delta.\mathtt{N}\ ]\!] & = & \bigcup_{\sigma,\ with\ \Delta\vdash\sigma} [\![\ \sigma(\mathtt{N})]\!] \end{array}$$

[4]We can easily extend our type interpretation to open types by introducing as usual environments mapping type variables to closed types. We avoid this generalisation since we can define subtyping for open types using only the interpretation of closed types, see Definition 3.

Since our semantic model is only defined for closed types, we handle type variables in the original syntactic types by considering all of their closed instantiations (which is after all a reasonable way to understand the meaning of a subtyping derivation containing free-variables - it is a schema for all such concrete cases).

Definition 3 (Semantic Subtyping). *Semantic subtyping is just subset inclusion on semantic types:*

$$S \subseteq S'$$

Semantic subtyping on syntactic *types (notation* $\Delta \models T \leq T'$*) is defined as follows:*

$$\Delta \models T \leq T' \qquad if \qquad \forall\sigma\ st.\ \Delta \vdash \sigma : [\![\sigma(T)]\!] \subseteq [\![\sigma(T')]\!]$$

Using these definitions, we can now define a semantic type system for our language.

Definition 4 (Semantic Type Assignment). *The semantic type assignment judgement* $\Gamma \models e : S$ *is defined by the rules in Fig. 4, in which when* $mType(m,\ N) = \overline{U} \to U$ *we define* $mType(m,N)\downarrow_1 = \overline{U}$ *and* $mType(m,N)\downarrow_2 = U$.

4. CORRESPONDENCE RESULTS

Our original motivation for studying a semantics model of Java types was to investigate completeness of subtyping, that is, whether or not all potential subtype relationships between types are captured by Java subtyping. This question is difficult because subtyping for wildcard types is complex and there is no obvious intuition to compare with (as is the case with plain class types: inheritance gives subtyping). In

this section we investigate the questions of soundness and completeness of both subtyping and typing, with respect to our corresponding semantic notions.

The most interesting result of this section is that Java subtyping is *not* complete; we give a counter-example below. We also prove a partial completeness result: Java subtyping is complete for a subset of Java types which we identify; it is future work to investigate whether completeness can be extended to larger subsets.

4.1 Soundness of Syntactic Typing

We investigate first whether the syntactic notions of subtyping and typing (based on those of Java), are *sound* with respect to our semantics. Since Java with wildcards is sound [4], one would hope this to be the case, and in fact we do answer these questions affirmatively. In the case of subtyping, we are able to show that all subtypings derivable in Java are sound with respect to our semantic notion of subtyping.

THEOREM 1 (SOUNDNESS OF SUBTYPING). *For all types T and T', and for all environments Δ, if $\Delta \vdash T <: T'$, then $\Delta \models T \leq T'$.*

Armed with this result, we can show further that all type derivations in our syntactic formulation of type assignment are *sound* with respect to the semantic notion of type assignment of Definition 4.

THEOREM 2 (SOUNDNESS OF TYPE ASSIGNMENT). *For all Δ,Γ,e and T, if $\Delta;\Gamma \vdash e : T$, then, for all substitutions σ such that $\Delta \vdash \sigma$, we have $\sigma(\Gamma) \models \sigma(e) : [\![\sigma(T)]\!]$.*

Since our syntactic type system is a generalisation of that used for Java (allowing more general bounds on existential quantifiers), this result also implies that all Java typings (for a suitably restricted sub-language of Java) are sound with respect to our semantic model of types.

4.2 Incompleteness of Syntactic Subtyping

As well as addressing soundness, it is natural to ask whether the Java type system is *complete*, in the sense that all of the type assignments which are valid semantically are also derivable in the syntactic system. This property is not completeness of type checking in the classical sense (i.e., that all programs which execute without errors will type check), but that all programs which will type check under a semantic model, type check in the syntactic model. Central to this question is the treatment of existential types. In the semantic model, such types are interpreted as the enumeration of all possible closed instances of the type quantified over, which are then considered independently when checking semantic subtypings. Could it be that the apparently-refined notion of semantic subtyping produces a more precise notion of subtyping? In fact, this is the case, as the following example shows.

EXAMPLE 1 (SUBTYPING IS INCOMPLETE). *Consider the types T_1 = `C<D>` and T_2 = `∃X:[D D].C<X>`. Then $\models T_2 \leq T_1$, but it is not the case that $\vdash T_2 <: T_1$.*

In particular, in the syntactic system we have no way of making use of the "knowledge" that there is only one way in which a pair of bounds could ever be satisfied. However, since this example relies on the use of syntactically identical lower and upper bounds for an existentially-bound variable, one might think that completeness of syntactic subtyping could be achieved by adding a simple extra rule such as

$$\frac{\Delta \vdash B <: U \quad \Delta \vdash U <: B}{\Delta \vdash \exists X : [B\ U].T <: [U/X]T} \text{(EQ)}$$

However, this is not sufficient in the general case. For example, the types T_3 = `∃X:[⊥ Y], Y:[X Object].C<X,Y>` and T_4 = `∃Z:[⊥ Object].C<Z,Z>` are semantically equivalent, because, to satisfy both sets of bounds, `X` and `Y` must be instantiated in the same way. Still, the judgment $\vdash T_3 <: T_4$ cannot be obtained. On the other hand, application of S-EXISTS-R and S-EXISTS-L gives $\vdash T_4 <: T_3$.

We do not believe that additional subtyping rules could help handle this kind of examples. Note, however, that a type like T_3 is not expressible in Java. This motivates our search for restrictions under which syntactic subtyping would be complete.

4.3 Completeness for Rich Types

Having demonstrated that syntactic subtyping is, in general, incomplete, in this section, we identify a syntactic restriction on types and environments, under which syntactic subtyping is both sound and complete. Our partial completeness result (Theorem 3) applies only to closed types, and under the restriction that the types are *rich*; a concept we define below (Definition 6). The full technical development of these results required other technical concepts and many lengthly proofs, which are omitted here for space reasons. We first generalise the notion of agreement between an environment and a closed substitution to open substitutions, employing a second environment to deal with the free variables introduced by the substitution.

DEFINITION 5. *$\Delta;\Delta' \vdash \overline{[U/Y]}$ if $dom(\Delta') = \overline{Y}$ and for all $Y_i \in dom(\Delta')$ we have $\Delta \vdash B_l\overline{[U/Y]} <: U_i <: B_u\overline{[U/Y]}$, where $[B_l\ B_u] = \Delta'(Y_i)$.*

The motivation for rich types is a technical property required for our completeness proof: given two types T and T', whose free variables are $\overline{X}$ and $\overline{Y}$ respectively, if whenever we replace $\overline{X}$ using a substitution σ, we can find a corresponding substitution σ' on $\overline{Y}$ such that the resulting types are equal ($\sigma(T) = \sigma'(T')$), then we need it to follow that there exist $\overline{U}$ (which can contain occurrences of variables in $\overline{X}$) such that $T = T'\overline{[U/Y]}$. This amounts to saying that the substitutions σ' can all be decomposed into a common part which changes T' to agree with T on the $\overline{X}$, followed by the appropriate substitutions for these variables.

DEFINITION 6. *$\Delta, T \ltimes \Delta', T'$, if $dom(\Delta) \cap dom(\Delta') = \emptyset$ and for all σ s.t. $\Delta \vdash \sigma$, there exists a σ' s.t. $\Delta' \vdash \sigma'$ and $\sigma(T) = \sigma'(T')$.*
An environment Δ is rich *if for all Δ', T, T', if $\Delta, T \ltimes \Delta', T'$, then there exist $\overline{U}$, $\overline{Y}$ s.t. $\Delta;\Delta' \vdash \overline{[U/Y]}$ and $T = T'\overline{[U/Y]}$.*
A type T is rich, *if $T = \exists\Delta.N$ and Δ is rich.*

It is obvious that all closed class types are rich; thus the type T_1 = `C<D>` is rich. We consider the remaining types from Section 4.2: T_4 = `∃Z:[⊥ Object].C<Z,Z>` is rich. The environment Δ_2 = `X:[D D]` is *not* rich – consider $\Delta'_2 = \emptyset$, and T=`X` and T'=`D`. Therefore, the type T_2 = `∃`Δ_2`.C<X>` is not rich. Similarly, Δ_3 = `X:[⊥ Y], Y:[X Object]` is not rich – consider Δ_4 = `Z:[⊥ Object]`, and T = `C<X,Y>` and

$T' = \texttt{C<Z,Z>}$. Therefore, the type $T_3 = \exists\Delta_3.\texttt{C<X,Y>}$ is not rich either.

In order to show that completeness holds for closed, rich types we require the following auxiliary lemma, which guarantees that substitution commutes with subclassing.

LEMMA 1. *If* $\vdash \sigma(N) \sqsubseteq N'$ *and* $\Delta \vdash \sigma$, *then there is* N'' *such that* $\Delta \vdash N \sqsubseteq N''$ *and* $N' = \sigma(N'')$. *Moreover,* N'' *only depends on* N *and* N', *and not on* σ.

THEOREM 3 (WEAK COMPLETENESS).
For all closed types T_1 *and* T_2, *if* $\models T_1 \leq T_2$ *and* T_1 *is rich, then* $\vdash T_1 <: T_2$.

PROOF. Let $T_1 = \exists\Delta_1.N_1$ and $T_2 = \exists\Delta_2.N_2$ with $\overline{U} = dom(\Delta_2)$. By definition of semantic interpretations and subtyping, we obtain that
$\bigcup_{\Delta_1 \vdash \sigma_1}\{\, N_3 \mid N_3 \sqsubseteq \sigma_1(N_1)\,\} \subseteq \bigcup_{\Delta_2 \vdash \sigma_2}\{\, N_4 \mid N_4 \sqsubseteq \sigma_2(N_2)\,\}$
Therefore, for all σ_1, N_3, s.t. $\Delta_1 \vdash \sigma_1$, and $N_3 \sqsubseteq \sigma_1(N_1)$ there exists a substitution σ_2 s.t. $\Delta_2 \vdash \sigma_2$ and $N_3 \sqsubseteq \sigma_2(N_2)$.

By weakening the above, we get that for all σ_1 s.t. $\Delta_1 \vdash \sigma_1$ there exists a σ_2 s.t. $\Delta_2 \vdash \sigma_2$ and $\sigma_1(N_1) \sqsubseteq \sigma_2(N_2)$.

By Lemma 1 there exists a N_5 s.t. $\Delta_1 \vdash N_1 \sqsubseteq N_5$ and for all σ_1 with $\Delta_1 \vdash \sigma_1$ there exists a σ_2 with $\Delta_2 \vdash \sigma_2$ such that $\sigma_2(N_2) = \sigma_1(N_5)$.

Because Δ_1 is a rich environment, and by Def. 6, there exist $\overline{U}$ such that $\Delta_1; \Delta_2 \vdash \overline{[U/Y]}$ and $N_5 = N_2\overline{[U/Y]}$.

From this, and rule S-EXISTS-R we can derive $\Delta_1 \vdash N_5 <: \exists\Delta_2.N_2$.

Furthermore, from $\Delta_1 \vdash N_1 \sqsubseteq N_5$ we obtain that $\Delta_1 \vdash N_1 <: N_5$.

Application of S-TRANS on the last two results gives $\Delta_1 \vdash N_1 <: \exists\Delta_2.N_2$.

Finally, application of S-EXISTS-L rule yields the desired result: $\vdash \exists\Delta_1.N_1 <: \exists\Delta_2.N_2$. □

For example, the type $T_1 = \texttt{C<D>}$ is rich, while the type $T_2 = \exists\texttt{X:[D D]}.\texttt{C<X>}$ is not. Furthermore, $\models T_1 \leq T_2$, and $\models T_2 \leq T_1$. In holding with our theorem $\vdash T_1 <: T_2$ can be derived (by application of S-EXISTS-R, and the substitution $[\texttt{D/X}]$), while $\vdash T_1 <: T_2$ can *not* be derived.

For a more interesting example, consider types $T_5 = \exists\texttt{V:[⊥ Object]}.\texttt{F<V>}$, and $T_6 = \exists\texttt{V:[A Object]}.\texttt{F<V>}$, and $T_7 = \exists\texttt{U:[}T_5\ T_6\texttt{]}.\texttt{E<U>}$. Then, T_7 is *not* rich, although T_5, T_6 *are*. Type $T_8 = \exists\texttt{Y:[⊥ A]}.\texttt{E<}\exists\texttt{X:[Y Object]}.\texttt{F<X>>}$ *is* rich. Furthermore, T_8 and T_7 are semantically equivalent, and $\vdash T_8 <: T_7$ can be derived, while $\vdash T_7 <: T_8$ can *not* be derived. Note that T_7 is denotable, while T_8 is expressible but not denotable in Java.

Since the notion of rich types is stated rather indirectly, we now identify a more syntactic criterion on environments which guarantees them to be rich. The key idea is (in the language of Definition 6) that Δ must allow different choices of σ to take "sufficiently different" choices for the types which replace the variables. In particular, if it is possible to instantiate each of the type variables belonging to $dom(\Delta) = \overline{X}$ in ways which differ completely in their structure, then the only way for the corresponding choices for $\overline{Y}$ (made in σ') to manage to account for this is for the occurrences of $\overline{Y}$ in T' to "sit above" the occurrences of $\overline{X}$ in T. This then allows for the decomposition described - we can replace the $\overline{Y}$ in a way which positions the occurrences of $\overline{X}$ in the same way as in T (the substitution $\overline{[U/Y]}$ achieves this). In order to formalize the notion of "sufficiently different" choices for the variables, we first define what it means for two types to be "sufficiently different":

DEFINITION 7. *Two types* T *and* T' *are* sufficiently different *(notation* $T \sharp T'$*) if:*
$T = \exists\Delta.C < \ldots >$, $T' = \exists\Delta'.C' < \ldots >$ *and* $C \neq C'$.

DEFINITION 8. *An environment* Δ *is* weakly independent *if for all* $X \in dom(\Delta)$ *there exist* σ_1 *and* σ_2 *such that*

1. $\Delta \vdash \sigma_1$
2. $\Delta \vdash \sigma_2$
3. $\sigma_1(X) \sharp \sigma_2(X)$
4. $\sigma_1(Y) = \sigma_2(Y)$ *for all* $Y \in dom(\Delta)$ *with* $Y \neq X$.

We use the term *weak* independence because we do not require that each variable in the environment can *always* be instantiated independently of the instantiations of the others. Instead, we make the weaker requirement (for each variable X) that there exists *at least one* instantiation of the environment in which the choices for the other variables do not determine the choice for X (cf. σ_1 and σ_2 above). For example, $\Delta_9 = \texttt{X:[A Object], Y:[X Object]}$ is weakly independent, since we can (for example) choose either to substitute `A` for `X` and `Object` for `Y`, or `Object` for both `X` and `Y`.The environment $\Delta_{10} = \texttt{X:[⊥ Object], Y:[C<X> C<X>]}$ is not weakly independent, since `Y`'s substitution is determined by that of `X`. Furthermore, the two environments Δ_2=`X:[D D]` and Δ_3=`X:[⊥ Y], Y:[X Object]` are not weakly independent.

One might wonder if (as in all our examples so far), environments which are not weakly independent involve nontrivial lower bounds (i.e., bounds different from ⊥). This question is relevant for the application of our results to Java, since there it is relatively difficult to denote types with both complex upper and lower bounds. However, the existence of mutually-dependent bounds is also enough to generate interesting examples of environments which are not weakly independent. As suggested in [11], expressing the parallel class hierarchies involved in the `Subject` and `Observer` pattern requires mutually recursive bounds, as in:
`class Subj<X ◁ Subj<X,Y>, Y ◁ Obs<X,Y>> {...}`,
and `class Obs<X ◁ Subj<X,Y>, Y ◁ Obs<X,Y>> {...}`.
Considering the constraints on the class parameters, the environment Δ_{11} = `X:[⊥ Subj<X,Y>], Y:[⊥ Obs<X,Y>]` is not weakly independent. Moreover, not only are the types $T_9 = \exists\Delta_{11}.\texttt{Subj<X,Y>}$ and $T_{10} = \exists\Delta_{11}.\texttt{Obs<X,Y>}$ denotable in Java, but they correspond to the wildcards version of the two classes involved in the `Subject` and `Observer` pattern.While the lack of weak independence rules out the application of our main results, we do not yet know whether this example actually yields a counter-example to completeness of subtyping - so far we have not been able to find such a counter-example using these classes.

The following result shows that the concept of weakly independent environments is sufficient for weak completeness.

THEOREM 4. *Any weakly independent environment is rich.*

Note that all examples in this section which were not rich were not weakly independent either. We have not encountered any examples of rich environments which were not weakly independent, and in further work we plan to try to prove that the opposite direction of Theorem 4 holds.

5. CONCLUSIONS AND FUTURE WORK

In this paper we have outlined our research on semantic models for Java types and our steps toward defining and proving a completeness theorem for Java subtyping and type checking. Ours is the first study on semantic foundations to take into account Java wildcards. Such a result is challenging to obtain, not only because the type language of Java is very expressive and incorporates technically challenging features such as F-bounded polymorphism, but also because the type system *underlying* the language is neither formally specified nor semantically well understood. A significant contribution of this work is to provide an approach to clarifying the semantics and properties of the underlying type system. To extend this contribution further, we aim to incorporate open, as well as closed types, for our completeness result, for which we already have much of the necessary machinery.

We have shown that our syntactic model of Java types is incomplete with respect to our semantics, and have identified areas of future work in the direction of extending our completeness results, and investigating whether counterexamples to full completeness can be expressed in Java. In particular, we have not yet found a non-trivial (i.e., without employing identical upper and lower bounds) example which can be denoted directly in the Java language. If such examples do not occur (as we conjecture), we may yet be able to extend our weak completeness result to a completeness result (modulo type equivalences) for the full Java type system.

Acknowledgements

We thank the anonymous FTfJP reviewers for feedback and good suggestions. Sophia and Nick are grateful to Dave Clarke for awakening their interest in semantic approaches to typing.

6. REFERENCES

[1] H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

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

[3] N. Cameron and S. Drossopoulou. On Subtyping, Wildcards, and Existential Types. In *FTfJP'09*, pages 1–7, New York, NY, USA, 2009. ACM.

[4] N. Cameron, S. Drossopoulou, and E. Ernst. A Model for Java with Wildcards. In *ECOOP'08*, number 5142 in LNCS, pages 2–26, Berlin / Heidelberg, 2008. Springer.

[5] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[6] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types. *Journal of the ACM*, 55(4):1–64, 2008. Extends and supersedes LICS '02 and ICALP/PPDP '05 articles.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.

[8] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001. An earlier version appeared at OOPSLA'99.

[9] A. Igarashi and M. Viroli. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, 2006. An earlier version appeared as "On variance-based subtyping for parametric types" at ECOOP'02.

[10] J. C. Mitchell and G. D. Plotkin. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. An earlier version appeared at POPL '85.

[11] M. Naftalin and P. Wadler. *Java Generics and Collections*. O'REILLY, Sebastopol, Ca.,USA, 2007.

[12] B. C. Pierce. Bounded quantification is undecidable. In *POPL '92*, pages 305–315, New York, NY, USA, 1992. ACM.

[13] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] M. Plümicke. Typeless Programming in Java 5.0 with Wildcards. In *PPPJ'07*, pages 73–82, New York, NY, USA, 2007. ACM.

[15] A. J. Summers. Modelling Java requires State. In *FTfJP'09*, pages 1–3, New York, NY, USA, 2009. ACM.

[16] M. Torgersen, E. Ernst, and C. P. Hansen. Wild FJ. In *FOOL 12*, 2005. http://homepages.inf.ed.ac.uk/wadler/fool/program.

[17] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahé, G. Bracha, and N. 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.

[18] M. Viroli and G. Rimassa. On Access Restriction with Java Wildcards. *Journal of Object Technology*, 4(10):117–139, 2005. Special issue: OOPS track at SAC 2005, Santa Fe/New Mexico. An earlier version appeared as "Understanding access restriction of variant parametric types and Java wildcards" at SAC 2005.

[19] S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized Interfaces for Java. In *ECOOP'07*, number 4609 in LNCS, pages 347–372, Berlin / Heidelberg, 2007. Springer.

[20] S. Wehr and P. Thiemann. On the decidability of subtyping with bounded existential types. In *APLAS'09*, number 5904 in LNCS, pages 111–127, Berlin / Heidelberg, 2009. Springer.