Wildcards Need Witness Protection: Extended Version*

KEVIN BIERHOFF, Google, USA

In this paper, we show that the unsoundness discovered by Amin and Tate (2016) in Java's wildcards is avoidable, even in the absence of a nullness-aware type system. The key insight of this paper is that soundness in type systems that implicitly introduce existential types through subtyping hinges on still making sure there are suitable witness types when introducing existentially quantified type variables. To show that this approach is viable, this paper formalizes a core calculus and proves it sound. We used a static analysis based on our approach to look for potential issues in a vast corpus of Java code and found *none* (with 1 false positive). This confirms both that Java's unsoundness has minimal practical consequence, and that our approach can avoid it entirely with minimal false positives.

CCS Concepts: • **Software and its engineering → Object oriented languages**; **Polymorphism**.

Additional Key Words and Phrases: Java Wildcards, Null, Existential Types

ACM Reference Format:
Kevin Bierhoff. 2022. Wildcards Need Witness Protection: Extended Version. In *OOPSLA 2022*. ACM, New York, NY, USA, 32 pages.

1 INTRODUCTION

Amin and Tate [2016] showed that Java's and Scala's type systems are unsound in the presence of `null` values. Somewhat remarkably, this discovery appears to have been rooted in research on Java's wildcards, rather than nullness, even though nullness has been the subject of both academic and industry attention for quite some time. Nonetheless, Scala is reportedly using the nullness awareness being introduced into its type system to avoid the discovered unsoundness [Nieto et al. 2020].

While Amin and Tate [2016] informally suggest that restrictions on wildcards could address the issue, to the best of our knowledge, no comprehensive proposal or formal proof of a suggested fix for Java exist. Moreover, as we will see, restrictions on wildcards would likely have occasional false positives in existing Java code (i.e., would occasionally rule out code occurring in practice that is in fact safe). And while it seems plausible that an approach similar to Scala's could be taken for Java, introducing a statically checked nullness typing discipline into Java is clearly a monumental undertaking with potentially wide-ranging implications for existing code. Java's unsoundness also raises the question whether Kotlin's mixed-site variance [Tate 2013], which allows patterns akin to Java wildcards, might be vulnerable to similar threats to soundness.

In this paper, we show that the discovered unsoundness in Java's wildcards is avoidable, even in the absence of a nullness-aware type system and with fewer restrictions than previous proposals.

The key insight of this paper is that soundness in type systems that implicitly introduce existential types through subtyping hinges on still making sure there are suitable witness types when introducing existentially quantified type variables, even when the expression being typed is the `null` literal. This can be achieved in at least 3 ways: (1) by making sure that all existential types

*This version includes supplemental material for the paper presented at OOPSLA 2022 and appearing in Proceedings of the ACM on Programming Languages, Volume 6, Number OOPSLA2, Article 138, https://doi.org/10.1145/3563301.

```java
public class UnsoundDerived {
  static class Constrain<Y extends CharSequence> {}
  static <Y extends CharSequence>
  CharSequence upcast(Constrain<Y> c, Y y) {
    return y;
  }
  static <X> CharSequence coerce(X x) {
    Constrain<? super X> c = null;
    return upcast(c, x);
  }
  public static void main(String[] args) {
    CharSequence zero = coerce(0);
  }
}
```

Fig. 1. Unsound Java program derived from Amin and Tate [2016, fig. 4]

allowed in the system are guaranteed to have witness types by construction, (2) by leveraging a nullness type system to restrict nullable existential types, or (3) by placing restrictions on how `null` literals can be typed so that they can only introduce valid existential types. As we shall see, Kotlin is in the first camp, suggesting that Kotlin does not suffer from the same unsoundness as Java. Scala is as mentioned taking the second route. But as we will see, it's surprisingly simple to restrict the typing of `null` values such that only valid existential types can be introduced, which would allow taking the third option for Java.

To show that this approach is viable, this paper formalizes a core calculus similar to FGJ [Igarashi et al. 2001] and TameFJ [Cameron et al. 2008] enriched with `null` values. The type system restricts the typing of `null` values (and imposes a few related restrictions) to guarantee that existential types always have witness types. We prove type safety to demonstrate that the imposed restrictions are sufficient to guarantee soundness.

We implemented a static analysis suggested by our formalization as a linear AST scan that runs after typechecking. We used it to look for potential issues stemming from Java's unsoundness in a vast corpus of several hundred million lines of Java code. We found a single false positive and no actual issues stemming from Java's unsoundness in the code we analyzed. This suggests that the tweaks needed to avoid Java's unsoundness have minimal impact on existing code. In our view, that's a very promising result compared to potential alternative fixes through the introduction of nullness types or through banning of potentially problematic wildcard types, whose occurrence in practice this paper shows to be rare but frequent enough to be concerning considering the vast amount of Java code in existence.

The remainder of this paper is organized as follows: Section 2 reviews Java's unsoundness and wildcards. Section 3 gives a high-level intuition for what goes wrong in Java's type system and a proposal for how to fix it. Section 4 formalizes the proposed solution in a core calculus that's proven sound. Relevant proofs are included as supplemental material. Section 5 evaluates implications for real-world Java code and Kotlin. Section 6 summarizes related work and Section 7 concludes.

2 BACKGROUND

Java's unsoundness was discovered several years ago by Amin and Tate [2016]. Figure 1 shows an example derived from theirs. This particular example is valid Java, but it's worth noting that recent versions of the Java compiler, `javac`, nonetheless reject it. However, Amin and Tate [2016] present a series of more complicated variations on this program, some of which the Java 17 compiler still accepts. We will stick with the basic example to simplify the exposition and return to why the compiler rejects the basic example—but not slight variations of it—later.

The program in figure 1 ends up typing the boxed 0 given in line 12 as both `Integer` and, once returned as-is from the `coerce` function, as `CharSequence`. When executing this program (or rather, one of the dressed-up versions `javac` accepts), it fails with a `ClassCastException` on line 12. But that shouldn't be possible! There are no explicit casts in this program, hence why this example demonstrates Java's unsoundness.

Before we dive into how this example—to Java's chagrin—"works", we'll briefly review the feature this example critically relies on, Java's *wildcards*. Wildcards and their challenges have been studied extensively [Cameron et al. 2008; Smith and Cartwright 2008; Tate et al. 2011], so we'll focus on the features relevant for this paper here. We'll also review *existential types* and how they relate to wildcards, since we'll use existential types throughout this paper.

2.1 Wildcards and Existential Types

Wildcards are Java's way of encoding variance in parametric types. Java doesn't consider `List<Integer>` to be a subtype of `List<Number>`, because it treats type arguments as invariant by default. But that's inconvenient when wanting to write a function that iterates any list of numbers, for instance. Enter wildcards: the type `List<? extends Number>` stands for some parametric type `List<...>` where we additionally know that the elided type argument is a subtype of `Number`. Thus, `List<Integer>`, `List<Double>`, and `List<Number>` are all considered subtypes of `List<? extends Number>`, which therefore encodes covariance in the type argument.

Similarly, `List<? super Number>` stands for lists of some supertype of `Number`, encoding contravariance, which is of course convenient when implementing a function that appends numbers to a list, for instance.

This however presents new challenges. For one, the compiler has to keep in mind that two expressions typed `List<? extends Number>` may represent different types even though they're syntactically the same. That is to say, each wildcard (?) that occurs in the program text must be considered distinct from all others. Additionally, we want to pass our `List<? extends Number>` to generic functions, such as `sorted`, that, when given a `List<T>`, return a list of the same type.

Java provides a mechanism called *capture conversion* [Gosling et al. 2021, ch. 5.1.10] that allows invoking `sorted` with a `List<? extends Number>` and knowing that the returned list has the *same* unknown type (allowing argument and result to be concatenated next, for instance). Capture conversion for our purposes means that the Java typechecker internally allocates fresh type variables for each wildcard being captured and then equates them with the type parameters of the method being invoked. That allows (1) calling `sorted` in the first place and (2) knowing that the call's argument and result types are the same.

Wildcard types are typically seen as akin to existential types in the literature [Cameron et al. 2008]. For instance, in the $F_{<:}$-like calculus presented by Pierce [2002, ch. 26.5], the covariant list type from above would be written as something like $\exists X <: \texttt{Number}.\texttt{List}\langle X\rangle$. Unlike Java wildcards, conventional existential types have explicit introduction and elimination forms, often called `pack` and `unpack`, respectively. The former explicitly specifies a *witness type* to be abstracted. For instance, we'd specify `Integer` as the witness type when packing a value of type $\texttt{List}\langle\texttt{Integer}\rangle$ to the existential type above. The elimination form, written something like $\texttt{unpack}\ X, x = \ldots \texttt{in} \ldots$, gives access to the wrapped value using x and also allows referring to the witness type using the declared type variable, X, in its body.

Java wildcards, by contrast, have neither explicit introduction nor elimination forms. Instead, wildcards are introduced implicitly through subtyping as alluded to above and eliminated implicitly through capture conversion. The latter as mentioned introduces fresh type variables, but unlike in `unpack` expressions, the program text can't refer to the type variables that result from Java's capture conversion.

Another difference to existential types in $F_{<:}$ as Pierce [2002] presents them is that Java wildcards have both an upper bound and a lower bound. Very commonly, one or both of these bounds are *trivial*, i.e., `Object` and $\bot$ ("bottom"), respectively. For instance, the wildcard in `List<? super Number>` has the (non-trivial) lower bound `Number` and the (trivial) upper bound `Object`, and `List<?>` has both trivial upper and lower bounds.

It's worth noting that Java wildcards with non-trivial lower bound, as the previous example, cannot explicitly also declare an upper bound. But all wildcards implicitly inherit any upper bounds declared on the type parameter they stand in for (called "implicit constraint" in Tate et al. [2011]). For instance, if we assume that `SortedSet`'s type parameter is declared to have the upper bound `Comparable` then the wildcard in `SortedSet<? super Number>` also has that upper bound, and additionally the explicitly declared lower bound, `Number`.

In this last example, the explicit lower bound, `Number`, is a known subtype of the implicit upper bound, `Comparable`. But it raises the question: can we encode wildcards in Java where the lower bound isn't apriori known to be subtype of the upper bound? Indeed this is possible, though (because of various restrictions enforced in the Java compiler that we detail in Section 5.1) only if the lower bound is a type variable. For instance, the type `Constrain<? super X>` in figure 1 (line 8) is such a type: the type variable, X, is *not* known to be a subtype of the wildcard's implicit upper bound, `CharSequence` (declared in line 2).

We will refer to such a wildcard as *subtype-asserting*: the wildcard establishes a subtype relationship between its lower and upper bounds that is not otherwise known in the context where the wildcard appears. Critically, capture conversion makes the asserted subtyping relationship available during subsequent typechecking, by assuming appropriate constraints between the introduced fresh type variables and their bounds. Assumably, Java allows this because subtype-asserting wildcards can still stand in for valid types. In this example, `String` as well as `CharSequence` itself are such types. As we will see, subtype-asserting wildcards are *not* unsound per se, but their presence is necessary to run afoul of Java's unsoundness.

2.2 Java's Unsoundness

With these preliminaries in mind we can turn our attention to how the program in figure 1 is able to trick the Java type system. As already discussed, the wildcard in line 8 implicitly claims a subtype relationship between the method's type parameter, X, and `CharSequence`. The clever use of a `null` literal to initialize the wildcard type makes it possible for the code not to have to "prove" that X is actually a subtype of `CharSequence`, since Java allows assigning `null` to any class type [Gosling et al. 2021, ch. 4.10.2] (as do many programming languages).

The code on line 9 makes the illusion complete: Java performs capture conversion of the wildcard type to call `upcast`. Doing so introduces a fresh type variable, let's say, W, which, critically, is assumed to be a supertype of X. Therefore, we can upcast `coerce`'s parameter x from its declared type X to W to call `upcast`. But because W is also assumed to be a subtype of `CharSequence` as discussed, `upcast`'s method body is now able to upcast the given value to `CharSequence`.

Note that the wildcard-typed variable, c, is never dereferenced, but its mere presence suffices to confuse Java: calling `coerce` with an `Integer` object as in line 12 of figure 1 allows assigning the given number to a `CharSequence`, which causes an unexpected `ClassCastException` at runtime. Of course, if we called `coerce` with a `String` object, everything would be fine.

As mentioned, recent versions of `javac` would reject the program in figure 1. The reason is that `javac`'s typechecking of wildcards is incomplete. Amin and Tate [2016] discuss this issue at length and provide variants of the basic example shown here that are accepted by recent versions of the Java compiler. One of the examples [Amin and Tate 2016, fig. 3] is still accepted by the compiler included in the latest LTS release of Java, Java 17, according to our manual experiments.[1]

Potential solutions informally discussed in previous papers include:

(1) Amin and Tate [2016, sect. 7] themselves informally discuss several ideas. One concrete idea they mention is to ban wildcards where both lower and (implicitly inherited) upper bounds are non-trivial. This approach appears sound, as it in particular rules out subtype-asserting wildcards as a special case (Section 5). However, the data presented in Section 5.1 shows that this approach would unnecessarily break existing code with worrisome frequency.
(2) Nieto et al. [2020] indicate that they intend to address Scala's unsoundness by relying on the nullness type system for Scala they describe. They give no proof or further details of any restrictions they intend to impose. While we believe a similar approach could be taken for Java, it would clearly be a monumental effort to introduce sound nullness checking into Java. Even leaving that aside, the details might matter. For instance, requiring non-null types wherever wildcards are used would, based on our data, likely be quite disruptive in practice (see Section 5.1). As we will see shortly, our approach not only doesn't need nullness types but is also technically more precise than using nullness types.

We're not aware of previous soundness proofs for either approach. As we will discuss in more detail, it appears that TameFJ (possibly "by accident") employs a more targeted variant of the former approach—ruling out subtype-asserting wildcards specifically instead of all wildcards with non-trivial upper and lower bounds—but TameFJ didn't consider `null` [Cameron et al. 2008].

3 AVOIDING JAVA'S UNSOUNDNESS

In this section we give a high-level intuition for the solution proposed in this paper. The previously proposed approaches discussed above make intuitive sense. But they don't really seem to explain what's going wrong in Java's type system, arguably treating symptoms instead of addressing the root cause of the problem.

In this paper we argue that Java's unsoundness is rooted in its subtyping rules. As alluded to in the previous section, Java assigns `null` literals the non-denotable "null type", which Java defines to be a subtype of all reference types [Gosling et al. 2021, ch. 4.10.2], including types containing wildcards, making the "null type" synonymous to $\bot$.

The key insight of this paper is that the "null type" shouldn't automatically be thought of as a direct subtype of types containing subtype-asserting wildcards. In other words, Java shouldn't type `null` as $\bot$.

We arrive at this conclusion by likening wildcard types to existential types [Pierce 2002]. Specifically, we observe that in Java, introduction and elimination of wildcard types happen implicitly through subtyping [Gosling et al. 2021, ch. 4.10.2]:

- Wildcards are *introduced* by wildcard type arguments "containing" the subtype's type arguments [Gosling et al. 2021, ch. 4.5.1]. For instance, Java considers `List<Integer>` to be a subtype of `List<? extends Number>` transitively in two steps: first, `? extends Number` "contains" `? extends Integer` by virtue of `Integer` being a subclass of `Number`. Second, `? extends Integer` contains `Integer` because wildcards are defined to contain their own bounds.
- Wildcards are *eliminated* by capture conversion [Gosling et al. 2021, ch. 5.1.10] as discussed in the previous section.

[1]We used Oracle's jdk-17.0.1 release for Mac to test how `javac` handles prolematic inputs such as figure 1.

These observations lead us to comparing Java's subtyping rules to how existential types are *explicitly* introduced and eliminated in traditional type theory. In particular, we would write something like the following in a hypothetical calculus that enriches FGJ [Igarashi et al. 2001] with an explicit introduction form for existential types [Pierce 2002, ch. 26.5]:

$$\{*\texttt{Integer}, \texttt{new List}\langle\texttt{Integer}\rangle()\} \texttt{ as } \exists X <: \texttt{Number.List}\langle X\rangle$$

Here, we *explicitly* tell the typechecker that the desired existential type, $\exists X <: \texttt{Number.List}\langle X\rangle$, "hides" the specified *witness type*, `Integer`. This gives the typechecker the opportunity to check that the witness type satisfies the type variable's specified bound. Here, the typechecker would verify that `Integer` <: `Number`.

Java's containment rules for wildcard type arguments can be thought of as an algorithm that, given a parameterized type—such as `List<Integer>`—and a desired supertype that includes wildcards, automatically makes sure there's a witness type for each wildcard type argument in the supertype.

But by making the "null type" a subtype of any type, wildcards can essentially also be introduced "through the back door", *without* making sure there are valid witness types.

Let's consider the example from line 8 in figure 1 again, `Constrain<? super X> c = null`. If we required a check that made sure there's a witness type for `? super X`, that check would fail, because there is no witness type in scope that we could use. In particular, X itself can't be a witness: given its declaration in `coerce` it doesn't have the needed upper bound, `CharSequence`.

The "fix", then, is to make sure that wildcards are always properly introduced through subtyping, even when assigning `null`. This makes sure that all wildcards have witnesses. With this fix, `Constrain<? super X> c = null` in our example no longer type-checks, since as discussed above there is no witness type we could use. In the next section we show that this "fix" can indeed achieve soundness. We will revisit the examples from Amin and Tate [2016] with this calculus in figure 7.

It's instructive to realize that with this approach, assigning `null` to types containing wildcards is allowed where sound *even if there are subtype-asserting wildcards*. To see that, let's consider a slightly modified example where `coerce` declares the variable `c` as a second formal parameter, `Constrain<? super T> c`, instead of a local variable. Java would (correctly) allow invoking this modified method with `coerce("foo", new Constrain<String>())` (and would reject `new Constrain<Integer>()` outright because `Integer` isn't a subtype of `CharSequence` and hence not a valid type argument for `Constrain`). Now consider invocations where the second argument is `null` (which Java as-is permits regardless of the type of the first argument for the same reasons as before):

- `coerce("foo", null)` is fine, by the same reasoning as for `coerce("foo", new Constrain<String>()`
  Specifically, `String` is a valid witness for the parameter `c`'s wildcard type argument. Moreover, as we noted before, `c` is never dereferenced, so this code would execute without error, and deservedly so.
- `coerce(0, null)` on the other hand is not fine, as it would result in the same `ClassCastException` as before, and should therefore be rejected. Crucially, our approach *does reject* this invocation by reasoning similar to where `c` was a local variable as in figure 1: there is no witness type for `c`'s wildcard that is both a supertype of `Integer` and a subtype of `CharSequence`. Which is of course the reason why the code in figure 1 demonstrates that Java is unsound in the first place.

These examples have implications for systems with nullness types: *nullable types containing subtype-asserting wildcards can be sound!* Meaning, proposals to fix Java's unsoundness using nullness types (similar to what's being done to Scala, cf. Nieto et al. [2020]), even if limited to subtype-asserting wildcards, would be unnecessarily conservative and rule out correct programs

Table 1. Syntax

Parameterized classes	N	::=	$C\langle\overline{T}\rangle$
Types	S, T, U	::=	$X \mid \exists\Delta.N$
Lower bounds	K, L	::=	$T \mid \bot$
Class declarations	D	::=	$\texttt{class}\ C\langle\overline{X \lhd T}\rangle \lhd N\{\overline{T\ f};\ \overline{M}\}$
Method declarations	M	::=	$\langle\overline{X \lhd T}\rangle\ T\ m(\overline{T\ x}) = t$
Terms	s, t	::=	$x \mid \texttt{null}$
		|	$\textsf{new}\ C\langle\overline{T}\rangle(\bar{t})$
		|	$t.f$
		|	$t.\langle\overline{T}\rangle m(\bar{t})$
		|	$\texttt{let}\ x : \exists\Delta.N = t\ \texttt{in}\ t$
		|	$t\ ?{:}\ t$
Values	V	::=	$\texttt{null} \mid \textsf{new}\ C\langle\overline{T}\rangle(\overline{V})$
Type variable contexts	Δ	::=	$\overline{X : L..T}$
Variable contexts	Γ	::=	$\overline{x : T}$

permitted by our approach. Simply put, the reason for this difference is that nullable types can be `null` or non-`null` at runtime, while our approach focuses on restricting definitely-`null` values. Thus, our approach not only doesn't require a nullness type system—which we consider a key advantage in itself—but is also strictly more precise than restricting which types can be nullable.

To summarize, Java is unsound because Java's subtyping rules allow invalid wildcards to be introduced, namely, wildcards without witnesses. In the remainder of this paper we show that the issue is fixable by restricting the typing of `null` values. As we will see, it's additionally necessary for soundness to place restrictions on type arguments used in object instantiation and generic method calls, to avoid problems stemming from `null` values typed using a type variable. Our data suggests that these restrictions would have minimal impact on existing Java code: we only found a single (false positive) violation of these restrictions in several hundred million lines of Java code, despite use of subtype-asserting wildcards in the analyzed corpus (see Section 5.1).

4 FORMAL SYSTEM

In this section we formalize a calculus that adds Java-style existential types (with upper and lower bounds) and `null` to FGJ [Igarashi et al. 2001] and prove it sound. While technically complex, due to the combination of generics, existentials, and `null`, our typing rules are ultimately mostly standard and mostly identical to the existing literature, *except* for the typing of `null` and type arguments in object instantiations and generic method calls. Specifically, we require these types to be witnessed (cf. figure 5), which formalizes types known to have a witness type.

4.1 Syntax

The abstract syntax of our system is given in table 1 and similar to TameFJ [Cameron et al. 2008] and FGJ [Igarashi et al. 2001]. As usual, we abbreviate Java's `extends` keyword to $\lhd$. C ranges over class names including the distinguished `Object`. f and g range over field names, and m over method names. x, y, and z range over variable names including `this`, and X, Y, and Z range over type variables. We write $\overline{T}$ and similar for possibly-empty sequences $T_1, \ldots, T_n$.

Metavariables S, T, and U range over types; K and L range over lower bounds, which can be types or $\bot$. Variable contexts Γ are standard, while type variable contexts Δ assign type variables a lower and an upper bound, written $X : L..U$ (which is comparable to the notation $X \rightarrow [L\ U]$ in Cameron et al. [2008]). Class types are written $\exists\Delta.C\langle\overline{T}\rangle$, which bind additional type variables in Δ in $C\langle\overline{T}\rangle$'s type arguments, $\overline{T}$. At times we'll use N to range over $C\langle\overline{T}\rangle$ constructs. For notational convenience we write all class types as existential types that bind a possibly-empty type variable context. We will refer to existential types that bind a non-empty set of type variables as *proper* or *non-trivial* existential types.

Metavariables s and t range over terms. Terms include variables x, the `null` literal, `new` constructions, field dereferences, and method calls. Method calls include a possibly-empty list of type arguments to instantiate the method's type parameters. Additionally, terms include a `let` construct and Kotlin's "Elvis" operator, written $t_1 ?: t_2$, which can be used to discriminate `null` values from objects. For simplicity we don't include other similar operators such as safe calls (often written ?.) or if-null-else-like constructs, but we believe they could be added without difficulty. Values V include `null` as well as objects, i.e., `new` terms whose arguments are themselves values.

Our `let` $x : \exists\Delta.N = t_1$ `in` t_2 binds additional type variables Δ and a variable x in t_2 and is therefore similar to `unpack` expressions in conventional formulations of existential types [Pierce 2002]. Note that $\exists\Delta.N$ is a user-written type that would appear in the surface language. This construct makes explicit what the JLS calls "capture conversion" [Gosling et al. 2021, ch. 5.1.10]: the binding of a wildcard type to fresh type variables. Unlike TameFJ, our calculus consequently doesn't need to include the notion of polymorphic method type arguments (written $*$ in TameFJ) that have to be inferred by the type system, and there's also no need to unpack and capture existential types on the fly. This not only simplifies the formalization but also isolates soundness concerns stemming from wildcards to `let` bindings where t_1 evaluates to `null`. Note we allow the bound variable context Δ to be empty, which recovers conventional variable bindings: while they could be encoded as in FGJ, they're convenient to have around.

As in FGJ, programs in this calculus consist of a fixed *class table*—a list of class declarations D—and a "main" expression t that is type-checked under empty variable context. Class and method declarations are standard. We also require the conventional well-formedness conditions on the class table, including that all class names C—except `Object`—are uniquely defined in the class table and no class is transitively declared to extend itself.

We will use the notations $[\overline{t}/\overline{x}]$ and $[\overline{T}/\overline{X}]$ for customary capture-avoiding substitution of possibly-empty lists of terms and types, respectively. The following sections additionally rely on the helper functions summarized in figure 2 that formalize field lookup, method type, and method body lookup (leveraging type substitution) as in FGJ. For instance, $\mathtt{mtype}(m, C\langle\overline{T}\rangle)$ either returns m's type as declared in C or forwards the lookup to C's declared superclass. Either way, `mtype` determines m's declared type variables and their upper bounds as well as method parameter and return types that may refer to the declared type variables, with all types adjusted for the provided type arguments $\overline{T}$ by substituting them for C's declared type variables.

4.2 Dynamic Semantics

Figure 3 summarizes the small-step evaluation judgment, $t \longmapsto t'$. The Read and Invoke rules are standard from FGJ [Igarashi et al. 2001], and the congruence rules formalize conventional call-by-value semantics. Present and Absent give meaning to the "Elvis" operator by evaluating $V ?: t$ to t if $V =$ `null` and to V otherwise.

Unpack is possibly the most interesting rule in our calculus. Given a `let` binding for existential type $\exists\overline{X : L..U}.N$ whose first subterm is an object value, `new` $N'(\overline{V})$, the rule "guesses" a substitution $\overline{T}$ for the bound type variables $\overline{X}$ that satisfies the specified bounds as well as the subtyping condition $\vdash \exists.N' <: [\overline{T}/\overline{X}]\exists.N$. Evaluation then continues by substituting the unpacked object as well as the

M-Class
$$\frac{\texttt{class}\ C\langle\overline{X \lhd \_}\rangle \lhd N\{\overline{S\ f};\ \overline{M}\} \qquad \langle\overline{Y \lhd U'}\rangle\ U\ m(\overline{U\ x}) = t \in \overline{M}}{\mathsf{mtype}(m, C\langle\overline{T}\rangle) = [\overline{T}/\overline{X}]\langle\overline{Y \lhd U'}\rangle\overline{U} \to U \qquad \mathsf{mbody}(m\langle\overline{T'}\rangle, C\langle\overline{T}\rangle) = \overline{x}.[\overline{T'}/\overline{Y}][\overline{T}/\overline{X}]t}$$

M-Super
$$\frac{\texttt{class}\ C\langle\overline{X \lhd \_}\rangle \lhd N\{\overline{S\ f};\ \overline{M}\} \qquad m \notin \overline{M}}{\mathsf{mtype}(m, C\langle\overline{T}\rangle) = \mathsf{mtype}(m, [\overline{T}/\overline{X}]N) \qquad \mathsf{mbody}(m\langle\overline{T'}\rangle, C\langle\overline{T}\rangle) = \mathsf{mbody}(m\langle\overline{T'}\rangle, [\overline{T}/\overline{X}]N)}$$

$$\mathsf{fields}(\texttt{Object}\langle\rangle) = \emptyset$$

F-Class
$$\frac{\texttt{class}\ C\langle\overline{X \lhd \_}\rangle \lhd N\{\overline{S\ f};\ \overline{M}\} \qquad \mathsf{fields}([\overline{T}/\overline{X}]N) = \overline{U\ g}}{\mathsf{fields}(C\langle\overline{T}\rangle) = \overline{U\ g}, [\overline{T}/\overline{X}]\overline{S\ f}}$$

Fig. 2. Lookup Functions

Read
$$\frac{\mathsf{fields}(N) = \overline{T f}}{\texttt{new}\ N(\overline{V}).f_i \longmapsto V_i}$$

Present
$$\texttt{new}\ N(\overline{V})\ ?{:}\ t \longmapsto \texttt{new}\ N(\overline{V})$$

Absent
$$\texttt{null}\ ?{:}\ t \longmapsto t$$

Invoke
$$\frac{\mathsf{mbody}(m\langle\overline{T}\rangle, N) = \overline{x}.t_0}{\texttt{new}\ N(\overline{V}).\langle\overline{T}\rangle m(\overline{V'}) \longmapsto [\overline{V'}/\overline{x}, \texttt{new}\ N(\overline{V})/\texttt{this}]t_0}$$

Unpack
$$\frac{\begin{array}{c}\vdash \exists.N' <: [\overline{T}/\overline{X}]\exists.N \\ \vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T} \qquad \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U} \qquad \vdash \overline{T}\ \text{witnessed}\end{array}}{\texttt{let}\ x : \exists\overline{X : L..U}.N = \texttt{new}\ N'(\overline{V})\ \texttt{in}\ t_2 \longmapsto [\texttt{new}\ N'(\overline{V})/x][\overline{T}/\overline{X}]t_2}$$

Stub
$$\frac{\begin{array}{c}\vdash \exists.N' <: [\overline{T}/\overline{X}]\exists.N \\ \vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T} \qquad \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U} \qquad \vdash \exists.N', \overline{T}\ \text{witnessed}\end{array}}{\texttt{let}\ x : \exists\overline{X : L..U}.N = \texttt{null}\ \texttt{in}\ t_2 \longmapsto [\texttt{null}/x][\overline{T}/\overline{X}]t_2}$$

C-Field
$$\frac{t \longmapsto t'}{t.f \longmapsto t'.f}$$

C-New
$$\frac{t \longmapsto t'}{\texttt{new}\ N(\overline{V}, t, \overline{t}) \longmapsto \texttt{new}\ N(\overline{V}, t', \overline{t})}$$

C-Receiver
$$\frac{t \longmapsto t'}{t.\langle\overline{T}\rangle m(\overline{t}) \longmapsto t'.\langle\overline{T}\rangle m(\overline{t})}$$

C-Arg
$$\frac{t \longmapsto t'}{V.\langle\overline{T}\rangle m(\overline{V}, t, \overline{t}) \longmapsto V.\langle\overline{T}\rangle m(\overline{V}, t', \overline{t})}$$

C-Let
$$\frac{t \longmapsto t'}{\texttt{let}\ x : T = t\ \texttt{in}\ t_2 \longmapsto \texttt{let}\ x : T = t'\ \texttt{in}\ t_2}$$

C-Elvis
$$\frac{t \longmapsto t'}{t\ ?{:}\ t_2 \longmapsto t'\ ?{:}\ t_2}$$

Fig. 3. Evaluation Rules

$$\textsc{Err-Read}\quad \texttt{null}.f\ \mathsf{err} \qquad \textsc{Err-Invoke}\quad \texttt{null}.\langle\overline{T}\rangle m(\overline{V})\ \mathsf{err} \qquad \textsc{Err-Field}\ \frac{t\ \mathsf{err}}{t.f\ \mathsf{err}} \qquad \textsc{Err-New}\ \frac{t\ \mathsf{err}}{\mathsf{new}\ N(\overline{V}, t, \overline{t})\ \mathsf{err}} \qquad \textsc{Err-Receiver}\ \frac{t\ \mathsf{err}}{t.\langle\overline{T}\rangle m(\overline{t})\ \mathsf{err}}$$

$$\textsc{Err-Arg}\ \frac{t\ \mathsf{err}}{V.\langle\overline{T}\rangle m(\overline{V}, t, \overline{t})\ \mathsf{err}} \qquad \textsc{Err-Let}\ \frac{t\ \mathsf{err}}{\texttt{let}\ x : T = t\ \texttt{in}\ t_2\ \mathsf{err}} \qquad \textsc{Err-Elvis}\ \frac{t\ \mathsf{err}}{t\ ?{:}\ t_2\ \mathsf{err}}$$

Fig. 4. Null Pointer Errors

inferred types in the second subterm. Stub does the same thing given a `null` value. Unpack's and Stub's premises are needed for the proof of preservation. The subtyping and witnessed judgments can be found in figure 5 and will be discussed in the next section.

It's useful to compare our Unpack rule to conventional formulations of existential types [Pierce 2002]. Those calculi include a pack operation that wraps a value, V, and a *witness type*, U, into a new value that represents the packed existential, written $\{*U, V\}$. An unpack $X, x = t_1$ in t_2 expression then unwraps the existential value by binding the witness type and the wrapped value to the given variables X and x, respectively.

Our calculus, like Java, doesn't include an explicit pack. Instead, non-trivial existential types are introduced implicitly by the subtyping judgment. Java also implicitly eliminates wildcards using capture conversion, but our calculus instead makes existential elimination explicit. And during that existential elimination we now have to synthesize the witness type(s) that would normally have been provided by the programmer using an explicit pack.

Note that our evaluation judgment doesn't handle `null` pointer dereferences, i.e., $\texttt{null}.f$ and $\texttt{null}.\langle\overline{T}\rangle m(\ldots)$ expressions. Following Harper [2016] we instead define a judgment t err that—largely analogously to $t \longmapsto t'$—in essence formalizes the occurrence of null pointer exceptions (figure 4). We'll use this judgment in the proof of progress.

Of course, this formulation is impractical: we wouldn't want to synthesize witness types during evaluation in a practical programming language. This calculus merely serves to study implicitly packed existential types in the presence of `null` values while drawing analogies to conventional existential types with explicit pack operations. We'll return to this point when discussing progress as well.

4.3 Type System

Rules for typechecking terms as well as class and method declarations are summarized in figure 6. They rely on subtyping and well-formedness rules in figure 5.

The subtyping and well-formedness rules are for the most part adapted from Cameron et al. [2008], with an important difference: While WF-Class requires that upper and lower bounds of the existentially quantified type variables be well-formed as in TameFJ [Cameron et al. 2008], we do *not* require that $\overline{L} <: \overline{U}$. Unpack and Stub do effectively require that property at runtime. However, omitting the property from WF-Class makes subtype-asserting wildcards expressible in our system. That's critical for our system's expressiveness because subtype-asserting wildcards are not only possible to specify with Java wildcards [Tate et al. 2011], but as discussed also lie at the heart of Java's unsoundness. We'll return to this point shortly.

Note that even though $\bot$ is not a type in our system, we "lift" the subtyping and well-formedness judgments to include $\bot$ so we can use them on lower bounds L as well as types.

S-Refl
$$\Delta \vdash L <: L$$

S-Bot
$$\Delta \vdash \bot <: L$$

S-Trans
$$\frac{\Delta \vdash K <: L' \qquad \Delta \vdash L' <: L}{\Delta \vdash K <: L}$$

S-VarLeft
$$\frac{X : L..U \in \Delta}{\Delta \vdash X <: U}$$

S-VarRight
$$\frac{X : L..U \in \Delta}{\Delta \vdash L <: X}$$

S-Extends
$$\frac{\begin{array}{c}\texttt{class } C\langle\overline{X \lhd U}\rangle \lhd N\{\ldots\} \\ \overline{X} \cap \mathsf{dom}(\Delta, \Delta') = \emptyset\end{array}}{\Delta \vdash \exists\Delta'.C\langle\overline{T}\rangle <: \exists\Delta'.[\overline{T}/\overline{X}]N}$$

S-Exists
$$\frac{\begin{array}{c}\Delta, \Delta' \vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T} \qquad \Delta, \Delta' \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U} \\ \mathsf{fv}(\overline{T}) \subseteq \mathsf{dom}(\Delta, \Delta') \qquad \mathsf{dom}(\Delta') \cap \mathsf{fv}(\exists\overline{X : L..U}.N) = \emptyset\end{array}}{\Delta \vdash \exists\Delta'.[\overline{T}/\overline{X}]N <: \exists\overline{X : L..U}.N}$$

WF-Bot
$$\Delta \vdash \bot \ \mathsf{ok}$$

WF-Top
$$\frac{\Delta, \overline{Y : L..U} \vdash \overline{L}, \overline{U} \ \mathsf{ok}}{\Delta \vdash \exists\overline{Y : L..U}.\texttt{Object}\langle\rangle \ \mathsf{ok}}$$

WF-Var
$$\frac{X : L..U \in \Delta \qquad \Delta \vdash L, U \ \mathsf{ok}}{\Delta \vdash X \ \mathsf{ok}}$$

WF-Class
$$\frac{\Delta' = \overline{Y : L..U} \qquad \Delta, \Delta' \vdash \overline{T}, \overline{L}, \overline{U} \ \mathsf{ok} \qquad \Delta, \Delta' \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U'} \qquad \texttt{class } C\langle\overline{X \lhd U'}\rangle \lhd N\{\ldots\}}{\Delta \vdash \exists\Delta'.C\langle\overline{T}\rangle \ \mathsf{ok}}$$

W-Var
$$\frac{\Delta \vdash X \ \mathsf{ok}}{\Delta \vdash X \ \mathsf{witnessed}}$$

W-Direct
$$\frac{\Delta \vdash \exists.C\langle\overline{T}\rangle \ \mathsf{ok} \qquad \Delta \vdash \overline{T} \ \mathsf{witnessed}}{\Delta \vdash \exists.C\langle\overline{T}\rangle \ \mathsf{witnessed}}$$

W-Wildcard
$$\frac{\begin{array}{c}\Delta' = \overline{Y : L..U} \neq \emptyset \qquad \Delta, \Delta' \vdash \overline{T}, \overline{L}, \overline{U} \ \mathsf{witnessed} \\ \Delta \vdash \exists.N <: \exists\Delta'.C\langle\overline{T}\rangle \qquad \Delta \vdash \exists\Delta'.C\langle\overline{T}\rangle \ \mathsf{ok} \qquad \Delta \vdash \exists.N \ \mathsf{witnessed}\end{array}}{\Delta \vdash \exists\Delta'.C\langle\overline{T}\rangle \ \mathsf{witnessed}}$$

$$\frac{\mathsf{mtype}(m, N) = \langle\overline{Y \lhd U}\rangle\overline{T} \to T}{\mathsf{override}(m, N, \langle\overline{Y \lhd U}\rangle\overline{T} \to T)} \qquad \frac{\mathsf{mtype}(m, N) \ \text{undefined}}{\mathsf{override}(m, N, \langle\overline{Y \lhd U}\rangle\overline{T} \to T)}$$

Fig. 5. Subtyping and Well-Formedness

Compared to the previous literature, our calculus includes an additional judgment, T witnessed, which in essence formalizes well-formed types known to have a witness type (see figure 5):

- W-Var postulates all type variables as witnessed. This will allow typing `null` using type variables, which is commonly needed in generic data structures such as maps.
- W-Direct considers any trivially existential type $\exists.C\langle\overline{T}\rangle$ as witnessed as long as the type parameters $\overline{T}$ are witnessed.
- W-Wildcard admits proper existential types as witnessed if they (1) have witnessed type arguments and type variable bounds and (2) have a witnessed, trivially existential subtype.

This judgment is stronger than and separate from the well-formedness judgment, T ok, so we can apply it selectively, and specifically, so we can admit existential types encoding subtype-asserting wildcards in WF-Class as mentioned above.

We'll now discuss each typing rule in turn.

T-Null. We require `null` literals be typed with a witnessed type. This premise is a departure from Java's type system and—as innocuous as it may seem—ultimately allows proving soundness of our calculus. The JLS assigns `null` literals the non-denotable "null type" which it specifies to be a subtype of all reference types (including types containing wildcard type arguments, [Gosling et al. 2021, ch. 4.10.2]). But we deliberately do not and cannot type `null` using $\bot$: for one, that would require $\bot$ to be a type, i.e., its use not be limited to type variable bounds. While that introduces unpleasant technical complications into $F_{<:}$-like systems, they are surmountable [Pierce 2002, ch. 28.8]. The second reason though is that $\bot$ is defined to be a subtype of any type, and specifically also of existential types.

The key insight of this paper is that it is *unsound* to type `null` literals as $\bot$ in a type system that features subtype-asserting existential types, as ours (and Java) does. Limiting `null` to witnessed types, as shown in T-Null, avoids this problem. Intuitively that's because our rule forces that existential types be properly introduced using S-Exists, instead of "through the backdoor" using S-Bot. Note that `null` can still be typed with a type variable, as long as we take appropriate precautions in T-New and T-Call as explained next.

T-Var, T-New, T-Field, and T-Call are similar to FGJ [Igarashi et al. 2001], except that we restrict type arguments in T-New and T-Call to be witnessed. This restriction enables safely typing `null` using type variables, as it makes sure that substituting type variables with provided type arguments still results in witnessed types. By only restricting type arguments in T-New and T-Call, as opposed to all types, we also allow most uses of subtype-asserting wildcards, e.g., in variable and field types, in the vast corpus of Java code we analyzed (see Section 5.1).

A restriction like this is intuitively needed to prevent programs from introducing existential types without witnesses in nested type arguments, e.g., in `new ArrayList<Constrain<? super X>>()`. That could be just as problematic as the type `Constrain<? super X>` from figure 1! Nonetheless, this restriction causes the one false positive warning our approach raised in the Java corpus we analyzed. Section 5.1 provides details about this false positive and discusses possible ways of further relaxing the restriction.

Note that thanks to W-Wildcard, known-safe existential types such as $\exists X : \bot..\texttt{Number}.\texttt{List}\langle X\rangle$ (the equivalent of `List<? extends Number>` in our calculus) that don't encode subtype-asserting wildcards *are permitted* as type arguments in T-New and T-Call.

T-Let. While elaborate, this rule mostly formalizes conventional unpacking of existential types [Pierce 2002]. The somewhat unusual condition that all bound type variables are free in the unpacked type is needed in the proof of preservation. It rules out examples such as $\exists X.\texttt{Object}\langle\rangle$, where X goes unused. This is not a limitation in practice: we simply ask that the user-written types in `let` bindinds only define type variables that are used. Java wildcard types such as `List<?>`, where existentially bound type variables are implicit, can always be encoded to satisfy this condition, here, with $\exists X : \bot..\texttt{Object}.\texttt{List}\langle X\rangle$.

Our approach to preventing free type variables from appearing in the `let` expression's overall type is to allow the second subterm's type to be a subtype of the expression's overall type, but require the overall type to be well-formed without the type variables bound by `let`. This in particular allows two important idioms: repacking the unpacked value as-is (by relying on S-Exists to re-introduce the unpacked existential type) and subterms whose type is one of the bound type variables, which can be abstracted to the type variable's bound in the overall type.

T-Elvis. This rule types "Elvis" expressions in the obvious way, by requiring both subterms' types to have a common supertype.

Finally, T-Class and T-Method check well-formedness of class and method declarations, respectively, and are mostly standard from FGJ [Igarashi et al. 2001]. For reasons similar to T-New and T-Call, the class's declared supertype is required to be witnessed.

T-Var
$$\frac{x : T \in \Gamma}{\Delta|\Gamma \vdash x : T}$$

T-Null
$$\frac{\Delta \vdash T \text{ witnessed}}{\Delta|\Gamma \vdash \mathtt{null} : T}$$

T-New
$$\frac{\Delta \vdash \exists.C\langle\overline{T}\rangle \text{ witnessed} \qquad \text{fields}(C\langle\overline{T}\rangle) = \overline{U f} \qquad \Delta|\Gamma \vdash \overline{t : S} \qquad \Delta \vdash \overline{S <: U}}{\Delta|\Gamma \vdash \text{new } C\langle\overline{T}\rangle(\overline{t}) : \exists.C\langle\overline{T}\rangle}$$

T-Field
$$\frac{\Delta|\Gamma \vdash t : T \qquad \Delta \vdash T <: \exists.N \qquad \text{fields}(N) = \overline{T f}}{\Delta|\Gamma \vdash t.f_i : T_i}$$

T-Call
$$\frac{\begin{array}{c}\Delta \vdash \overline{T} \text{ witnessed} \qquad \Delta|\Gamma \vdash t : T \qquad \Delta|\Gamma \vdash \overline{t : S} \qquad \Delta \vdash T <: \exists.N \\ \text{mtype}(m, N) = \langle\overline{X \lhd U'}\rangle\overline{U} \to U \qquad \Delta \vdash \overline{S} <: [\overline{T}/\overline{X}]\overline{U} \qquad \Delta \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U'}\end{array}}{\Delta|\Gamma \vdash t.\langle\overline{T}\rangle m(\overline{t}) : [\overline{T}/\overline{X}]U}$$

T-Let
$$\frac{\begin{array}{c}\Delta|\Gamma \vdash t_1 : T_1 \qquad \Delta \vdash T_1 <: \exists\Delta'.N \\ \Delta, \Delta'|\Gamma, x : \exists.N \vdash t_2 : T_2 \qquad \Delta, \Delta' \vdash T_2 <: T \qquad \text{dom}(\Delta') \subseteq \text{fv}(N) \qquad \Delta \vdash T, \exists\Delta'.N \text{ ok}\end{array}}{\Delta|\Gamma \vdash \mathtt{let}\ x : \exists\Delta'.N = t_1\ \mathtt{in}\ t_2 : T}$$

T-Elvis
$$\frac{\Delta|\Gamma \vdash t_1 : T_1 \qquad \Delta|\Gamma \vdash t_2 : T_2 \qquad \Delta \vdash T_1 <: T \qquad \Delta \vdash T_2 <: T}{\Delta|\Gamma \vdash t_1 \mathrel{?:} t_2 : T}$$

T-Method
$$\frac{\begin{array}{c}\text{dom}(\Delta) = \overline{X} \qquad \Delta' = \overline{Y : \bot..U} \qquad \Delta, \Delta' \vdash \overline{U}, T, \overline{T} \text{ ok} \qquad \mathtt{class}\ C\langle\overline{X \lhd \_}\rangle \lhd N\{\ldots\} \\ \Delta, \Delta'|\overline{x : T}, \mathtt{this} : \exists.C\langle\overline{X}\rangle \vdash t : S \qquad \Delta, \Delta' \vdash S <: T \qquad \text{override}(m, N, \langle\overline{Y \lhd U}\rangle\overline{T} \to T)\end{array}}{\Delta \vdash \langle\overline{Y \lhd U}\rangle T\ m(\overline{T\ x}) = t \text{ ok in } C}$$

T-Class
$$\frac{\Delta = \overline{X : \bot..U} \qquad \Delta \vdash \overline{U}, \overline{T} \text{ ok} \qquad \Delta \vdash \exists.N \text{ witnessed} \qquad \Delta \vdash \overline{M} \text{ ok in } C}{\mathtt{class}\ C\langle\overline{X \lhd U}\rangle \lhd N\{\overline{T\ f}; \overline{M}\} \text{ ok}}$$

Fig. 6. Typing Rules

4.4 Type Safety

We're now ready to prove type safety using the familiar preservation and progress theorems. While we can state preservation in the usual way, the progress theorem has to account for the possibility of null pointer exceptions. We do so using the judgment t err from figure 4, following Harper [2016, ch. 6.3], instead of expecting "stuck" terms as was done to handle invalid casts in FGJ [Igarashi et al. 2001].

Note that we could enrich our system to statically avoid the possibility of null pointer exceptions. But in this paper we're specifically intending to show that *static nullness checking isn't needed* to avoid the unsoundness discovered in Java.

THEOREM 1 (PRESERVATION). *If* $\vdash t : T$ *and* $t \longmapsto t'$ *then* $\vdash t' : T'$ *and* $\vdash T' <: T$ *for some* T'.

PROOF. By induction on $t \longmapsto t'$, see appendix A. □

THEOREM 2 (PROGRESS). *If* $\vdash t : T$ *then either (1)* t *is a value or (2)* t err *or (3)* $t \longmapsto t'$ *for some* t'.

PROOF. By induction on $\vdash t : T$, see appendix B. □

The proof of preservation is lengthy but mostly standard. Even UNPACK and STUB are ultimately straightforward to handle because their premises give us types that can be substituted safely.

Instead, the progress property is arguably the one in jeopardy, essentially because it needs to come up with a suitable type substitution when wanting to appeal to UNPACK or STUB.

We're able to do so thanks to a lemma we call "witness lemma", which states that, given $\vdash \exists.N$ witnessed and $\vdash \exists.N <: \exists\overline{Y : L'..U'}.N'$, there exists a substitution $\overline{T}$ such that $\exists.N <: [\overline{T}/\overline{Y}]\exists.N'$ and $\vdash [\overline{T}/\overline{Y}]\overline{L'} <: \overline{T} <: [\overline{T}/\overline{Y}]\overline{U'}$ and $\vdash [\overline{T}/\overline{Y}]\exists.N'$ witnessed. That is to say, if we have a witnessed type that's a subtype of an existential type then there exist types $\overline{T}$ we can use to substitute the free type variables in N' and get another witnessed type. The lemma's conclusions then enable us to appeal to UNPACK or STUB as needed to make progress, including making sure that null's substituted type remains witnessed.

At this point, the seemingly odd premise we use in T-NULL comes into focus: it guarantees us that even when unpacking null, we can appeal to the witness lemma.

The intuition behind proving the witness lemma is surprisingly simple: we can more or less read the needed substitution off the subtyping judgment. Specifically, S-EXISTS establishes the existence of such a substitution, and we then simply need to propagate it through other subtyping rules and combine substitutions when encountering more uses of S-EXISTS. Of course, the subtyping judgment still synthesizes the needed substitution in our calculus. A formalization and proof of the witness lemma can be found as lemma 13 in appendix B.

Instead of having to "guess" witness types, an implementation of this system could store them where null and object values are introduced, and then look them up when applying STUB and UNPACK, respectively. This would avoid the runtime overhead of synthesizing witness types in our system to make progress. In Java, generic types including wildcards are of course erased at runtime instead [Igarashi et al. 2001].

4.5 Java's Unsoundness Revisited

It's instructive to check that our system indeed rejects programs like the ones in Amin and Tate [2016] that demonstrate Java's unsoundness, not just to make sure our system rejects them, but also to make sure they are rejected for the right reasons.

Figure 7 shows the simplest example from Amin and Tate [2016]—a generalized form of what we saw in figure 1—written in our calculus, simplified by omitting trivial bounds and ∃.s. The critical line is the `let` binding of `null` to the existential type $\exists W : X..Y.\texttt{Constrain}\langle Y, W\rangle$ on line 6. Our type system statically rejects that binding. To see that, consider that we would need to derive $X, Y | x : X \vdash \texttt{null} : S$ for some type S such that

$$X, Y \vdash S <: \exists W : X..Y.\texttt{Constrain}\langle Y, W\rangle \tag{1}$$

(per T-LET) and additionally $X, Y \vdash S$ witnessed (per T-NULL). But there is no such type! Specifically, we can't use the needed type, $\exists W : X..Y.\texttt{Constrain}\langle Y, W\rangle$, because we can't derive that it's

```
class Constrain<A, B ◁ A> {}
class UnsoundSpec {
  <A, B ◁ A> A upcast(Constrain<A, B> constrain, B b) = b

  <X, Y> Y coerce(X x) =
    let c : ∃W:X..Y. Constrain<Y, W> = null
    in upcast(c, x)
}
```

Fig. 7. Unsound program after Amin and Tate [2016, fig. 4] that our calculus rejects

witnessed, and $\exists$.Constrain$\langle Y, X\rangle$ doesn't satisfy Constrain's second type parameter's declared bound—since we don't know whether X is a subtype of Y—and is therefore ill-formed. Finally, $\exists$.Constrain$\langle Y, Y\rangle$ isn't a subtype of $\exists W : X..Y$.Constrain$\langle Y, W\rangle$: the only subtype rule we could apply is S-Exists, but appealing to S-Exists would require $X, Y \vdash X <: Y$ and since that relationship isn't assumed we can't derive it.

Let's compare to how other systems would reason about equivalent programs.

Java 17 [Gosling et al. 2021]. As detailed by Amin and Tate [2016] and Section 2, programs equivalent to figure 7 represent valid Java because Java effectively types null literals as $\bot$. While recent versions of javac reject this particular example because of the incompleteness of its type-checking algorithm where wildcards are involved, the same versions of javac accept variations of the problematic program [Amin and Tate 2016]. Thus, javac rejects some problematic programs, but for the wrong reasons, and accepts others.

TameFJ [Cameron et al. 2008]. TameFJ doesn't include null literals. But somewhat interestingly, we believe even if enhanced with null, the system presented in Cameron et al. [2008] could be proven sound, and would reject the equivalent of the program in figure 7. That's because in TameFJ, in order for existential types such as the one needed here to be well-formed, bound type variables' lower bounds are required to be known subtypes of their corresponding upper bounds [Cameron et al. 2008, fig. 3]. But that is not the case with our needed existential type, $\exists W : X..Y$.Constrain$\langle Y, W\rangle$, given the type environment X, Y that defines X and Y with only trivial bounds.

While TameFJ therefore seems protected from Java's unsoundness, these rules mean that existential types such as the above, i.e., existential types encoding subtype-asserting wildcards, simply aren't expressible. Our system *does allow* the introduction of existential types like this where they're backed by valid witnesses, while TameFJ would reject them outright as ill-formed. As we will see, that makes TameFJ akin to Kotlin, whereas the JLS specifically allows subtype-asserting wildcards by materializing them during capture conversion [Gosling et al. 2021, ch. 5.1.10].

Nullness types (e.g., [Fähndrich and Leino 2003; Papi et al. 2008]). Nullness-aware type systems, such as Scala's, could be used to rule out programs such as the one in figure 7, for instance by requiring proper existential types be non-null. But similar to TameFJ, requiring potentially "dangerous" existential types to be non-null arguably rejects the program in figure 7 for the wrong reason: the existential type isn't problematic per se as a nullable type; instead, the issue is that it can be unsoundly introduced by considering null a subtype of any (nullable) type.

Conventional existential types in $F_{<:}$, e.g., as presented in Pierce [2002], include a pack operation that specifies a witness type for each existential variable. This witness type is then checked against the type variable's declared bounds. It seems plausible that such a system could be extended to allow packing null values, and it would reject any such packing where the provided witness type doesn't validate the existential type's declared bounds. In other words, packing `null` values would be rejected when the existential type being introduced doesn't have a witness. Such a system would therefore reject the program in figure 7, and would arguably do so for the right reasons. However, in such a system, a packed existential is a different value than its contents, i.e., a packed `null` value is not the same thing as `null` itself. It's then fine to type `null` literals as $\bot$ as long as unpacking them fails with a null pointer exception, which it presumably would, since `unpack` would expect a packed `null` value, not `null` itself.

Thus, languages that have an explicit introduction form ("`pack`") for existential types can type `null` values as $\bot$, but systems like ours—where existential types are introduced implicitly through subtyping—cannot always soundly do so, simply because `null` values are *indistinguishable* from a packed existential containing `null`. We speculate that this subtle difference is at least partially the reason why the unsoundness discussed here made it into Java's specification in the first place and then remained undiscovered for over a decade.

5 WILDCARDS IN THE WILD

In this section we report a number of empirical results around wildcards, including statistics gathered by analyzing several hundred million lines of Java code in Google's monorepo. These results support the following conclusions:

(1) Java's unsoundness is of limited practical consequence: we didn't find any code affected by it.
(2) Java developers could realistically use the static analysis we developed to keep their code safe from Java's unsoundness: we only found a single, easily fixed false positive in the vast amount of Java code we analyzed.
(3) our statistics suggest that alternative fixes would more frequently unnecessarily break the code we analyzed.
(4) Kotlin's type system isn't susceptible to the source of unsoundness plaguing Java.

5.1 Problematic Wildcards in Java Code

In the previous sections we showed that by limiting the typing of `null` literals to types guaranteed to have a witness, we get a sound system, suggesting we can repair Java's unsoundness.

The direct way of doing so following the formal system presented in the previous section would be to modify the Java compiler's typechecker to find witness types where required. But that is inconvenient when wanting to analyze existing code, and finding witness types is non-trivial as well.

Instead, to evaluate our approach on real-world Java code, we created a static analysis using the Error-Prone framework[2] that plugs into `javac` and scans ASTs after typechecking is done to look for problematic types "after the fact". Because Error-Prone is integrated into Google's build system, Bazel[3], this enabled us to look for instances where Java code in Google's vast internal monorepo might run afoul of Java's unsoundness.

To determine whether a given type is witnessed without synthesizing witness types, our analysis leverages the following property that is straightforward to prove in our system:

Corollary 1. *If* $\Delta \vdash \overline{L} <: \overline{U}$ *and* $\Delta \vdash \overline{L}, \overline{U}$ witnessed *and* $\Delta \vdash \exists \overline{X : L..U}.C\langle\overline{T}\rangle$ ok *and* $\Delta, \overline{X : L..U} \vdash \overline{T}$ witnessed *then* $\Delta \vdash \exists \overline{X : L..U}.C\langle\overline{T}\rangle$ witnessed.

Proof. If $\overline{X} = \emptyset$ then the conclusion follows immediately from W-Direct. Otherwise, observe that $\Delta \vdash [\overline{U}/\overline{X}]\exists.C\langle\overline{T}\rangle <: \exists\overline{X : L..U}.C\langle\overline{T}\rangle$ by S-Exists. Moreover, $\Delta \vdash [\overline{U}/\overline{X}]\overline{T}$ witnessed by lemma

[2] https://errorprone.info
[3] https://bazel.build

7 and $\Delta \vdash [\overline{U}/\overline{X}]\exists.C\langle\overline{T}\rangle$ ok by lemma 6, and thus $\Delta \vdash [\overline{U}/\overline{X}]\exists.C\langle\overline{T}\rangle$ witnessed by W-Direct. Now the conclusion follows from lemma 2 and W-Wildcard. □

This means that any wildcard whose lower bound is known to be a subtype of its upper bound (without assuming that relationship, as Java does during capture conversion) is guaranteed to have a witness, namely the upper bound. Thus, to check whether a given type definitely has a witness, it suffices to recursively check whether the type is free of subtype-asserting wildcards, i.e., only contains wildcards whose lower bound is a known subtype of the respective wildcard's upper bound. While this approach is an incomplete (but sound per corollary 1) approximation, it is sufficient for our purposes because the Java compiler already rules out most subtype-asserting wildcards as we will discuss below.

Note that a wildcard can never be subtype-asserting if its lower bound is trivial ($\bot$). That means we can limit ourselves to wildcards with explicit lower bounds, i.e., `? super` wildcards, and then compare the declared lower bound to any implicit upper bound. (Recall the upper bound is "implicit" because it is inherited from the type parameter that the wildcard stands in for.)

Then, to find existing code that might suffer from Java's unsoundness, it suffices to check cases where our formal system requires witnessed types and making sure they don't contain subtype-asserting wildcards. Specifically, our static analysis checks the following for occurrences of subtype-asserting wildcards:[4]

- types that `null` literals are assigned to, and similar with other expressions of "null type", such as parenthesized "`(null)`". This broadly covers `null` values flowing through the program, including initialization and assignment (incl. mutation) of fields and variables, method arguments, method returns, etc.
- type arguments $\overline{T}$ in expressions `new` $C\langle\overline{T}\rangle(\ldots)$ and $t_0.\langle\overline{T}\rangle m(\ldots)$, whether they are explicitly provided in the program text or inferred by the compiler.
- type arguments $\overline{T}$ in "extends" (and "implements") clauses, written $\ldots \triangleleft C\langle\overline{T}\rangle\{\ldots\}$ in our formal system.
- types of non-final fields without initializer, since these fields implicitly start out as `null` and may not be otherwise checked by any of the above. This is to account for Java's field initialization semantics.

Our static analysis recursively searches parameterized types for nested subtype-asserting wildcards, as stipulated by the witnessed judgment. We treat array types `Foo[]` like a parameterized type `Box<Foo>`, i.e., we recursively look through array types as well.

We ran this analysis over a corpus of several hundred million lines of Java code, consisting of about 20% open-source and 80% proprietary software, as archived in Google's vast internal monorepo. The proprietary portion included Java code written for Android as well as frontend and backend server applications by thousands of Googlers over many years. The analyzed corpus also included tens of millions of lines of open-source Java code stemming from hundreds of open-source repositories, including repositories commonly used for corpus analysis in the literature, such as PMD and various Apache projects. Some of the analyzed open-source repositories, such as Guava, are being primarily maintained by Google, but many are not. (For comparison, our corpus's open-source portion alone was several times larger than the corpus considered in a previous wildcard-related study [Tate et al. 2011, sect. 9].)

Our analysis found no assignment of `null` to a type containing a subtype-asserting wildcard in the entire analyzed corpus. However, we found a (single) file containing a call to a generic method

[4]Our static analysis is open-source as part of https://errorprone.info here: https://github.com/google/error-prone/blob/master/core/src/main/java/com/google/errorprone/bugpatterns/nullness/UnsafeWildcard.java.

Table 2. `<? super ...>` wildcard occurrences per 100 million lines of analyzed Java code (approximate). "Null assignments" count assignments of null-typed expressions to a type containing a wildcard. "Any assignments" count any assignments to such a type. "Inferred type arguments" count wildcards in generic class and method type arguments that the compiler inferred, without appearing explicitly in source. Conversely, "declarations in source" count wildcards that appear in source code. Row A only considers wildcards whose given lower bound is *not* a known subtype of its implicit upper bound. Row B considers wildcards with given lower bound and an implicit upper bound other than `Object`. Row C considers all `<? super ... >` wildcards regardless of their upper bound. The entry marked * represents a single file in the entire analyzed corpus.

	Wildcards with non-trivial lower bounds (? super wildcards) …	Null assignment	Inferred type arguments	Any assignment	Declarations in source
A	… that are *subtype-asserting*	None found	<.5*	2	3
B	… and non-trivial implicit upper bound	14	230	6,900	1,100
C	… and any upper bound	230	1,300	740,000	11,000

```
1  import java.util.List;
2  import java.util.function.Function;
3  import java.util.stream.*;
4
5  public class GenericMethod {
6    public interface Marker {}
7    public interface Converter<T extends Marker> { List<?> convert(T input); }
8
9    public <T extends Marker> Function<? super T, List<?>>
10   transformerFor(Iterable<Converter<? super T>> cs) { return new Impl<T>(cs); }
11
12   // Error below can be avoided here with "class Impl<T extends Marker> ..."
13   private static class Impl<T> implements Function<T, List<?>> {
14     private final Iterable<Converter<? super T>> cs;
15     private Impl(Iterable<Converter<? super T>> cs) { this.cs = cs; }
16
17     @Override public List<?> apply(T input) {
18       // WARNING: inferred type argument Spliterator<Converter<? super T>>
19       // for flatMap() contains subtype-asserting wildcard
20       return StreamSupport.stream(cs.spliterator(), false)
21           .flatMap(c -> c.convert(input).stream())
22           .collect(Collectors.toList());
23     }
24   }
25 }
```

Fig. 8. Generic method call with potentially unsafe inferred type argument our analysis found (simplified). The code as shown is sanitized and logically simplified from the (proprietary) original and inlines helper declarations originally declared in separate files.

whose inferred type argument contained a nested subtype-asserting wildcard (see table 2). Figure 8 shows a simplified and sanitized, standalone version of the file, with the flagged generic method call on line 20.

As can be seen, this particular issue is easily fixed by adding the missing upper bound to the inner class's type parameter declaration in line 13. The so-modified code successfully compiles because the same upper bound is already declared in line 9. With that modification, the code also isn't flagged by our analysis anymore.

Thus, even though this code comes quite close, it doesn't run afoul of Java's latent unsoundness. Therefore, we conclude that *none* of the code we analyzed appears to be affected by Java's unsoundness. This confirms at very large scale the prevailing belief that Java's unsoundness is of minimal practical significance.

However, that also arguably makes the warning our analysis raises in figure 8 a false positive. In future work we plan to investigate ways of recognizing this code as safe as-is, e.g., by leveraging that the private inner class in question is always instantiated with a suitably bounded type parameter as mentioned above.

Even with the false positive discussed above, table 2 shows that our approach is substantially more precise than potential other ways of detecting sources of unsoundness in existing Java code.

- As alluded to before, we could instead flag all wildcard *declarations* with non-trivial lower and upper bounds, which Amin and Tate [2016, sect. 7] had suggested could be used to address Java's unsoundness. This would correspond to the last column of row B in table 2 (detailed below), which shows that in the analyzed corpus, this approach would (unnecessarily) flag roughly one out of every 90 thousand lines of code.
- We could alternatively seemingly only flag declarations of subtype-asserting wildcards, which as discussed we understand TameFJ to effectively do [Cameron et al. 2008]. Per the last column of row A in table 2, this would also trigger more frequently in the code we analyzed, though only a handful of times per 100 million lines of code.

That suggests that these alternative approaches would more commonly unnecessarily flag actually safe code in practice.[5] For instance, TameFJ would consider all wildcard declarations counted in the last column of row A as ill-formed.

To understand how the corpus of Java code we analyzed relates to other Java code, we note that we only found a single subtype-asserting wildcard in the open-source portion of our corpus (namely, the wildcard detailed in footnote 5), while we found such wildcards slightly more commonly in proprietary code (see last column of row A in table 2). Given that such wildcard declarations are necessary (though not sufficient) for triggering Java's unsoundness, we conclude that the code we analyzed overall appears to be representative of or more challenging than the large cross-section of open-source Java we analyzed, and therefore likely other Java code, when it comes to Java's unsoundness specifically.

It's instructive to ask why subtype-asserting wildcards are so rare in practice to begin with (see row A in table 2). At least in part, this is due to the Java compiler already often enforcing the rule we used, i.e., that a wildcard's lower bound must be a known subtype of its upper bound. For instance, when a wildcard's lower bound is a concrete type, such as `? super String`, then the compiler—according to our manual experiments with `javac` 17—appears to make sure that the concrete type is indeed a subtype of the bound declared for the type parameter the wildcard corresponds to. If the lower bound is a type variable, `? super T`, the compiler also performs this check, but *only if the referenced type variable is declared with a non-trivial upper bound* (i.e., something other than `Object`). The only time the Java compiler allows subtype-asserting wildcards

[5]One example of a subtype-asserting wildcard that our approach permits but would no longer compile if subtype-asserting wildcards were forbidden outright in Java we found in the open-source project PMD here: https://github.com/pmd/pmd/blob/05d5bead37e04690d4f5169898801a6b1ec6dd1e/pmd-core/src/main/java/net/sourceforge/pmd/lang/ast/SignedNode.java#L24.

is therefore when a type variable whose own upper bound is `Object` is used as the lower bound of a wildcard that stands in for a type parameter with non-trivial upper bound, as is the case in figure 7. Additionally, as Amin and Tate [2016] discuss, the compiler's incomplete handling of wildcards *sometimes* leads it to reject even remaining cases, just not always. These circumstances are fortunate, in a way, since they mean the Java compiler already makes it very unlikely to accidentally run afoul of the existing unsoundness. Our simple additional rules therefore allow eliminating the problem completely with seemingly marginal impact on existing code.

For comparison, we collected similar statistics to above while reducing precision in the wildcard types we look for. That is, the remaining rows in table 2 show `null` assignments etc. for progressively larger supersets of wildcards compared to row A. Specifically, row B includes lower-bounded wildcards that have any non-trivial upper bounds, whether they are subtype-asserting or not. As the table shows, such wildcards occur orders of magnitude more frequently in the corpus we analyzed. Finally, row C shows all wildcards with non-trivial lower bound—whether they have trivial or non-trivial upper bounds—for comparison. Again, these cases occur substantially more frequently compared to the previous condition.

5.2 Implications for Kotlin

Kotlin[6] is a Java-like programming language that can execute on Java Virtual Machines. By supporting both declaration-site and use-site variance, Kotlin allows expressing types that are semantically similar to Java wildcards [Tate 2013]. It is therefore reasonable to wonder whether Kotlin might be susceptible to soundness issues similar to Java's. The paper that reported Java's and Scala's unsoundness is silent on this point [Amin and Tate 2016].

Based on the results in this paper, Kotlin appears to be protected from the unsoundness plaguing Java. The reason we come to this conclusion is that Kotlin's typechecker, which reportedly doesn't support "implicit constraints" [Tate 2013], appears to consider the equivalent of subtype-asserting wildcards as ill-formed. Based on corollary 1 this trivially ensures that every wildcard-like type in Kotlin has a witness. Moreover, `null` literals can always be assigned to types containing wildcards, namely by typing `null` with the witness type we know must exist. In terms of our formal system, Kotlin's rule therefore guarantees `null` can always be typed, because T-Null's premise always holds (thanks to corollary 1), and that Stub can always make progress when evaluating a well-typed expression, which in turn guarantees type safety.

In other words, Kotlin appears to be "correct by construction", by which we mean that unlike in Java, `null` can be typed bottom ($\perp$) in Kotlin. We'll note that TameFJ's aforementioned restrictions on well-formed existential types [Cameron et al. 2008] appear similar to Kotlin's. However, Kotlin's mixed-site variance, which neither our formal system nor TameFJ directly capture, still warrants further study, though Tate [2013] suggests that mixed-site variance can be encoded with Java-style use-site variance.

We'll point out that Kotlin's type system is nullness-aware, but by forbidding subtype-asserting wildcards, Kotlin appears protected from Java's unsoundness thanks to corollary 1 and thus in a manner orthogonal to Kotlin's nullness checking.

6 RELATED WORK

Amin and Tate [2016] demonstrate that Java and Scala are unsound with a series of examples. In Java's case they show that unsoundness can arise when `null` references are assigned to subtype-asserting wildcards. Java wildcards have been extensively studied [Cameron et al. 2008; Smith and

[6]https://kotlinlang.org/

Cartwright 2008; Tate et al. 2011], and Kotlin's mixed-site variance was specifically designed to avoid many of the previously identified pitfalls [Tate 2013].

Amin and Tate [2016] informally discuss possible fixes for Java's unsoundness that would restrict allowable wildcard declarations, e.g., by forbidding wildcards with non-trivial lower bounds that also inherit a non-trivial upper bound from the type parameter they stand in for. Nieto et al. [2020] informally suggest that nullness types can be used to fix Scala's unsoundness. To our knowledge, this paper proposes a novel "fix" for Java that wasn't previously considered: our approach is to ensure the sound introduction of Java wildcards in all cases while allowing subtype-asserting wildcards where they are safe. This is to our knowledge also the first paper to prove soundness (in a core calculus) of a suggested fix for Java. As a corollary we're able to show that the approach Kotlin appears to employ—namely forbidding subtype-asserting wildcards—is also sound in our calculus.

Tony Hoare famously described the `null` pointer as a "billion dollar mistake". Nullness types have found their way into widely used programming languages in recent years, including C#, Dart, Kotlin, Scala, Swift, and TypeScript. They've also long been the subject of academic research (e.g., [Chalin and James 2007; Fähndrich and Leino 2003; Papi et al. 2008]), not least as motivation for studying sum and union types [Harper 2016]. Sound initialization of non-null fields in particular has been the subject of extensive study [Fähndrich and Leino 2003; Fähndrich and Xia 2007; Liu et al. 2020; Summers and Mueller 2011].

Scala is reportedly using nullness types to fix its unsoundness [Nieto et al. 2020]. By contrast, this paper shows that Java's unsoundness is avoidable *without* nullness types, and supports the hypothesis that Kotlin likewise avoids unsoundness without reliance on its nullness type system [Tate 2013].

Amin and Tate [2016] survey the extensive literature formalizing different subsets of Java, such as Drossopoulou et al. [1999]; Flatt et al. [1998]; Igarashi et al. [2001]. Existing formalizations of Java wildcards [Cameron et al. 2008; Torgersen et al. 2005] didn't consider `null` nor subtype-asserting wildcards (see Section 4.5). The formal system in this paper builds in particular on FGJ [Igarashi et al. 2001] and TameFJ [Cameron et al. 2008] but includes `null` as well as existential types capable of expressing subtype-asserting wildcards.

Tate et al. [2011] included an empirical analysis of Java wildcards in nearly 10 million lines of open-source Java code. This paper includes an empirical analysis of wildcards in a much larger corpus of Java code. Most of the empirical findings presented in this paper are novel and specific to Java's unsoundness, but some of the reported numbers are comparable to data Tate et al. [2011] previously reported for their different corpus. In particular, the corpus analyzed for this paper included (rare) subtype-asserting wildcards (cf. table 2, row A) where the (much smaller) corpus from Tate et al. [2011] included none.

7 CONCLUSIONS

This paper shows that Java's unsoundness [Amin and Tate 2016] is fixable with small tweaks to Java's typing rules to ensure that Java wildcards are never introduced without witness types, which Java currently unsoundly permits when assigning `null`. We formalized this idea in a core calculus and proved it sound. We also implemented our approach as a static analysis and used it to analyze a corpus of several hundred million lines of Java code. Our analysis found no issues and a single false positive in the code we analyzed, suggesting that our approach may be practical to adopt by Java developers—and maybe even into Java—with minimal impact on existing code. Our formal system also suggests that Kotlin doesn't suffer from unsoundness similar to Java's.

ACKNOWLEDGMENTS

This work would not have been possible without the fantastic code analysis tools that Error-Prone team and others at Google have built over the years. The author thanks Erik Ernst, Jeffrey van Gogh, Steven Schäfer, Error-Prone team and the anonymous reviewers for their thoughtful help and feedback. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of Google or its affiliates.

REFERENCES

Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 838–848. https://doi.org/10.1145/2983990.2984004

Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. 2008. A Model for Java with Wildcards. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–26. https://doi.org/10.1007/978-3-540-70592-5_2

Patrice Chalin and Perry R. James. 2007. Non-Null References by Default in Java: Alleviating the Nullity Annotation Burden. In *Proceedings of the 21st European Conference on Object-Oriented Programming* (Berlin, Germany) *(ECOOP'07)*. Springer-Verlag, Berlin, Heidelberg, 227–247. https://doi.org/10.1007/978-3-540-73589-2_12

Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. 1999. Is the Java type system sound? *Theory and Practice of Object Systems* 5, 1 (1999), 3–24. https://doi.org/10.1002/(SICI)1096-9942(199901/03)5:1<3::AID-TAPO2>3.0.CO;2-T

Manuel Fähndrich and K. Rustan M. Leino. 2003. Declaring and Checking Non-Null Types in an Object-Oriented Language. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications* (Anaheim, California, USA) *(OOPSLA '03)*. Association for Computing Machinery, New York, NY, USA, 302–312. https://doi.org/10.1145/949305.949332

Manuel Fähndrich and Songtao Xia. 2007. Establishing Object Invariants with Delayed Types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 337–350. https://doi.org/10.1145/1297027.1297052

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 171–183. https://doi.org/10.1145/268946.268961

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. 2021. *The Java Language Specification: Java SE 17 Edition*. Retrieved Oct 20, 2021 from https://docs.oracle.com/javase/specs/jls/se17/html/index.html

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, USA.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

Fengyun Liu, Ondřej Lhoták, Aggelos Biboudis, Paolo G. Giarrusso, and Martin Odersky. 2020. A Type-and-Effect System for Object Initialization. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 175 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428243

Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. 2020. Scala with Explicit Nulls. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:26. https://doi.org/10.4230/LIPIcs.ECOOP.2020.25

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/1390630.1390656

Benjamin C. Pierce. 2002. *Types and Programming Languages*. The MIT Press, Cambridge, MA, USA.

Daniel Smith and Robert Cartwright. 2008. Java Type Inference is Broken: Can We Fix It?. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 505–524. https://doi.org/10.1145/1449764.1449804

Alexander J. Summers and Peter Mueller. 2011. Freedom before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 1013–1032. https://doi.org/10.1145/2048066.2048142

Ross Tate. 2013. Mixed-Site Variance. In *FOOL '13: Informal Proceedings of the 20th International Workshop on Foundations of Object-Oriented Languages* (Indianapolis, IN, USA). https://fool2013.cs.brown.edu/tate.pdf

Ross Tate, Alan Leung, and Sorin Lerner. 2011. Taming Wildcards in Java's Type System. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 614–627. https://doi.org/10.1145/1993498.1993570

Mads Torgersen, Erik Ernst, and Christian Plesner Hansen. 2005. Wild FJ. In *FOOL: Foundations of Object-Oriented Languages* (Long Beach, CA, USA). https://homepages.inf.ed.ac.uk/wadler/fool/program/14.html

A PROOF OF THEOREM 1 (PRESERVATION)

Lemma 1 (Permutation). *Let* Γ *and* Δ *be variable and type contexts, respectively. Assume that* Γ' *and* Δ' *are permutations of* Γ *and* Δ*, respectively.*

(1) *If* $\Delta \vdash K <: L$ *then* $\Delta' \vdash K <: L$.
(2) *If* $\Delta \vdash L$ ok *then* $\Delta' \vdash L$ ok.
(3) *If* $\Delta \vdash T$ witnessed *then* $\Delta' \vdash T$ witnessed.
(4) *If* $\Delta|\Gamma \vdash t : T$ *then* $\Delta'|\Gamma' \vdash t : T$.

Proof. Immediate from the rules, recalling that we assume the same variable name isn't defined twice in any context Γ or Δ. □

Lemma 2 (Weakening). *Let* $\Delta' = \overline{X : L..U}$ *and assume* $\overline{X} \cap dom(\Delta) = \emptyset$ *and* $x \notin \text{dom}(\Gamma)$.

(1) *If* $\Delta \vdash K <: L$ *then* $\Delta, \Delta' \vdash K <: L$.
(2) *If* $\Delta \vdash L$ ok *then* $\Delta, \Delta' \vdash L$ ok.
(3) *If* $\Delta \vdash T$ witnessed *then* $\Delta, \Delta' \vdash T$ witnessed.
(4) *If* $\Delta|\Gamma \vdash t : T$ *then* $\Delta, \Delta'|\Gamma \vdash t : T$ *and* $\Delta|\Gamma, x : S \vdash t : T$.

Proof. Part (1) by induction on the derivation of $\Delta \vdash K <: L$.
Cases S-Refl and S-Bot are immediate.
Case S-Trans. We have $\Delta \vdash K <: L'$ and $\Delta \vdash L' <: L$. By i.h. (induction hypothesis) we get $\Delta, \Delta' \vdash K <: L'$ and $\Delta, \Delta' \vdash L' <: L$. Then $\Delta, \Delta' \vdash K <: L$ by S-Trans as desired.
Cases S-VarLeft and S-VarRight are immediate because the type variable has to be defined in Δ and we assume that $\overline{X}$ don't overlap with Δ.
Case S-Extends. $K = \exists\Delta''.C\langle\overline{T'}\rangle$ and $L = \exists\Delta''.[\overline{T'}/\overline{Y}]N$ and `class` $C\langle\overline{Y \lhd U'}\rangle \lhd N\{\ldots\}$ and $\overline{Y} \cap \text{dom}(\Delta, \Delta'') = \emptyset$. Without loss of generality we may further assume that $\overline{Y} \cap \text{dom}(\Delta') = \emptyset$, since type variables can be renamed as needed. Then the conclusion is immediate from S-Extends.
Case S-Exists immediate from i.h. and lemma 1.
Proof of part (2) by induction on the derivation of $\Delta \vdash L$ ok using part (1) and lemma 1.
Proof of part (3) by induction on the derivation of $\Delta \vdash T$ witnessed using parts (1-2) and lemma 1.
Proof of part (4) by induction on the derivation of $\Delta|\Gamma \vdash t : T$ using parts (1-3) and lemma 1. □

Lemma 3. *If* $\Delta \vdash T$ witnessed *then* $\Delta \vdash T$ ok.

Proof. Immediate: all possible derivations of $\Delta \vdash T$ witnessed require $\Delta \vdash T$ ok. □

Lemma 4. *If* $\Delta \vdash L$ ok *then* $\text{fv}(L) \subseteq \text{dom}(\Delta)$.

Proof. By induction on $\Delta \vdash L$ ok.
WF-Bot trivial: $\text{fv}(\bot) = \emptyset$.
WF-Top. $L = \exists\overline{X : L..U}.\texttt{Object}\langle\rangle$ and $\Delta, \overline{X : L..U} \vdash \overline{L}, \overline{U}$ ok. Without loss of generality we assume that $\overline{X} \cap \text{dom}(\Delta) = \emptyset$. By i.h., $\text{fv}(\overline{L}, \overline{U}) \subseteq \text{dom}(\Delta) \cup \overline{X}$. Since $\text{fv}(\texttt{Object}\langle\rangle) = \emptyset$ and $\overline{X}$ are bound in L, $\text{fv}(L) \subseteq \text{dom}(\Delta)$ as needed.

WF-Var. $L = X$ and $X : K..U \in \Delta$. Thus $\mathsf{fv}(L) = \{X\}$ and $X \in \mathsf{dom}(\Delta)$ and the conclusion is immediate.

WF-Class. $L = \exists \overline{X : L..U}.C\langle\overline{T}\rangle$ and $\Delta, \overline{X : L..U} \vdash \overline{T}, \overline{L}, \overline{U}$ ok. Without loss of generality we assume that $\overline{X} \cap \mathsf{dom}(\Delta) = \emptyset$. By i.h., $\mathsf{fv}(\overline{T}, \overline{L}, \overline{U}) \subseteq \mathsf{dom}(\Delta) \cup \overline{X}$. Because $\mathsf{fv}(C\langle\overline{T}\rangle) = \mathsf{fv}(\overline{T})$ and $\overline{X}$ are bound in L, $\mathsf{fv}(L) \subseteq \mathsf{dom}(\Delta)$ as needed. □

Lemma 5 (Type substitution preserves subtyping). *If* $\Delta_1, \overline{X : L..U}, \Delta_2 \vdash K <: L$ *and* $\Delta_1 \vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$ *and* $\Delta_1 \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$ *and* $\mathsf{fv}(\overline{T}) \subseteq \mathsf{dom}(\Delta_1)$ *and none of* $\overline{X}$ *appear in* Δ_1 *then* $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]K <: [\overline{T}/\overline{X}]L$.

Proof. By induction on the derivation of $\Delta_1, \overline{X : L..U}, \Delta_2 \vdash K <: L$.

Cases S-Refl and S-Bot are immediate.

Case S-Trans. We have $\Delta_1, \overline{X : L..U}, \Delta_2 \vdash K <: L'$ and $\Delta_1, \overline{X : L..U}, \Delta_2 \vdash L' <: L$. By i.h., $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]K <: [\overline{T}/\overline{X}]L'$ and $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]L' <: [\overline{T}/\overline{X}]L$. Then $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]K <: [\overline{T}/\overline{X}]L$ follows by S-Trans.

Case S-VarLeft. We have $K = X$ and $X : L_X..U_X \in \Delta_1, \overline{X : L..U}, \Delta_2$ and $L = U_X$.

- If $X \in \mathsf{dom}(\Delta_1)$ then trivially $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash X <: U_X$ by S-VarLeft because $\overline{X}$ aren't free in U_X and therefore X and U_X are unaffected by the substitution.
- If $X \in \mathsf{dom}(\Delta_2)$ then $X : [\overline{T}/\overline{X}]L_X..[\overline{T}/\overline{X}]U_X \in [\overline{T}/\overline{X}]\Delta_2$ and $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]X <: [\overline{T}/\overline{X}]U_X$ by S-VarLeft.
- Else $X = X_i$. By assumption, $\Delta_1 \vdash T_i <: [\overline{T}/\overline{X}]U_i$. Since $[\overline{T}/\overline{X}]X_i = T_i$, by lemma 2, we get $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash T_i = [\overline{T}/\overline{X}]X_i <: [\overline{T}/\overline{X}]U_i$ as required.

Case S-VarRight is symmetric to S-VarLeft.

Case S-Extends. $K = \exists\Delta'.C\langle\overline{T'}\rangle$ and $L = \exists\Delta'.[\overline{T'}/\overline{Y}]N$ and $\texttt{class}\ C\langle\overline{Y \lhd U}\rangle \lhd N\{\ldots\}$ and $\overline{Y} \cap (\overline{X} \cup \mathsf{dom}(\Delta_1, \Delta_2, \Delta')) = \emptyset$. Without loss of generality we may further assume that $\overline{X} \cap \mathsf{dom}(\Delta_2, \Delta') = \emptyset$. Since $\mathsf{dom}([\overline{T}/\overline{X}](\Delta_2, \Delta')) = \mathsf{dom}(\Delta_2, \Delta')$ we know $\overline{Y} \cap \mathsf{dom}(\Delta_1, [\overline{T}/\overline{X}](\Delta_2, \Delta')) = \emptyset$.

Let $\Delta'' = [\overline{T}/\overline{X}]\Delta'$. By T-Class and lemmas 3 and 4, $\mathsf{fv}(N) \subseteq \overline{Y}$. Then $[\overline{T}/\overline{X}]L = \exists\Delta''.[\overline{[\overline{T}/\overline{X}]T'}/\overline{Y}]N$ and $[\overline{T}/\overline{X}]K = \exists\Delta''.C\langle\overline{[\overline{T}/\overline{X}]T'}\rangle$. Then the needed conclusion, $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]K <: [\overline{T}/\overline{X}]L$, follows by S-Extends.

Case S-Exists. Letting $\Delta = \Delta_1, \overline{X : L..U}, \Delta_2$, we have $K = \exists\Delta'.[\overline{T'}/\overline{Y}]N$ and $L = \exists\overline{Y : L'..U'}.N$ and $\Delta, \Delta' \vdash [\overline{T'}/\overline{Y}]\overline{L'} <: \overline{T'}$ and $\Delta, \Delta' \vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$ and $\mathsf{fv}(\overline{T'}) \subseteq \mathsf{dom}(\Delta, \Delta')$ and $\mathsf{dom}(\Delta') \cap \mathsf{fv}(L) = \emptyset$. We can assume without loss of generality that $\overline{X}$, $\overline{Y}$, and $\mathsf{dom}(\Delta')$ are pairwise disjoint.

By i.h., we get $\Delta_1, [\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta' \vdash [\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]\overline{L'} <: [\overline{T}/\overline{X}]\overline{T'}$ and $\Delta_1, [\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta' \vdash [\overline{T}/\overline{X}]\overline{T'} <: [\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]\overline{U'}$. Since we assume $\mathsf{fv}(\overline{T}) \subseteq \mathsf{dom}(\Delta_1)$, $\mathsf{fv}([\overline{T}/\overline{X}]\overline{T'}) \subseteq \mathsf{dom}(\Delta_1, [\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta')$ and $\mathsf{dom}([\overline{T}/\overline{X}]\Delta') \cap \mathsf{fv}([\overline{T}/\overline{X}]L) = \emptyset$. Then, using that $[\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]L' = [\overline{[\overline{T}/\overline{X}]T'}/\overline{Y}][\overline{T}/\overline{X}]L'$ for any L', we can derive $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\exists\Delta'.[\overline{T'}/\overline{Y}]N <: [\overline{T}/\overline{X}]\exists\overline{Y : L'..U'}.N$ by S-Exists as required. □

Lemma 6 (Type substitution preserves well-formedness). *If* $\Delta_1, \overline{X : L..U}, \Delta_2 \vdash L$ ok *and* $\Delta_1 \vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$ *and* $\Delta_1 \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$ *and* $\Delta_1 \vdash \overline{T}$ ok *and none of* $\overline{X}$ *appear in* Δ_1 *then* $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]L$ ok.

Proof. By induction on $\Delta \vdash L$ ok, letting $\Delta = \Delta_1, \overline{X : L..U}, \Delta_2$.

Case WF-Bot is immediate: type substitution has no effect.

Case WF-Var. We have $L = X$, $X : L_X..U_X \in \Delta$, and $\Delta \vdash L_X, U_X$ ok.

By i.h., $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]L_X, [\overline{T}/\overline{X}]U_X$ ok.

- If $X \in \mathsf{dom}(\Delta_1)$ then X, L_X, and U_X are unaffected by the substitution, and with the i.h., WF-Var gives the required $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash X$ ok.

- If $X \in \mathrm{dom}(\Delta_2)$ then $[\overline{T}/\overline{X}]X = X$ and $X : [\overline{T}/\overline{X}]L_X..[\overline{T}/\overline{X}]U_X \in [\overline{T}/\overline{X}]\Delta_2$. With the i.h., WF-Var gives the required $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]X = X$ ok.
- Else $X = X_i$ and $[\overline{T}/\overline{X}]X_i = T_i$. By assumption we know that $\Delta_1 \vdash T_i$ ok. Then lemma 2 gives us $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash T_i$ ok as required.

Case WF-Class. We have $L = \exists\Delta'.C\langle\overline{T'}\rangle$ and $\Delta' = \overline{X' : L'..U'}$ and $\Delta, \Delta' \vdash \overline{T'}, \overline{L'}, \overline{U'}$ ok and $\Delta, \Delta' \vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U''}$ and class $C\langle\overline{Y \lhd U''}\rangle \lhd N\{\ldots\}$.

By i.h., $\Delta_1, [\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta' \vdash [\overline{T}/\overline{X}]\overline{T'}, [\overline{T}/\overline{X}]\overline{L'}, [\overline{T}/\overline{X}]\overline{U'}$ ok. $\Delta_1 \vdash \overline{T}$ ok implies $\mathrm{fv}(\overline{T}) \subseteq \mathrm{dom}(\Delta_1)$ by lemma 4, and thus by lemma 5, $\Delta_1, [\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta' \vdash [\overline{T}/\overline{X}]\overline{T'} <: [\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]\overline{U''}$. Without loss of generality we can assume that $\overline{X}$, $\overline{X'}$, and $\overline{Y}$ are disjoint. Moreover $\mathrm{fv}(\overline{U''}) \subseteq \overline{Y}$ per T-Class and lemma 4, and therefore $[\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]\overline{U''} = [[\overline{T}/\overline{X}]\overline{T'}/\overline{Y}]\overline{U''}$. Then $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\exists\Delta'.C\langle\overline{T'}\rangle$ ok by WF-Class as required.

Case WF-Top like the relevant part of case WF-Class is trivial by i.h. □

Lemma 7 (Type substitution preserves witnesses). *If* $\Delta_1, \overline{X : L..U}, \Delta_2 \vdash T$ witnessed *and* $\Delta_1 \vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$ *and* $\Delta_1 \vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$ *and* $\Delta_1 \vdash \overline{T}$ witnessed *and none of* $\overline{X}$ *appear in* Δ_1 *then* $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ witnessed.

Proof. By induction on $\Delta \vdash T$ witnessed, letting $\Delta = \Delta_1, \overline{X : L..U}, \Delta_2$ and using that $\Delta_1 \vdash \overline{T}$ ok by lemma 3 and $\mathrm{fv}(\overline{T}) \subseteq \mathrm{dom}(\Delta_1)$ by lemma 4.

Case W-Var. $T = X$ and $\Delta \vdash X$ ok.

- If $X = X_i \in \overline{X}$ then $[\overline{T}/\overline{X}]X_i = T_i$. By assumption, $\Delta_1 \vdash T_i$ witnessed and $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash T_i$ witnessed follows by lemma 2 as required.
- Else $X \in \mathrm{dom}(\Delta_1, \Delta_2)$. By lemma 6, $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]X = X$ ok and thus $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]X = X$ witnessed by W-Var.

Case W-Direct. $T = \exists.C\langle\overline{S}\rangle$ and $\Delta \vdash T$ ok and $\Delta \vdash \overline{S}$ witnessed.

By i.h., $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\overline{S}$ witnessed. By lemma 6, $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ ok. Then $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ witnessed by W-Direct.

Case W-Wildcard. $T = \exists\Delta'.C\langle\overline{S}\rangle$ and $\Delta' = \overline{Y : L'..U'} \neq \emptyset$ and $\Delta \vdash \exists.N <: T$ and $\Delta \vdash T$ ok and $\Delta \vdash \exists.N$ witnessed and $\Delta, \Delta' \vdash \overline{S}, \overline{L'}, \overline{U'}$ witnessed.

By i.h., $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\exists.N$ witnessed and $\Delta_1, [\overline{T}/\overline{X}](\Delta_2, \Delta') \vdash [\overline{T}/\overline{X}](\overline{S}, \overline{L'}, \overline{U'})$ witnessed. By lemma 5, $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\exists.N <: [\overline{T}/\overline{X}]T$. By lemma 6, $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ ok. Then $\Delta_1, [\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ witnessed by W-Wildcard.

□

Lemma 8 (Type substitution preserves typing). *If* $\overline{X : L..U}, \Delta_2|\Gamma \vdash t : T$ *and* $\vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$ *and* $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$ *and* $\vdash \overline{T}$ witnessed *then* $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t : [\overline{T}/\overline{X}]T$.

Proof. By induction on the derivation of $\Delta|\Gamma \vdash t : T$, letting $\Delta = \overline{X : L..U}, \Delta_2$, and using that $\vdash \overline{T}$ ok by lemma 3 and $\mathrm{fv}(\overline{T}) = \emptyset$ by lemma 4.

T-Var. We have $t = x$ and $x : T \in \Gamma$. Then $x : [\overline{T}/\overline{X}]T \in [\overline{T}/\overline{X}]\Gamma$ and $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]x = x : [\overline{T}/\overline{X}]T$ by T-Var.

Case T-Null. $t =$ `null` and $\Delta \vdash T$ witnessed. By lemma 7, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ witnessed. Then $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]$`null` $=$ `null` $: [\overline{T}/\overline{X}]T$ by T-Null.

Case T-New. $t =$ `new` $N(\overline{t})$, $T = \exists.N$, $\Delta \vdash \exists.N$ witnessed, $\mathrm{fields}(N) = \overline{T'f}$, $\Delta|\Gamma \vdash \overline{t : S}$, and $\Delta \vdash \overline{S <: T'}$.

By i.h., $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]\overline{t} : [\overline{T}/\overline{X}]\overline{S}$. By lemma 5, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\overline{S} <: [\overline{T}/\overline{X}]\overline{T'}$. By lemma 7, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\exists.N$ witnessed. Moreover, fields$([\overline{T}/\overline{X}]N) = [\overline{T}/\overline{X}]\overline{T'f}$ by definition of fields. Then, $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]\mathsf{new}\ N(\overline{t}) : [\overline{T}/\overline{X}]\exists.N$ by T-New.

Case T-Field. We have $t = t_0.f_i$, $T = S_i$, $\Delta|\Gamma \vdash t_0 : T_0$, $\Delta \vdash T_0 <: \exists.N$, and fields$(N) = \overline{Sf}$. By i.h., $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_0 : [\overline{T}/\overline{X}]T_0$. By lemma 5, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T_0 <: [\overline{T}/\overline{X}]\exists.N$, and fields$([\overline{T}/\overline{X}]N) = [\overline{T}/\overline{X}]\overline{Sf}$ by definition of fields. Then $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_0.f_i : [\overline{T}/\overline{X}]S_i$ by T-Field as required.

Case T-Call. $t = t_0.\langle\overline{T'}\rangle m(\overline{t})$ and $\Delta|\Gamma \vdash \overline{T'}$ witnessed and $\Delta|\Gamma \vdash t_0 : T_0$ and $\Delta|\Gamma \vdash \overline{t : S}$ and $\Delta \vdash T_0 <: \exists.N$ and mtype$(m, N) = \langle\overline{Y \lhd U'}\rangle\overline{S'} \to S'$ and $\Delta \vdash \overline{S} <: [\overline{T'}/\overline{Y}]\overline{S'}$ and $\Delta \vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$ and finally $T = [\overline{T'}/\overline{Y}]S'$. Without loss of generality we can assume $\overline{X}$ and $\overline{Y}$ are disjoint.

By i.h., $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_0 : [\overline{T}/\overline{X}]T_0$ and $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]\overline{t} : [\overline{T}/\overline{X}]\overline{S}$. By lemma 5, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T_0 <: [\overline{T}/\overline{X}]\exists.N$, and mtype$([\overline{T}/\overline{X}]N) = [\overline{T}/\overline{X}](\langle\overline{Y \lhd U'}\rangle\overline{S'} \to S')$ by definition of mtype. Additionally, by lemma 7, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\overline{T'}$ witnessed, and by lemma 5, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\overline{S} <: [\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]\overline{S'}$ and $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\overline{T'} <: [\overline{T}/\overline{X}][\overline{T'}/\overline{Y}]\overline{U'}$.

Then $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_0.\langle\overline{T'}\rangle m(\overline{t}) : [\overline{T}/\overline{X}]T$ by T-Call as required.

Case T-Let. $t = \texttt{let}\ x : \exists\Delta'.N = t_1\ \texttt{in}\ t_2$ and $\Delta|\Gamma \vdash t_1 : T_1$ and $\Delta, \Delta'|\Gamma, x : \exists.N \vdash t_2 : T_2$ and $\Delta \vdash T_1 <: \exists\Delta'.N$ and $\Delta, \Delta' \vdash T_2 <: T$ and $\Delta \vdash T, \exists\Delta'.N$ ok and dom$(\Delta') \subseteq$ fv(N).

Without loss of generality we assume dom$(\Delta') \cap \overline{X} = \emptyset$ and therefore dom$(\Delta') \subseteq$ fv$([\overline{T}/\overline{X}]N)$. By i.h., $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_1 : [\overline{T}/\overline{X}]T_1$ and $[\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta'|[\overline{T}/\overline{X}]\Gamma, x : [\overline{T}/\overline{X}]\exists.N \vdash [\overline{T}/\overline{X}]t_2 : [\overline{T}/\overline{X}]T_2$.

Additionally, by lemma 6, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T$ ok and $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]\exists\Delta'.N$ ok, and by lemma 5, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T_1 <: [\overline{T}/\overline{X}]\exists\Delta'.N$ and $[\overline{T}/\overline{X}]\Delta_2, [\overline{T}/\overline{X}]\Delta' \vdash [\overline{T}/\overline{X}]T_2 <: [\overline{T}/\overline{X}]T$. Then $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t : [\overline{T}/\overline{X}]T$ by T-Let as required.

Case T-Elvis. $t = t_1\ ?{:}\ t_2$ and $\Delta|\Gamma \vdash t_1 : T_1$, $\Delta|\Gamma \vdash t_2 : T_2$ and $\Delta \vdash T_1 <: T$ and $\Delta \vdash T_2 <: T$.

By i.h., $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_1 : [\overline{T}/\overline{X}]T_1$ and $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_2 : [\overline{T}/\overline{X}]T_2$. By lemma 5, $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T_1 <: [\overline{T}/\overline{X}]T$ and $[\overline{T}/\overline{X}]\Delta_2 \vdash [\overline{T}/\overline{X}]T_2 <: [\overline{T}/\overline{X}]T$.

Then $[\overline{T}/\overline{X}]\Delta_2|[\overline{T}/\overline{X}]\Gamma \vdash [\overline{T}/\overline{X}]t_1\ ?{:}\ t_2 : [\overline{T}/\overline{X}]T$ by T-Elvis as required. □

Lemma 9 (Term substitution preserves typing). *If* $\Delta|\overline{x : T}, \Gamma_2 \vdash t : T$ *and* $\Delta|\emptyset \vdash \overline{t : S}$ *where* $\Delta \vdash \overline{S} <: \overline{T}$ *then* $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]t : S$ *for some* S *such that* $\Delta \vdash S <: T$.

Proof. By induction on the derivation of $\Delta|\Gamma \vdash t : T$ letting $\Gamma = \overline{x : T}, \Gamma_2$.

Case T-Var. $t = x$ and $x : T \in \Gamma$.

- If $x \in$ dom(Γ_2) then $[\overline{t}/\overline{x}]x = x$ and therefore $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]x : T$ by T-Var. Letting $S = T$ finishes the case since then $\Delta \vdash S <: T$ by S-Refl.
- Otherwise, $x = x_i$ and thus $T = T_i$. Letting $S = S_i$ finishes the case, since we're in particular assuming that $\Delta \vdash S_i <: T_i$.

Case T-Null. $t = \texttt{null}$ and $\Delta \vdash T$ witnessed. Since $[\overline{t}/\overline{x}]\texttt{null} = \texttt{null}$, $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]\texttt{null} : T$ by T-Null and letting $S = T$ finishes the case.

Case T-New. $t = \mathsf{new}\ N(\overline{s})$, $T = \exists.N$, $\Delta \vdash \exists.N$ witnessed, fields$(N) = \overline{Uf}$, $\Delta|\Gamma \vdash \overline{s : T'}$, and $\Delta \vdash \overline{T'} <: \overline{U}$.

By i.h., $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]\overline{s} : \overline{S'}$ with $\Delta \vdash \overline{S'} <: \overline{T'}$. Then $\Delta \vdash \overline{S'} <: \overline{U}$ by S-Trans and $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]\mathsf{new}\ N(\overline{s}) : \exists.N$ by T-New. Letting $S = \exists.N = T$ finishes the case.

Case T-Field. We have $t = t_0.f_i$, $T = U_i$, $\Delta|\Gamma \vdash t_0 : T_0$, $\Delta \vdash T_0 <: \exists.N$, and fields$(N) = \overline{Uf}$. By i.h., $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]t_0 : S_0$ with $\Delta \vdash S_0 <: T_0$. By S-Trans, $\Delta \vdash S_0 <: \exists.N$, Then $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]t_0.f_i : U_i$ by T-Field and letting $S = U_i = T$ finishes the case.

Case T-Call. $t = t_0.\langle\overline{T'}\rangle m(\overline{s})$ and $\Delta|\Gamma \vdash \overline{T'}$ witnessed and $\Delta|\Gamma \vdash t_0 : T_0$ and $\Delta|\Gamma \vdash \overline{s : T''}$ and $\Delta \vdash T_0 <: \exists.N$ and $\mathsf{mtype}(m, N) = \langle\overline{Y \lhd U'}\rangle\overline{S'} \to S'$ and $\Delta \vdash \overline{T''} <: [\overline{T'}/\overline{Y}]\overline{S'}$ and $\Delta \vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$ and finally $T = [\overline{T'}/\overline{Y}]S'$.

By i.h., $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]t_0 : S_0$ with $\Delta \vdash S_0 <: T_0$ and $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]\overline{s : S''}$ with $\Delta \vdash \overline{S''} <: \overline{T''}$. By S-Trans, $\Delta \vdash S_0 <: \exists.N$ and $\Delta \vdash \overline{S''} <: [\overline{T'}/\overline{Y}]\overline{S'}$. Then $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]t_0.\langle\overline{T'}\rangle m(\overline{s}) : [\overline{T'}/\overline{Y}]S'$ by T-Call and letting $S = [\overline{T'}/\overline{Y}]S' = T$ finishes the case.

Case T-Let. $t = \mathtt{let}\ x : \exists\Delta'.N = s_1\ \mathtt{in}\ s_2$ and $\Delta|\Gamma \vdash s_1 : S_1$ and $\Delta, \Delta'|\Gamma, x : \exists.N \vdash s_2 : S_2$ and $\Delta \vdash S_1 <: \exists\Delta'.N$ and $\Delta, \Delta' \vdash S_2 <: T$ and $\Delta \vdash T, \exists\Delta'.N$ ok and $\mathsf{dom}(\Delta') \subseteq \mathsf{fv}(N)$. Without loss of generality we assume that $x \notin \overline{x}$.

By i.h., $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]s_1 : S_1'$ with $\Delta \vdash S_1' <: S_1$ and $\Delta, \Delta'|\Gamma_2, x : \exists.N \vdash [\overline{t}/\overline{x}]s_2 : S_2'$ with $\Delta, \Delta' \vdash S_2' <: S_2$. By S-Trans, $\Delta \vdash S_1' <: \exists\Delta'.N$ and $\Delta, \Delta' \vdash S_2' <: T$. Then $\Delta|\Gamma \vdash [\overline{t}/\overline{x}]t : T$ by T-Let and letting $S = T$ finishes the case.

Case T-Elvis. $t = s_1\ ?{:}\ s_2$ and $\Delta|\Gamma \vdash s_1 : S_1$ and $\Delta|\Gamma \vdash s_2 : S_2$ and $\Delta \vdash S_1 <: T$ and $\Delta \vdash S_2 <: T$.

By i.h., $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]s_1 : S_1'$ with $\Delta \vdash S_1' <: S_1$ and $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]s_2 : S_2'$ with $\Delta \vdash S_2' <: S_2$.

Then $\Delta \vdash S_1' <: T$ and $\Delta \vdash S_2' <: T$ follow by S-Trans. Finally, $\Delta|\Gamma_2 \vdash [\overline{t}/\overline{x}]s_1\ ?{:}\ s_2 : T$, and letting $S = T$ finishes the case.

□

Lemma 10. *If* $\mathsf{mtype}(m, C\langle\overline{T}\rangle) = \langle\overline{Y \lhd U'}\rangle\overline{U} \to U$ *and* $\mathsf{mbody}(m\langle\overline{T'}\rangle, C\langle\overline{T}\rangle) = \overline{x}.t$ *where* $\vdash \exists.C\langle\overline{T}\rangle, \overline{T'}$ witnessed *and* $\vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$, *then there exist N and T such that* $\emptyset|\overline{x} : [\overline{T'}/\overline{Y}]\overline{U}, \mathtt{this} : \exists.N \vdash t : T$ *and* $\vdash \exists.C\langle\overline{T}\rangle <: \exists.N$ *and* $\vdash T <: [\overline{T'}/\overline{Y}]U$ *and* $\vdash \exists.N$ witnessed.

Proof. By induction on the derivation of $\mathsf{mbody}(m\langle\overline{T'}\rangle, C\langle\overline{T}\rangle) = \overline{x}.t$.

Case M-Class. $\mathtt{class}\ C\langle\overline{X \lhd U'''}\rangle \lhd N'\{\ldots\overline{M}\}$ and $\langle\overline{Y \lhd U''''}\rangle S\ m(\overline{S\ x}) = t_0 \in \overline{M}$ and $t = [\overline{T'}/\overline{Y}][\overline{T}/\overline{X}]t_0$. Without loss of generality we assume $\overline{X}$ and $\overline{Y}$ are disjoint.

Let $\Delta = \overline{X : \bot..U'''}, \overline{Y : \bot..U''''}$ and $\Gamma = \overline{x : S}, \mathtt{this} : \exists.C\langle\overline{X}\rangle$. By T-Class and T-Method, $\Delta|\Gamma \vdash t_0 : S_0$ and $\Delta \vdash S_0 <: S$. Since $\vdash \exists.C\langle\overline{T}\rangle$ witnessed, we know $\vdash \exists.C\langle\overline{T}\rangle$ ok and $\vdash \overline{T}$ witnessed by W-Direct and $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U'''}$ by WF-Class. By lemmas 3 and 4, $\mathsf{fv}(\overline{T}, \overline{T'}) = \emptyset$, and by S-Bot, $\vdash \bot <: \overline{T}, \overline{T'}$.

Then by lemmas 5 and 8, respectively, using that $[\overline{T}/\overline{X}]\exists.C\langle\overline{X}\rangle = \exists.C\langle\overline{T}\rangle$:

$$[\overline{T}/\overline{X}]\overline{Y : \bot..U''''} \vdash [\overline{T}/\overline{X}]S_0 <: [\overline{T}/\overline{X}]S$$

$$[\overline{T}/\overline{X}]\overline{Y : \bot..U''''}|\overline{x} : [\overline{T}/\overline{X}]\overline{S}, \mathtt{this} : \exists.C\langle\overline{T}\rangle \vdash [\overline{T}/\overline{X}]t_0 : [\overline{T}/\overline{X}]S_0$$

By M-Class, $\overline{U'} = [\overline{T}/\overline{X}]\overline{U''''}$, $\overline{U} = [\overline{T}/\overline{X}]\overline{S}$, and $U = [\overline{T}/\overline{X}]S$. Additionally, $\mathsf{fv}(\overline{T}, \overline{T'}) = \emptyset$ means that $[\overline{T'}/\overline{Y}]\overline{T} = \overline{T}$. By using lemmas 5 and 8 again and applying the above simplifications we get:

$$\vdash [\overline{T'}/\overline{Y}][\overline{T}/\overline{X}]S_0 <: [\overline{T'}/\overline{Y}]U$$

$$\emptyset|\overline{x} : [\overline{T'}/\overline{Y}]\overline{U}, \mathtt{this} : \exists.C\langle\overline{T}\rangle \vdash t : [\overline{T'}/\overline{Y}][\overline{T}/\overline{X}]S_0$$

Since $\vdash \exists.C\langle\overline{T}\rangle <: \exists.C\langle\overline{T}\rangle$ by S-Refl, letting $N = C\langle\overline{T}\rangle$ and $T = [\overline{T'}/\overline{Y}][\overline{T}/\overline{X}]S_0$ finishes the case.

Case M-Super. $\mathtt{class}\ C\langle\overline{X \lhd \_}\rangle \lhd N'\{\ldots\overline{M}\}$, $m \notin M$, and $\mathsf{mbody}(m\langle\overline{T'}\rangle, C\langle\overline{T}\rangle) = \mathsf{mbody}(m\langle\overline{T'}\rangle, [\overline{T}/\overline{X}]N')$.

By M-Super, $\mathsf{mtype}(m, C\langle\overline{T}\rangle) = \mathsf{mtype}(m, [\overline{T}/\overline{X}]N')$. By i.h., there exist N and T such that $\emptyset|\overline{x} : [\overline{T'}/\overline{Y}]\overline{U}, \mathtt{this} : \exists.N \vdash t : T$ and $\vdash [\overline{T}/\overline{X}]\exists.N' <: \exists.N$ and $\vdash T <: [\overline{T'}/\overline{Y}]U$ and $\vdash \exists.N$ witnessed.

Since $\vdash \exists.C\langle\overline{T}\rangle <: [\overline{T}/\overline{X}]\exists.N'$ by S-Extends, we get $\vdash \exists.C\langle\overline{T}\rangle <: \exists.N$ by S-Trans.

□

Lemma 11. *If* $\vdash \exists.N' <: \exists.N$ *and* $\mathsf{fields}(N) = \overline{T f}$ *then* $\mathsf{fields}(N') = \overline{T f}, \overline{U g}$.

Proof. We prove the following more general property by induction on $\vdash \exists\overline{Y : L'..U'}.N' <: \exists\overline{X : L..U}.N$: *If* $\vdash \exists\overline{Y : L'..U'}.N' <: \exists\overline{X : L..U}.N$ *and* $\mathsf{fields}(N) = \overline{S f}$ *then there exist* $\overline{T}$ *such that* $fv(\overline{T}) \subseteq \overline{Y}$ *and* $\mathsf{fields}(N') = [\overline{T}/\overline{X}]\overline{S f}, \overline{S' g}$.

S-Refl. $\overline{Y} = \overline{X}$ and $N' = N$. This case is immediate letting $\overline{T} = \overline{Y}$.

S-Bot, S-VarLeft, and S-VarRight don't apply.

S-Trans. $\vdash \exists\overline{Y : L'..U'}.N' <: K$ and $\vdash K <: \exists\overline{X : L..U}.N$.

It must be that $K = \exists\overline{Z : L''..U''}.N''$ because K can neither be $\bot$—since class types N can't be subtypes of $\bot$—nor X (because there are no free variables). Without loss of generality we assume that $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ are pairwise disjoint.

Then by applying the i.h. to the second premise, there exist $\overline{T''}$ such that $\mathsf{fv}(\overline{T''}) \subseteq \overline{Z}$ and $\mathsf{fields}(N'') = [\overline{T''}/\overline{X}]\overline{S f}, \overline{S'' g''}$.

Then by i.h. there also exist $\overline{T'}$ such that $\mathsf{fv}(\overline{T'}) \subseteq \overline{Y}$ and $\mathsf{fields}(N') = [\overline{T'}/\overline{Z}]([\overline{T''}/\overline{X}]\overline{S f}, \overline{S'' g''}), \overline{S''' g'''}$.

Letting $\overline{T} = [\overline{T'}/\overline{Z}]\overline{T''}$, we can rewrite $\mathsf{fields}(N') = [\overline{T}/\overline{X}]\overline{S f}$, $[\overline{T'}/\overline{Z}]\overline{S'' g''}, \overline{S''' g'''}$. Letting $\overline{S' g} = [\overline{T'}/\overline{Z}]\overline{S'' g''}, \overline{S''' g'''}$ and observing that $\mathsf{fv}(\overline{T}) \subseteq \overline{Y}$ finishes the case.

S-Extends. $\vdash \exists\overline{X : L..U}.C\langle\overline{T'}\rangle <: \exists\overline{X : L..U}.[\overline{T'}/\overline{Z}]N''$ and `class` $C\langle\overline{Z \lhd \_}\rangle \lhd N''\{\overline{S''\ g;}\ldots\}$, and thus $N' = C\langle\overline{T'}\rangle$ and $N = [\overline{T'}/\overline{Z}]N''$ and $\overline{Y} = \overline{X}$.

By rule F-Class, $\mathsf{fields}(C\langle\overline{T'}\rangle) = \overline{S f}, [\overline{T'}/\overline{Z}]\overline{S'' g}$, where $\mathsf{fields}(N) = \overline{S f}$. Letting $\overline{T} = \overline{Y}$ and $\overline{S'} = [\overline{T'}/\overline{Z}]\overline{S''}$ therefore finishes the case.

S-Exists. $N' = [\overline{T}/\overline{X}]N$ and $\mathsf{fv}(\overline{T}) \subseteq \overline{Y}$. This case is immediate with the given $\overline{T}$, since $\mathsf{fields}(N') = [\overline{T}/\overline{X}]\mathsf{fields}(N)$. □

Lemma 12. *If* $\vdash \exists.N' <: \exists.N$ *and* $\mathsf{mtype}(m, N) = \langle\overline{X \lhd U}\rangle\overline{T} \to T$ *is defined then* $\mathsf{mtype}(m, N') = \langle\overline{X \lhd U}\rangle\overline{T} \to T$

Proof. We prove the following more general property by induction on $\vdash \exists\overline{Y : L'..U'}.N' <: \exists\overline{X : L..U}.N$: *If* $\vdash \exists\overline{Y : L'..U'}.N' <: \exists\overline{X : L..U}.N$ *and* $\mathsf{mtype}(m, N) = \langle\overline{X' \lhd S'}\rangle\overline{S} \to S$ *and* $\overline{X}, \overline{Y} \cap \overline{X'} = \emptyset$ *then there exist* $\overline{T}$ *such that* $fv(\overline{T}) \subseteq \overline{Y}$ *and* $\mathsf{mtype}(m, N') = [\overline{T}/\overline{X}]\mathsf{mtype}(m, N)$.

S-Refl. $\overline{Y} = \overline{X}$ and $N' = N$. This case is immediate letting $\overline{T} = \overline{Y}$.

S-Bot, S-VarLeft, and S-VarRight don't apply.

S-Trans. $\vdash \exists\overline{Y : L'..U'}.N' <: K$ and $\vdash K <: \exists\overline{X : L..U}.N$.

It must be that $K = \exists\overline{Z : L''..U''}.N''$ because K can neither be $\bot$—since class types N can't be subtypes of $\bot$—nor X (because there are no free variables). Without loss of generality we assume that $\overline{X}$, $\overline{Y}$, and $\overline{Z}$ are pairwise disjoint.

Then by applying the i.h. to the second premise, there exist $\overline{T''}$ such that $\mathtt{fv}(\overline{T''}) \subseteq \overline{Z}$ and $\mathsf{mtype}(m, N'') = [\overline{T''}/\overline{X}]\mathsf{mtype}(m, N)$.

Then by i.h. there also exist $\overline{T'}$ such that $\mathtt{fv}(\overline{T'}) \subseteq \overline{Y}$ and $\mathsf{mtype}(m, N') = [\overline{T'}/\overline{Z}][\overline{T''}/\overline{X}]\mathsf{mtype}(m, N)$.

Letting $\overline{T} = [\overline{T'}/\overline{Z}]\overline{T''}$, we can rewrite $\mathsf{mtype}(m, N') = [\overline{T}/\overline{X}]\mathsf{mtype}(m, N)$. Observing that $\mathtt{fv}(\overline{T}) \subseteq \overline{Y}$ finishes the case.

S-Extends. $N' = C\langle\overline{T''}\rangle$, $N = [\overline{T''}/\overline{Z}]N''$, $\overline{Y} = \overline{X}$, `class` $C\langle\overline{Z \lhd U''}\rangle \lhd N''\{\ldots\overline{M}\}$, and $\overline{Z} \cap \overline{X} = \emptyset$.

If $m \notin \overline{M}$ then $\mathsf{mtype}(m, N) = \mathsf{mtype}(m, N')$ by M-Super and the conclusion is immediate with $\overline{T} = \overline{Y}$.

Otherwise, $m \in \overline{M}$. By induction on the derivation of mtype we can show that $\mathsf{mtype}(m, N') = [\overline{T''}/\overline{Z}]\mathsf{mtype}(m, N'') = [\overline{T''}/\overline{Z}]\langle\overline{X' \lhd U'''}\rangle\overline{T'''} \to T'''$. Without loss of generality we can assume $\overline{X'}$ is disjoint from $\overline{Z}$, which means that $\overline{S'} = [\overline{T''}/\overline{Z}]\overline{U'''}$, $\overline{S} = [\overline{T''}/\overline{Z}]\overline{T'''}$, and $S = [\overline{T''}/\overline{Z}]T'''$. By T-Method and the definition of override, it must be that $\langle\overline{X' \lhd U'''}\rangle T'''\ m(\overline{T'''\ x}) = t \in \overline{M}$.

Without loss of generality we also assume $\overline{Z} \cap \overline{X} = \emptyset$. By M-Class and using the above equations we know: $\mathsf{mtype}(m, C\langle\overline{T''}\rangle) = [\overline{T''}/\overline{Z}]\langle\overline{X' \lhd U'''}\rangle\overline{T'''} \longrightarrow T''' = \langle\overline{X' \lhd S'}\rangle\overline{S} \longrightarrow S = \mathsf{mtype}(m, N')$. Letting $\overline{T} = \overline{Y}$ therefore finishes the case.

S-Exists. $N' = [\overline{T}/\overline{X}]N$ and $\mathsf{fv}(\overline{T}) \subseteq \overline{Y}$. This case is immediate with the given $\overline{T}$ since $\mathsf{mtype}(m, [\overline{T}/\overline{X}]N) = [\overline{T}/\overline{X}]\mathsf{mtype}(m, N)$. □

Proof of theorem 1 (Preservation). By induction on the derivation of $t \longmapsto t'$.

Case Read. $t = \mathsf{new}\ N(\overline{V}).f_i$, $t' = V_i$, and $\mathsf{fields}(N) = \overline{T f}$.

By T-New, $\vdash \mathsf{new}\ N(\overline{V}) : \exists.N$ and $\vdash \overline{V : S}$ and $\vdash \overline{S} <: \overline{T}$. By T-Field, $\vdash \exists.N <: \exists.N'$ and $\mathsf{fields}(N') = \overline{Ug}$ and $T = U_i$.

By lemma 11, $\mathsf{fields}(N) = \overline{Ug}, \overline{U'g'}$. Therefore, $f_i = g_i$ and $T = T_i = U_i$. Letting $T' = S_i$ finishes the case, since in particular $\vdash V_i : S_i$ and $\vdash S_i <: T_i$ per above.

Case Present. $t = \mathsf{new}\ N(\overline{V})\ ?{:}\ t_2$ and $t' = \mathsf{new}\ N(\overline{V})$.

From T-Elvis and T-New we know $\vdash \mathsf{new}\ N(\overline{V}) : \exists.N$ and $\exists.N <: T$. Letting $T' = \exists.N$ therefore finishes the case.

Case Absent. $t = \mathtt{null}\ ?{:}\ t_2$ and $t' = t_2$.

From T-Elvis we know $\vdash t_2 : T_2$ and $\vdash T_2 <: T$. Thus letting $T' = T_2$ finishes the case.

Case Invoke. $t = \mathsf{new}\ N(\overline{V}).\langle\overline{T}\rangle m(\overline{V'})$, $t' = [\overline{V'/x}, \mathsf{new}\ N(\overline{V})/\mathtt{this}]t_0$, and $\mathsf{mbody}(m\langle\overline{T}\rangle, N) = \overline{x}.t_0$.

By T-New, $\vdash \mathsf{new}\ N(\overline{V}) : \exists.N$ and $\vdash \exists.N$ witnessed. By T-Call, $\vdash \overline{T}$ witnessed, $\vdash \exists.N <: \exists.N'$, $\mathsf{mtype}(m, N') = \langle\overline{X \lhd U'}\rangle\overline{U} \longrightarrow U$, $\vdash \overline{V' : S}$, $\vdash \overline{S} <: [\overline{T}/\overline{X}]\overline{U}$, $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U'}$, and finally $T = [\overline{T}/\overline{X}]U$.

By lemma 12, $\mathsf{mtype}(m, N) = \langle\overline{X \lhd U'}\rangle\overline{U} \longrightarrow U$. By lemma 10, $\emptyset|\overline{x} : [\overline{T}/\overline{X}]\overline{U}, \mathtt{this} : \exists.N'' \vdash t_0 : S$ such that $\vdash \exists.N <: \exists.N''$, and $\vdash S <: [\overline{T}/\overline{X}]U$. Then by lemma 9, $\vdash t' : T'$ for some T' such that $\vdash T' <: S$.

Then $\vdash T' <: [\overline{T}/\overline{X}]U = T$ follows by S-Trans.

Case Unpack. $t = \mathtt{let}\ x : \exists\overline{X : L..U}.N = \mathsf{new}\ N'(\overline{V})\ \mathtt{in}\ t_2$, $t' = [\mathsf{new}\ N'(\overline{V})/x][\overline{T}/\overline{X}]t_2$, $\vdash \exists.N' <: [\overline{T}/\overline{X}]\exists.N$, $\vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$, $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$, and $\vdash \overline{T}$ witnessed.

By T-New, $\vdash \mathsf{new}\ N'(\overline{V}) : \exists.N'$ and $\vdash \exists.N'$ witnessed. By T-Let, $\vdash \exists.N' <: \exists\overline{X : L..U}.N$ and $\overline{X : L..U}|x : \exists.N \vdash t_2 : T_2$ and $\overline{X : L..U} \vdash T_2 <: T$ and $\vdash T, \exists\overline{X : L..U}.N$ ok and $\overline{X} \subseteq \mathsf{fv}(N)$.

By lemma 8, $\emptyset|x : [\overline{T}/\overline{X}]\exists.N \vdash [\overline{T}/\overline{X}]t_2 : [\overline{T}/\overline{X}]T_2$. Then, by lemma 9, $\vdash [\mathsf{new}\ N'(\overline{V})/x][\overline{T}/\overline{X}]t_2 = t' : S$ where $\vdash S <: [\overline{T}/\overline{X}]T_2$.

Moreover, by lemmas 3, 4 and 5, $\vdash [\overline{T}/\overline{X}]T_2 <: [\overline{T}/\overline{X}]T$. Since $\vdash T$ ok, by lemma 4, T can't have free type variables, and thus $[\overline{T}/\overline{X}]T = T$. Then $\vdash S <: T$ by S-Trans and letting $T' = S$ finishes the case.

Case Stub. $t = \mathtt{let}\ x : \exists\overline{X : L..U}.N = \mathtt{null}\ \mathtt{in}\ t_2$, $t' = [\mathtt{null}/x][\overline{T}/\overline{X}]t_2$, $\vdash \exists.N' <: [\overline{T}/\overline{X}]\exists.N$, $\vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$, $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$, and $\vdash \exists.N', \overline{T}$ witnessed.

By T-Let, $\vdash T_1 <: \exists\overline{X : L..U}.N$ and $\overline{X : L..U}|x : \exists.N \vdash t_2 : T_2$ and $\overline{X : L..U} \vdash T_2 <: T$ and $\vdash T, \exists\overline{X : L..U}.N$ ok and $\overline{X} \subseteq \mathsf{fv}(N)$.

By lemma 8, $\emptyset|x : [\overline{T}/\overline{X}]\exists.N \vdash [\overline{T}/\overline{X}]t_2 : [\overline{T}/\overline{X}]T_2$. By T-Null, $\vdash \mathtt{null} : \exists.N'$. Then, by lemma 9, $\vdash [\mathtt{null}/x][\overline{T}/\overline{X}]t_2 = t' : S$ where $\vdash S <: [\overline{T}/\overline{X}]T_2$. The remainder of the case proceeds like Unpack.

Case C-Field. $t = t_0.f_i$, $t' = t_0'.f_i$, and $t_0 \longmapsto t_0'$.

By T-Field, $\vdash t_0 : T_0$ and $\vdash T_0 <: \exists.N$ and $\mathsf{fields}(N) = \overline{U\ f}$ and $T = U_i$. By i.h., $\vdash t_0 : T_0'$ where $\vdash T_0' <: T_0$. Then $\vdash T_0' <: \exists.N$ by S-Trans and $\vdash t_0'.f_i : U_i$ by T-Field. Letting $T' = U_i = T$ finishes the case.

Case C-New. $t = \mathsf{new}\ N(\overline{V}, t_i, \overline{t})$, $t' = \mathsf{new}\ N(\overline{V}, t_i', \overline{t})$, and $t_i \longmapsto t_i'$.

By T-New, $\vdash \exists.N$ witnessed and $\mathsf{fields}(N) = \overline{U\ f}$ and $\vdash \overline{V}, t_i, \overline{t} : \overline{T}$ and $\vdash \overline{T} <: \overline{U}$ and $T = \exists.N$. By i.h., $\vdash t_i' : T_i'$ where $\vdash T_i' <: T_i$. Then $\vdash T_i' <: U_i$ by S-Trans and $\vdash \mathsf{new}\ N(\overline{V}, t_i', \overline{t}) : T$ by T-New. Letting $T' = T$ finishes the case.

Cases C-Receiver, C-Let, and C-Elvis similar to C-Field.

Case C-Arg similar to C-New.

□

B PROOF OF THEOREM 2 (PROGRESS)

Lemma 13 (Witness). *If* $\vdash \exists.N <: \exists\overline{Y : L'..U'}.N'$ *and* $\vdash \exists.N$ witnessed *then there exist* $\overline{T'}$ *such that* $fv(\overline{T'}) = \emptyset$ *and* $\vdash \exists.N <: [\overline{T'}/\overline{Y'}]\exists.N'$ *and* $\vdash [\overline{T'}/\overline{Y'}]\exists.N' <: \exists\overline{Y : L'..U'}.N'$ *and* $\vdash [\overline{T'}/\overline{Y}]\overline{L'} <: \overline{T'}$ *and* $\vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$ *and* $\vdash [\overline{T'}/\overline{Y}]\exists.N'$ witnessed.

Proof. We prove the following more general property by induction on derivation of $\vdash \exists\overline{X : L..U}.N <: \exists\overline{Y : L'..U'}.N'$, often writing $\vdash S <: T <: U$ as shorthand for $\vdash S <: T$ and $\vdash T <: U$: *If* $\vdash [\overline{T}/\overline{X}]\exists.N <: \exists\overline{X : L..U}.N <: \exists\overline{Y : L'..U'}.N'$ *and* $\vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$ *and* $fv(\overline{T}) = \emptyset$ *and* $\vdash [\overline{T}/\overline{X}]\exists.N$ witnessed *then there exist* $\overline{T'}$ *such that* $fv(\overline{T'}) = \emptyset$ *and* $\vdash [\overline{T}/\overline{X}]\exists.N <: [\overline{T'}/\overline{Y}]\exists.N' <: \exists\overline{Y : L'..U'}.N'$ *and* $\vdash [\overline{T'}/\overline{Y}]\overline{L'} <: \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$ *and* $\vdash [\overline{T'}/\overline{Y}]\exists.N'$ witnessed.

Case S-Refl immediate with $\overline{T'} = \overline{T}$.

Case S-Bot doesn't apply because $\bot$ is not a type.

Case S-Trans. We have $\vdash \exists\overline{X : L..U}.N <: K <: \exists\overline{Y : L'..U'}.N'$. It must be that $K = \exists\overline{Z : L''..U''}.N''$ because K can neither be $\bot$—since class types N can't be subtypes of $\bot$—nor X (because there are no free variables).

Then by applying the i.h. to the first premise, there exist $\overline{T''}$ such that $\mathsf{fv}(\overline{T''}) = \emptyset$, $\vdash [\overline{T}/\overline{X}]\exists.N <: [\overline{T''}/\overline{Z}]\exists.N'' <: K$, $\vdash [\overline{T''}/\overline{Z}]\overline{L''} <: \overline{T''} <: [\overline{T''}/\overline{Z}]\overline{U''}$, and $[\overline{T''}/\overline{Z}]\exists.N''$ witnessed.

Then by i.h. there also exist $\overline{T'}$ such that $\mathsf{fv}(\overline{T'}) = \emptyset$, $\vdash [\overline{T''}/\overline{Z}]\exists.N'' <: [\overline{T'}/\overline{Y}]\exists.N' <: \exists\overline{Y : L'..U'}.N'$, $\vdash [\overline{T'}/\overline{Y}]\overline{L'} <: \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$, and $\vdash [\overline{T'}/\overline{Y}]\exists.N'$ witnessed

Then, $\vdash [\overline{T}/\overline{X}]\exists.N <: [\overline{T'}/\overline{Y}]\exists.N'$ follows from S-Trans, and $\overline{T'}$ satisfy the conditions of the needed substitution.

Cases S-VarLeft and S-VarRight don't apply because there are no free variables.

Case S-Extends. $\vdash \exists\overline{X : L..U}.C\langle\overline{S}\rangle <: \exists\overline{X : L..U}.[\overline{S}/\overline{Z}]N''$ and $\mathtt{class}\ C\langle\overline{Z \triangleleft U''}\rangle \triangleleft N''\{\ldots\}$ and $\overline{X} \cap \overline{Z} = \emptyset$, i.e., $N = C\langle\overline{S}\rangle$ and $N' = [\overline{S}/\overline{Z}]N''$ and $\overline{X} = \overline{Y}$ and $\overline{L} = \overline{L'}$ and $\overline{U} = \overline{U'}$.

By S-Extends, $\vdash \exists.[\overline{T}/\overline{X}]C\langle\overline{S}\rangle <: \exists.[\overline{T}/\overline{X}][\overline{S}/\overline{Z}]N'' = \exists.[\overline{T}/\overline{X}]N'$. By S-Exists, $\exists.[\overline{T}/\overline{X}]N' <: \exists\overline{X : L..U}.N'$.

By T-Class, $\overline{Z : \bot..U''} \vdash \exists.N''$ witnessed. Since $\vdash [\overline{T}/\overline{X}]\exists.C\langle\overline{S}\rangle$ witnessed, we know $\vdash [\overline{T}/\overline{X}]\exists.C\langle\overline{S}\rangle$ ok and $\vdash [\overline{T}/\overline{X}]\overline{S}$ witnessed per W-Direct and $\vdash [\overline{T}/\overline{X}]\overline{S} <: [[\overline{T}/\overline{X}]\overline{S}/\overline{Z}]\overline{U''}$ per WF-Class. Since also $\vdash \bot <: [\overline{T}/\overline{X}]\overline{S}$ by S-Bot, we get $\vdash [[\overline{T}/\overline{X}]\overline{S}/\overline{Z}]\exists.N''$ witnessed by lemma 7, which by rewriting is the same as $\vdash [\overline{T}/\overline{X}]\exists.N'$ witnessed. Thus, letting $\overline{T'} = \overline{T}$ finishes the case.

Case S-Exists. $\vdash \exists\overline{X : L..U}.[\overline{T''}/\overline{Y}]N' <: \exists\overline{Y : L'..U'}.N'$ and thus $N = [\overline{T''}/\overline{Y}]N'$, and also $\mathsf{fv}(\overline{T''}) \subseteq \overline{X}$ and $\overline{X} \cap \mathsf{fv}(\exists\overline{Y : L'..U'}.N') = \emptyset$ and $\overline{X : L..U} \vdash [\overline{T''}/\overline{Y}]\overline{L'} <: \overline{T''} <: [\overline{T''}/\overline{Y}]\overline{U'}$.

Let $\overline{T'} = [\overline{T}/\overline{X}]\overline{T''}$ and assume without loss of generality that $\overline{X}$ and $\overline{Y}$ are disjoint. Also note that $\overline{X}$ can't be free in $N', \overline{L'}, \overline{U'}$ and therefore $[\overline{T}/\overline{X}](N', \overline{L'}, \overline{U'}) = N', \overline{L'}, \overline{U'}$. Then by lemma 5, $\vdash [\overline{T'}/\overline{Y}]\overline{L'} <: \overline{T'}$ and $\vdash \overline{T'} <: [\overline{T'}/\overline{Y}]\overline{U'}$. Also $\mathsf{fv}(\overline{T'}) = \emptyset$ and therefore $\vdash [\overline{T'}/\overline{Y}]\exists.N' <: \exists\overline{Y : L'..U'}.N'$ by S-Exists.

Since $N = [\overline{T''}/\overline{Y}]N'$ and we assume $\vdash [\overline{T}/\overline{X}]\exists.N$ witnessed we get $\vdash [\overline{T}/\overline{X}][\overline{T''}/\overline{Y}]\exists.N'$ witnessed and therefore $\vdash [\overline{T'}/\overline{Y}]\exists.N'$ witnessed by rewriting.

$N = [\overline{T''}/\overline{Y}]N'$ also gives us $\vdash [\overline{T}/\overline{X}]\exists.N <: [\overline{T'}/\overline{Y}]\exists.N'$ by S-Refl, and it follows that $\overline{T'}$ satisfy the conditions of the needed substitution.

□

Lemma 14. *If* $\vdash T$ witnessed *then there exists* N *such that* $\vdash \exists.N <: T$ *and* $\exists.N$ witnessed.

Proof. By case analysis of $\vdash T$ witnessed.
T-Var. Doesn't apply: no variables in scope.
T-Direct. $T = \exists.N$. Immediate since $\vdash T <: T$ by S-Refl.
T-Wildcard. Immediate: the rule's premises include the needed properties. □

Lemma 15. *If* $\Delta \vdash [\overline{T}/\overline{X}]T$ witnessed *and* $\overline{X} \cap \mathsf{dom}(\Delta) = \emptyset$ *then* $\Delta \vdash T_i$ witnessed *for all* $X_i \in \overline{X} \cap \mathsf{fv}(T)$.

Proof. By induction on the derivation of $\Delta \vdash [\overline{T}/\overline{X}]T$ witnessed.
W-Var. $T = X$ and therefore $\mathsf{fv}(T) = X$.

- If $X = X_i \in \overline{X}$ then $[\overline{T}/\overline{X}]X_i = T_i$. Since $\Delta \vdash [\overline{T}/\overline{X}]X_i$ witnessed by assumption, $\Delta \vdash T_i$ witnessed follows by rewriting.
- Otherwise, $X \notin \overline{X}$ and conclusion holds vacuously.

W-Direct. $T = \exists.C\langle\overline{T'}\rangle$ and $\Delta \vdash [\overline{T}/\overline{X}]\overline{T'}$ witnessed. By i.h., $\Delta \vdash T_i$ witnessed for all T_i where $X_i \in \mathsf{fv}(\overline{T'})$, which is the needed property since $\overline{X} \cap \mathsf{fv}(T) = \overline{X} \cap \mathsf{fv}(\overline{T'})$.

W-Wildcard. $T = \exists\Delta'.C\langle\overline{T'}\rangle$ and $\Delta' = \overline{Y : L'..U'}$ and $\Delta, \Delta' \vdash [\overline{T}/\overline{X}](\overline{T'}, \overline{L'}, \overline{U'})$ witnessed. Without loss of generality we can assume that $\overline{X} \cap \overline{Y} = \emptyset$. By i.h., $\Delta, \Delta' \vdash T_i$ witnessed for all T_i where $X_i \in \mathsf{fv}(\overline{T'}, \overline{L'}, \overline{U'})$, which is the needed property. □

Proof of theorem 2 (Progress). By induction on typing derivation $\vdash t : T$.
Case T-Var. cannot occur: t is closed.
Case T-Null. $t = \texttt{null}$ is a value.
Case T-New. $t = \mathsf{new}\ N(\overline{t})$. By i.h., either all $\overline{t}$ are values or there is a smallest i such that t_i err or $t_i \longmapsto t_i'$ can take a step. If all $\overline{t}$ are values then t is also a value. Else if t_i err then t err by Err-New. Otherwise, $t \longmapsto \mathsf{new}\ N(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n)$ per C-New.
Case T-Field. We have $t = t_0.f_i$ and $\vdash t_0 : T_0$ and $\vdash T_0 <: \exists.N$ and $\mathsf{fields}(N) = \overline{T\ f}$.
By i.h., t_0 is either a value, or else t_0 err, or else $t_0 \longmapsto t_0'$. We distinguish the following cases:

- If t_0 err the t err by Err-Field.
- If t_0 can step then $t \longmapsto t_0'.f_i$ by C-Field.
- If $t_0 = \texttt{null}$ then t err by Err-Read.
- Otherwise, it must be $t_0 = \mathsf{new}\ N'(\overline{V})$ and thus $T_0 = \exists.N'$ per T-New. By lemma 11, $\mathsf{fields}(m, N') = \overline{T\ f}, \overline{U\ g}$, and therefore $t \longmapsto V_i$ by Read.

Case T-Call. We have $t = t_0.\langle\overline{T}\rangle m(\overline{t})$, $\vdash \overline{T}$ witnessed, $\vdash t_0 : T_0$, $\vdash \overline{t : S}$, $\vdash T_0 <: \exists.N$, $\mathsf{mtype}(m, N) = \langle\overline{X \lhd U'}\rangle\overline{U} \to U$, $\vdash \overline{S} <: [\overline{T}/\overline{X}]\overline{U}$, and $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U'}$.
By i.h., t_0 and $\overline{t}$ are either all values or there is a smallest $i \geq 0$ such that t_i err or $t_i \longmapsto t_i'$ can take a step. We distinguish the following cases:

- If $t_0 = \texttt{null}$ and all $\overline{t}$ are values then t err by Err-Invoke.
- Else if all $t_0, \overline{t}$ are values then it must be that $t_0 = \mathsf{new}\ N'(\overline{V})$. Per T-New, $T_0 = \exists.N'$ and $\vdash \exists.N'$ witnessed. By lemma 12, $\mathsf{mtype}(m, N') = \langle\overline{X \lhd U'}\rangle\overline{U} \to U'$. Since $\mathsf{mtype}(m, N')$ is defined, $\mathsf{mbody}(m\langle\overline{T}\rangle, N') = \overline{x}.t_b$ must also be defined, and $\overline{x}$ must have the same cardinality as $\overline{U}$. Then $t \longmapsto [\overline{t}/\overline{x}, t_0/\texttt{this}]t_b$ by Invoke.
- Else if t_i err then t err by Err-Receiver if $i = 0$ or by Err-Arg otherwise.

- Else if t_0 can step then $t \longmapsto t'_0.\langle\overline{T}\rangle m(\overline{t})$ by C-Receiver.
- Otherwise t_i with $i > 0$ can step and $t \longmapsto t_0.\langle\overline{T}\rangle m(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n)$ by C-Arg.

Case T-Let. We have $t = \texttt{let}\ x : \exists\Delta.N = t_1\ \texttt{in}\ t_2$, $\vdash t_1 : T_1$, $\vdash T_1 <: \exists\Delta.N$, and $\overline{X} \subseteq \mathsf{fv}(N)$, letting $\Delta = \overline{X : L..U}$.

By i.h., t_1 is value or else t_1 err or else $t_1 \longmapsto t'_1$ for some t'_1:

- If t_1 err then t err by Err-Let.
- Else if t_1 can step then $t \longmapsto \texttt{let}\ x : \exists\Delta.N = t'_1\ \texttt{in}\ t_2$ by C-Let.
- Otherwise, t_1 is a value, i.e., either $\texttt{null}$ or new. If $t_1 = \texttt{null}$ then $\vdash \texttt{null} : T_1$ and $\vdash T_1$ witnessed per T-Null. Then by lemma 14 there exists N' such that $\vdash \exists.N' <: T_1$ and $\vdash \exists.N'$ witnessed, and we can derive $\vdash \exists.N' <: \exists\Delta.N$ by S-Trans. Otherwise it must be $t_1 = \mathsf{new}\ N'(\overline{V})$ and therefore $T_1 = \exists.N'$ and $\vdash \exists.N'$ witnessed per T-New.
  Either way, by lemma 13 there exist $\overline{T}$ such that $\mathsf{fv}(\overline{T}) = \emptyset$, $\vdash \exists.N' <: [\overline{T}/\overline{X}]\exists.N$, $\vdash [\overline{T}/\overline{X}]\overline{L} <: \overline{T}$, $\vdash \overline{T} <: [\overline{T}/\overline{X}]\overline{U}$, and $\vdash [\overline{T}/\overline{X}]\exists.N$ witnessed. Since all $\overline{X}$ are free in N, by lemma 15, $\vdash \overline{T}$ witnessed. Finally, $t \longmapsto [t_1/x][\overline{T}/\overline{X}]t_2$ by Stub if $t_1 = \texttt{null}$ and by Unpack otherwise.

Case T-Elvis. We have $t = t_1 \mathrel{?:} t_2$ and $\vdash t_1 : T_1$ (among other facts). By i.h., t_1 is value or t_1 err or $t_1 \longmapsto t'_1$ for some t'_1. Then we have the following:

- If t_1 err then t err by Err-Elvis.
- if t_1 can step then $t \longmapsto t'_1 \mathrel{?:} t_2$ by C-Elvis.
- Else if $t_1 = \texttt{null}$ then $t \longmapsto t_2$ by Absent.
- Else $t_1 = \mathsf{new}\ N(\overline{V})$, and then $t \longmapsto \mathsf{new}\ N(\overline{V})$ by Present.

□