Type-Outference with Label-Listeners

Foundations for Decidable Type-Consistency for Nominal Object-Oriented Generics

ROSS TATE*, Independent Researcher and Consultant, USA

We present the first sound and complete decision procedure capable of validating nominal object-oriented programs without type annotations in expressions even in the presence of generic interfaces with irregularly-recursive signatures. Such interface signatures are exemplary of the challenges an algorithm must overcome in order to scale to the needs of modern major typed object-oriented languages. We do so by incorporating event-driven *label-listeners* into our constraint system, enabling more complex aspects of program validation to be generated on demand while still ensuring termination. Furthermore, we define *type-consistency* as a novel declarative notion of program validity that ensures safety without requiring a complex grammar of types to perform inference within. While type-inferability ensures type-consistency, the converse does not necessarily hold. Thus our algorithm decides program validity with*out* inferring the types missing from the program, and so we instead call it a *type-outference* algorithm. By bypassing type-inference, the proofs involving type-consistency more directly connect the design of and reasoning about constraints to the operational semantics of the language, simplifying much of the design and verification process. We mechanically formalize and verify these concepts and techniques—type-consistency, type-outference, and label-listeners—in Rocq to provide a solid foundation for scaling decidability beyond the limitations of structural types.

CCS Concepts: • **Software and its engineering → Compilers**; **Semantics**; **Object oriented languages**; • **Theory of computation → Type structures**.

Additional Key Words and Phrases: Type-Consistency, Type-Outference, Type-Inference, Decidability, Generics

ACM Reference Format:
Ross Tate. 2025. Type-Outference with Label-Listeners: Foundations for Decidable Type-Consistency for Nominal Object-Oriented Generics. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 408 (October 2025), 27 pages. https://doi.org/10.1145/3763797

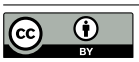
1 Introduction

When faced with the challenge of validating a complex unannotated expression in a type-safe language, type-inference immediately comes to mind. That is, one can ensure safety by determining how to ascribe types to the components of the expression so that it type-checks. This is so commonplace that one often assumes validity of such programs is defined as type-inferability. Consequently, decidability of program validity is often taken as decidability of type-inferability.

But when developing such decision procedures, one often encounters a problematic class of programs: programs that have no contradictions but also no obvious ascribable types. Given these programs, type-inferability forces a design choice: add more types to ascribe to these programs, or add a stage to determine if there are non-obvious ascriptions. The former complicates the language and proofs. The latter is not always easy to decide. In both cases, the grammar of types seems to play an overbearing role, especially given that grammars tend to change over time.

*As consultant for the Kotlin team at JetBrains

Author's Contact Information: Ross Tate, research@rosstate.org, Independent Researcher and Consultant, Ithaca, NY, USA.



ACM 2475-1421/2025/10-ART408
https://doi.org/10.1145/3763797

Here we introduce a novel notion of program validity for unannotated programs: *type-consistency*. A type-consistent program is one that can be type-checked within *some* "consistent type space". A type space is akin to a grammar, so type-consistency allows one to accept a program by constructing a grammar specific to that program (e.g. with abstract types representing subexpressions in the program) provided the behavior of those types (e.g. assumed subtypings corresponding to flows in the program) can be shown to be *consistent* (e.g. respect the operational semantics of the language). In particular, the user-expressible types form a consistent type space, as does any safe grammar of extended types, so type-consistency includes any notion of type-inferability.

Even though type-consistency is more permissive than type-inferability, we determined it is still safe, and we found it to be easier to decide soundly and completely. Because a decision procedure for type-consistency cannot be called a type-inference algorithm as it does not decide type-inferability, we instead call it a *type-outference* algorithm as a pun on how such algorithms will generally work by factoring unknown types *out* of constraints rather than filling them *in*. We apply these insights concretely to nominal object-oriented generics, for which decidable type-inference has been an open problem for decades despite significant relevance to many major industry languages. Although type-consistency does not solve the problem entirely on its own, we found that it facilitates the development of techniques and their proofs of correctness. In particular, we will illustrate how the extended grammars used for type-inferability caused unexpected problems for the mechanical verification of early attempts at this work. Because type-consistency bypasses the need for such extended grammars entirely, it enabled us to develop a much simpler mechanical formalization and verification of all the definitions and theorems presented here [Tate 2025].

In order to focus on high-level foundational concepts, here we formally address only one key challenge of object-oriented generics: arbitrarily-recursive generic interfaces. For example, our interfaces are restricted to monomorphic methods, and inheritance is restricted to just having a common type that all interfaces inherit. Yet, despite these restrictions, decidable validation is still quite challenging because *structural* subtyping with such interfaces is undecidable [DeYoung et al. 2024]. Thus, we exploit *nominality* in order to overcome the algorithmic limitations of structural types. In particular, we introduce *label-listeners* as a controlled means for dynamically adding type variables and constraints when lower bounds on the *label*—e.g. interface *name*—of an unknown type are derived. Thus, although our calculus is minimal, our techniques provide critical foundations for exploiting nominality to support more advanced features of object-oriented generics.

The paper is divided into two major parts: the intuition and the formalization. In Section 2, we demonstrate how arbitrarily-recursive generic interfaces violate a key premise of existing algorithmic approaches. In Section 3, we discuss how incorporating event-driven concepts like *label-listeners* enables our algorithm to support this feature. In Section 4, we illustrate how traditional means for ensuring type-inferability using constraint-based algorithms can make significant demands on the metatheory, which can cause problems for mechanical verification. In Section 5, we provide insights into how we can develop a simpler, more direct proof by viewing constraint-derivation as a direct construction of a proof of safety rather than as a means to make constraints easier to solve. After contributing an informal understanding of the problems and solutions at hand, we move on to our formal contributions. In Section 6, we define a very simple calculus that is still expressive enough to require our innovations to overcome past obstacles. In Section 7, we introduce our novel declarative notion of program validity, *type-consistency*, and prove it ensures safety with*out* ensuring type-inferability (though it accepts all type-inferable programs). In Section 8, we formalize our *type-outference* algorithm, incorporating a mutable configuration state into constraint-derivation in order to scale functionality while still ensuring termination. In Section 9, we prove that our type-outference algorithm (efficiently) decides type-consistency. Altogether, these form a new nominal foundation for decidable validity of programs without expression-level type annotations.

Interface	Method
`java.util.stream.Stream<T>`	`<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>>)`
`System.Thread.Tasks.Task<TResult>`	`Task<TNewResult> ContinueWith<TNewResult>(Func<Task<TResult>,TNewResult>)`
`scala.collection.immutable.Map[K, +V]`	`abstract def updated[V1 :> V](K, V1): Map[K, V1]`
`kotlinx.coroutines.CancellableContinuation<in T>`	`fun resume<R : T>(R, (Throwable, R, CoroutineContext) -> Unit?)`

Fig. 1. Interfaces with generic methods from standard libraries of various major industry languages

```
interface Foo<in X> {
  fun foo1(): Foo<Foo<Foo<X>>>
  fun foo2(x: X): Foo<Foo<Foo<X>>>
}
```

```
fun bar1(i1: Foo<Any>) {
  var v1 = i1
  while (true)
    v1 = v1.foo1()
}
```

```
fun bar2(i2: Foo<Any>) {
  var v2 = i2
  while (true)
    v2 = v2.foo2(v2)
}
```

Fig. 2. Running example of challenging programs to validate

2 Algorithmic Challenges

While many interfaces in practice are quite simple (albeit sometimes quite large), projects tend to have a few critical interfaces with complex method signatures. For example, Figure 1 presents interfaces from standard libraries across major industry languages with a method that is itself *generic*, i.e. parametrically polymorphic. In some cases, the signature is furthermore *recursive*, i.e. referencing the interface declaring the method, and even *irregularly* so, meaning with type arguments differing from the type parameters of the declaring interface. Each of these are problematic because they cause the *structural* encoding of these generic interfaces to use features that are known to be undecidable [DeYoung et al. 2024; Pierce 1992].

While generic methods are beyond the scope that we can cover here, we use irregular recursion as a first proxy for the challenges posed by such critical generic interfaces. As an example, the interface in Figure 2 has been minified to illustrate why even just this feature poses a significant challenge. The interface `Foo` has method signatures that reference `Foo` applied to type arguments that are syntactically larger than its type parameter `X`. (The reason that the return types are not simply `Foo<Foo<X>>` is that such a signature would not be *contravariant* with respect to `X`. Note that `X` is declared as an `in` type parameter, which here means it is contravariant, so that `Foo<`τ`>` is a *sub*type of `Foo<`τ'`>` whenever τ is a *super*type of τ'—note the change from sub- to super-.) Such an interface signature is *expansively* recursive, and as such it cannot be structurally encoded using regular—i.e. parameterless—fixed-point types. Regularity has been assumed by prior works on decidable inference [Binder et al. 2022; Castagna et al. 2016; Dolan 2017; Dolan and Mycroft 2017; Jim and Palsberg 1999; Palsberg and O'Keefe 1995; Parreaux 2020; Parreaux and Chau 2022; Pottier 1998b,b; Sekiguchi and Yonezawa 1994; Tiuryn and Wand 1993],[1] but unfortunately the assumption does not hold in practice for many major typed object-oriented languages. In the following we illustrate why such signatures make validation so challenging.

[1]MLstruct [Parreaux and Chau 2022] does support methods with irregularly-recursive signatures and even type parameters, but each invocation of these methods needs to effectively explicitly specify the receiver's intended interface name so that the signature can be resolved prior to inference, whereas neither we nor industry languages need such annotation.

2.1 Infinite Chains of Supertypes

Consider the two example programs also in Figure 2. Both these programs have a mutable variable, `v1` or `v2`, without a type annotation. Both programs also use an `Any` type, which here all interfaces inherit (but which is *not* a top type here).

One way to attempt to validate these programs is to use the type of the initializing expression as the inferred type of the mutable variable, which for both programs would be `Foo<Any>`. However, due to subtyping, this initial type can be overly precise: subsequent assignments might not be assignable to that initial type, but they could be assignable to some supertype thereof. For example, the assignment in the loop has the type `Foo<Foo<Foo<Any>>>`, which—because `Foo` is contravariant—is a supertype rather than a subtype of `Foo<Any>`. This problem with overprecise inference arises frequently in practice particularly due to `null` being a common initializing expression.

As such, a second way to attempt to validate these programs is to progressively relax the inferred type of a mutable variable as the variable is subsequently assigned expressions with less precise types. This relaxation might require revalidating parts of the program. In our example programs, relaxing the variable's type to `Foo<Foo<Foo<Any>>>` requires revalidating the loop body with this less precise type. This revalidation results in assigning an even less precise type to the variable: `Foo<Foo<Foo<Foo<Any>>>>>`.

Unfortunately, our example programs demonstrate that even this simple type system admits an infinite chain of supertypes: for any $i \in \mathbb{N}$, $\texttt{Foo}^{2i+1}\texttt{<Any>}$ is a subtype of $\texttt{Foo}^{2i+3}\texttt{<Any>}$. This means such iteration continues forever, failing to ever find a fixed point. Thus, work ensuring subtyping is decidable [Greenman et al. 2014; Kennedy and Pierce 2007] is only one step towards the goal.

2.2 Infinite Derivations of Constraints

Rather than iterate over the lines of the program, another approach is to collect subtyping constraints across the entire (encapsulated portion of) the program—introducing "flexible" type variables as placeholders for unknown types—and then reducing the constraints until the unknowns are solvable. In some cases, the solutions are constructed by the algorithm during constraint-reduction or satisfiability-checking [Castagna et al. 2016; Dolan 2017; Dolan and Mycroft 2017; Fuh and Mishra 1988, 1989, 1990; Kaes 1992; Kozen et al. 1994; Mitchell 1984; Palsberg et al. 1997; Sekiguchi and Yonezawa 1994; Smith 1991; Stansifer 1988; Tiuryn and Wand 1993]. In other cases, the constraints are reduced to a point where solutions for the unknowns are guaranteed to exist, but the algorithm never actually constructs them [Aiken and Wimmers 1993; Aiken et al. 1994; Binder et al. 2022; Jim and Palsberg 1999; Palsberg and O'Keefe 1995; Parreaux 2020; Parreaux and Chau 2022; Pottier 1998a,b; Trifonov and Smith 1996]. In our view, these latter cases are primarily type-outference algorithms deciding type-consistency, but the respective type grammars have been extended in order to ensure that type-consistency furthermore implies type-inferability so that these can also be classified as traditional type-inference algorithms.

A key challenge for these algorithms is to ensure that constraint-reduction terminates. For this reason, those that support generic interfaces restrict them to be regularly recursive. This ensures that constraint-reduction can remain within a finite set of types; thus all infinite reductions necessarily cycle and can be short-circuited using memoization.

Unfortunately, our example interface `Foo` requires relaxing this restriction, and our example programs indeed cause these algorithms to fail to terminate. In particular, the method-invocation sites are processed into a constraint that the unknown receiver type must have that method with some signature that is compatible with the unknown input and result types. Initially, the unknown receiver type is lower-bounded by just `Foo<Any>`, which does have the required method. Its result type for that method is `Foo<Foo<Foo<Any>>>`, which becomes a derived lower bound

```
open class Biz1 {
  open fun biz() { baz1() }
  open fun baz1() { ... }
}
```

```
open class Biz2 {
  open fun baz2() { ... }
  open fun biz() { baz2() }
}
```

```
fun buzz(b1: Biz1, b2: Biz2) {
  val v = if (flip()) b1 else b2
  v.biz()
}
```

Fig. 3. Example of a challenging program to invalidate

on the unknown result type of the invocation site. That unknown is a lower bound of the loop variable's unknown type, which itself is a lower bound of the receiver's unknown type; consequently, `Foo<Foo<Foo<Any>>>` is derived as another lower bound for the receiver. This in turn is checked to have a compatible signature, from which yet another constraint on the result's unknown type is derived, which causes constraint-derivation to proceed forever.

Thus, it seems we need a more specialized technique for method invocation in order to ensure termination even in the presence of irregularly-recursive generic interfaces.

2.3 Nominal Method Invocations

Another issue is that many languages need to be able to *reject* programs that many existing approaches would need to accept. To understand why, consider the programs in Figure 3. While both `open` (i.e. inheritable) classes `Biz1` and `Biz2` have a (`open`, i.e. overridable) `biz` method, in many implementations (such as the JVM) the address of their objects' implementations for `biz` might reside at different offsets within the virtual-method table. As such, in order to use offset-based implementations (or straightforwardly compile to the JVM), one often wants to reject the invocation `v.biz()`. That is, one wants method invocations to be *nominal*. This is in contrast to existing approaches, which treat method invocations as *structural*, relying on them being implemented uniformly as dictionary-lookups in order to model them as interface-independent constraints.

Interestingly, we will be able to use the same technique to simultaneously enforce this restriction and prevent the infinite generation of constraints even for irregularly-recursive generic interfaces.

3 An Event-Driven Type-Outference Algorithm

We overcome the aforementioned challenges by incorporating event-driven programming into constraint-based program validation. Here we informally discuss the technique and walk through its application to our (first) running example: `bar1` from Figure 2.

3.1 Label-Listeners

The aforementioned algorithm essentially resolves the method `foo1` for each derived lower bound of the receiver. This fails to work because each time it does so it ends up creating a new lower bound for the receiver. Note, though, that each of these lower bounds is of the form `Foo<...>`. Our key insight is that, *once* we know the receiver is at most a `Foo`—regardless of the type arguments—then we can determine its *generic* signature: `() -> Foo<Foo<Foo<X>>>` for some `X`. While the concrete signature will differ for each lower bound due to differing type arguments, they will each be instantiations of this generic form. However, because there might in general be many interfaces with a `foo1` method, each with a different signature, in order to apply this insight we need some way to react to such abstracted information gained during constraint-derivation.

To this end, we introduce *label-listeners*. These listen to the *label* (e.g. interface *name*—but not type arguments) of each lower bound derived for certain unknown types during constraint-derivation. If the set of labels lower-bounding a label-listener changes, eventually a corresponding action fires, adding unknowns and constraints to the system in order to accommodate the implications of that

change. For example, we can install a label-listener on the receiver of the invocation of `foo1`, whose action—in the case that the receiver's label becomes lower-bounded by `Foo`—is to introduce an unknown X and to require the receiver to be a subtype of `Foo<`X`>` and the result to be a supertype of `Foo<Foo<Foo<`X`>>>`. This effectively resolves the method at most once per lower-bounding *label* rather than at least once per lower-bounding type.

By making a label-listener sensitive to only changes in bounding labels, rather than types, we cap how many times its corresponding action can fire and thereby cause the set of unknowns and constraints to expand. Although not necessary for the features in this paper, these actions can more generally even introduce more label-listeners so long as the total collection of potential label-listeners remains finite, which is easy to guarantee if label-listeners are tied to syntactic elements of the (finite) program—such as method-invocation sites. By capping how many times each label-listener's action can fire, we ensure termination.

3.2 Supporting Nominal Method Invocation

Now consider the effect this strategy for method-resolution has on the `buzz` program from Figure 3. The receiver of `biz` acquires two lower bounds: `Biz1` and `Biz2`. If we were to resolve `biz` for each lower bound, we would accept this program because both lower bounds provide an appropriate `biz` method. However, using a label-listener enables us to resolve `biz` differently for each *set* of lower-bounding labels (so long as the action is more restricting the larger the set is).

This gives us a choice in the case where that set is both `Biz1` and `Biz2`. We could resolve the method for both labels, conceptually making the receiver's type be the union of `Biz1` and `Biz2`. Alternatively, we could simply reject the invocation due to `Biz1` and `Biz2` having no common superinterfaces. The former choice recreates structural method invocation, accepting the `buzz` program. Unfortunately, this is problematic for the compilation strategies discussed in Section 2.3. The latter choice, on the other hand, rejects the `buzz` program. In particular, with the latter choice we essentially are requiring that every label-listener be resolved to use at most one interface's method signature, directly supporting nominal method invocation. We go with this latter choice, though it is worth noting that label-listeners are flexible enough to support the former choice.

3.3 Unreachable Method Invocations

Initially, all label-listeners are *tentatively* configured to have no lower-bounding labels. For method-resolution, this means that initially all invocation sites are *tentatively* resolved to be unreachable, thereby imposing no constraints on the types of the arguments or result. As constraints are processed, lower-bounding labels will be identified, and label-listener actions will fire to constrain these types. However, it is possible that, at the end, no lower-bounding labels are determined for the receiver of a method invocation. In this case, one can compile the site as unreachable; for example, on the JVM one can emit an exception throw to bypass its limited unreachability reasoning. Having this unreachable-resolution option is important because otherwise we would need to arbitrarily pick an interface providing the method and hope the resulting constraints on the argument and result types are unproblematic.

3.4 Walkthrough

Now we walk through our type-outference algorithm as applied to `bar1` of Figure 2. Type-outference first recurses through this program to construct a core graphical representation of the constraints at hand. This representation can be found in Step (1) of Figure 4. A straightforward construction would introduce many more unknowns, but to keep the visualization compact we exploit transitivity to optimize away the uninteresting unknowns. As such, the visualization starts with just one unknown—`V1`—standing in for the unknown type of the mutable variable `v1`.

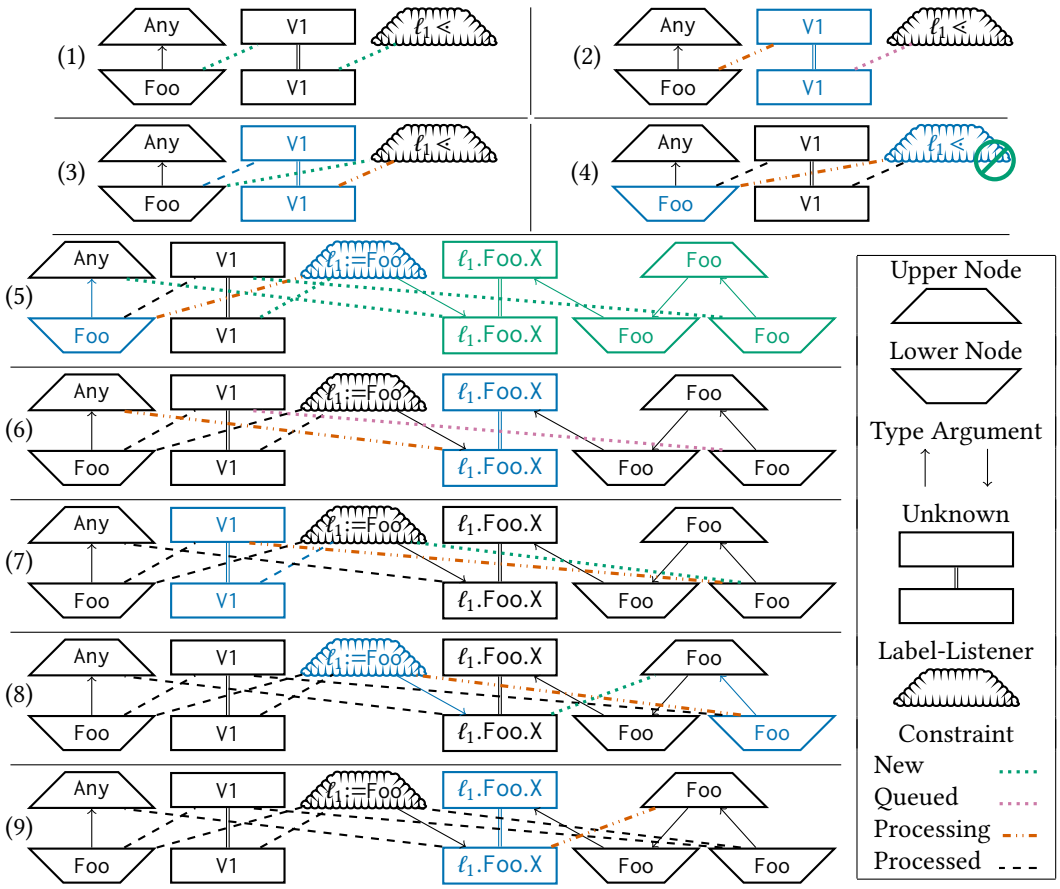


Fig. 4. Example of type-outference with label-listeners (highlight indicates contributors to current step)

Most components of the graphical representation can be sorted into "lower" and "upper" components, so that all constraints indicate that a lower component must be a subtype of an upper component. The visualization displays lower components along the bottom, and upper components along the top, with each subtyping constraint visualized as a dashed edge between a lower component (the subtype) and an upper component (the supertype). On the left of Step (1) is the representation of the type `Foo<Any>` of the input `i1`. Being the type of an input parameter, it imposes only a lower bound, and as such the Foo node is visualized as a lower component. The ↑ edge indicates the node's type argument for the type parameter `X` of the `Foo` interface. Because `X` is contravariant, such edges will always cross ↑ or ↓ between lower and upper components. In this case, it points to an Any node.

In the middle is the representation of the unknown `V1` introduced to represent the unknown type of the program variable `v1`. Such unknowns are special in that they can occur as both lower and upper bounds in constraints, which is why V1 — V1 spans the vertical.

On the right is the representation of the single label-listener ℓ_1 corresponding to the single method-invocation site in `bar1`, which the algorithm initially *tentatively* resolves as unreachable, as visualized by ℓ_1<. In order to accommodate the changing configurations of label-listeners

without requiring backtracking, they are restricted to be upper components. As this example will illustrate, when a derived constraint determines that a tentative resolution would fail due to a newly derived lower-bounding label on the label-listener, it updates the resolution to a new tentative solution based on the newly understood constraints (if any potential solutions remain). (Of course, one could maintain some information during construction to avoid many of these re-resolutions.)

Once the upper and lower components of the graphical representation are constructed, type-outference proceeds to reasoning about the constraints through a seeding and saturation process:

(1) The seeding constraints are those capturing the guaranteed flows in the expression. In this case, there are two such flows: from the input `i1` to the initializing assignment of `v1`, and from the use of `v1` to the label-listener for the invocation site of `foo1`. (Because the invocation is tentatively resolved as unreachable, the flow from its result to `v1` is not yet guaranteed.) In Figure 4, we use ····· for constraints being added in the current step.
(2) Nearly every step is the processing of a constraint. In the previous step, we added two constraints, only one of which is processed at a time. We use ····· for constraints in the queue, -·-·-· for the constraint that is currently being processed, and ----- for constraints that have been processed. When we process a constraint, we completely ignore any queued constraints. Because of this, the constraint from `Foo<Any>` to `V1` has no immediate consequences. It adds a lower bound to `V1`, but when we examine `V1` for upper bounds (we highlight any additional components that are examined while processing the current constraint), we ignore the queued upper bound from `V1` to ℓ_1.
(3) Next we process that constraint from `V1` to ℓ_1. This adds an upper bound to `V1`. Because `V1` has a *processed* lower bound, we transitively compose the two constraints, resulting in a new constraint from `Foo<Any>` to ℓ_1. This marks a key characteristic of outference algorithms: we do *not* use these constraints to determine what `V1` must be; we only explore what their transitive implications are, conceptually pulling the unknown `V1` *out* of the constraints.
(4) Next we process the constraint from `Foo<Any>` to ℓ_1. This adds a new lower-bounding label—`Foo`—to the label-listener ℓ_1. This new label is incompatible with ℓ_1's current tentative resolution. As such, ℓ_1's action fires. According to the new collection of lower-bounding labels, the new best resolution candidate for the invocation site is the `Foo` interface.
(5) The new resolution uses the signature of `foo1` declared by `Foo`. The `X` type argument of the receiver is unknown, so we introduce an unknown—ℓ_1`.Foo.X`—to represent it. ℓ_1's tentative resolution is updated to be `Foo<`ℓ_1`.Foo.X>`. Because this was prompted by the new lower bound `Foo<Any>` on ℓ_1, a constraint is added to make ℓ_1`.Foo.X` a subtype (due to contravariance of `X`) of `Any`. Lastly, the label-listener's action adds nodes to represent the type `Foo<Foo<Foo<`ℓ_1`.Foo.X>>>` in `Foo.foo1`'s substituted signature, as well as adds a constraint to make it a subtype of `V1` due to the now-guaranteed flow from the invocation's result to `v1`.
(6) We first process the newly added constraint between the type arguments of the constraint we just processed, which has no consequences because ℓ_1`.Foo.X` has no lower bounds.
(7) We next process the constraint that arose from the new result type and flow. In this case, we compose it using transitivity via `V1` to add a constraint from `Foo<Foo<Foo<`ℓ_1`.Foo.X>>>` to ℓ_1.
(8) We process this newly added constraint between a `Foo` node and a tentatively `Foo` label-listener, adding a new constraint between their type arguments.
(9) We process that newly added constraint from ℓ_1`.Foo.X` to `Foo<Foo<`ℓ_1`.Foo.X>>`, which has no consequences because ℓ_1`.Foo.X` still has no lower bounds.

At this point, all constraints have been processed without error. Because saturation determined that the constraints were *consistent*, our algorithm accepts the program, with ℓ_1's invocation site resolved as `Foo.foo1`.

3.5 Guaranteeing Termination

Note that the only time nodes were added was when a label-listener's action fired to change a tentative method-resolution. Because there are only a finite number of method-invocation sites in the program, there are only a finite number of label-listeners in the graphical representation, and each firing necessarily moves the nominal resolution further up the well-founded inheritance hierarchy. As such, saturation necessarily terminates. In particular, with there being only a finite number of components (after the relevant finite number of label-listener actions have fired), there can only be a finite number of constraints to process. By using this finite graphical representation, when we derive new constraints, we can track which of them have already been processed, and we can be guaranteed that eventually all of them will have been processed, even in the presence of irregularly-recursive generic interfaces.

4 Proof Challenges

After developing such an algorithm, traditionally one proves that it decides type-inferability, sometimes extending the grammar of types in order to express the solutions to consistent constraint sets. But this approach requires complex constructions that seem to us to obfuscate rather than elucidate the reasons why our algorithm works correctly. We even discovered that these complexities create concrete obstacles when one attempts to mechanically verify this approach.

4.1 Extending the Grammar

At the end of our walkthrough, we concluded with the following constraints on our unknowns (which imply the remaining constraints):

$$\texttt{Foo<Any>} <: \texttt{V1} \qquad \texttt{V1} <: \texttt{Foo<}\ell_1\texttt{.Foo.X>} \qquad \texttt{Foo<Foo<Foo<}\ell_1\texttt{.Foo.X>>>} <: \texttt{V1}$$

$$\ell_1\texttt{.Foo.X} <: \texttt{Any} \qquad \ell_1\texttt{.Foo.X} <: \texttt{Foo<Foo<}\ell_1\texttt{.Foo.X>>}$$

These constraints can be solved with simple types using the assignments $\texttt{V1} := \texttt{Foo<Foo<Any>>}$ and $\ell_1\texttt{.Foo.X} := \texttt{Foo<Any>}$. However, if we had applied our algorithm to `bar2` instead, then we would have one more constraint to account for (left below) and its derived implication (right below):

$$\texttt{V1} <: \ell_1\texttt{.Foo.X} \qquad \texttt{Foo<}\ell_1\texttt{.Foo.X>} <: \ell_1\texttt{.Foo.X}$$

In this case, there is no solution expressible with simple types; the solutions for both `V1` and ℓ_1`.Foo.X` must necessarily be of the form "`Foo` applied to itself". Such a type requires extending the grammar, say with regular (i.e. unparameterized) equirecursive fixed points: $\mu t.\texttt{Foo<}t\texttt{>}$.

Should we choose to accept `bar1` but reject `bar2`, we would encounter some problems. First, distinguishing the two circumstances requires another step in the algorithm. Even if this additional step were decidable, performing it takes time. Second, many languages also support overloading, so if we were to eventually add regular equirecursive types to one, then overloadings that were rejected due to this restriction would become valid, thereby potentially changing the run-time behavior of existing programs. In general, this latter point suggests such languages would benefit from a specification of program validity that is *not* so sensitive to the current grammar of types.

Thus, we opt to accept both programs. This, in turn, requires a proof of type-inferability to use an extended grammar. In this example, solely regular equirecursive types are sufficient, but even basic features like inheritance seem to further necessitate unions because an unknown type can be lower-bounded by interfaces that have multiple common superinterfaces but are not themselves directly related. And as one looks ahead to advanced features—flow-sensitive refinements of variable types or higher-kinded polymorphism—it seems even arbitrary equirecursive types might eventually be necessary. Such a feature creep in the complexity of the extended grammar, even if the types that

```
Parameter Interface: Type.
Parameter Arity: Interface → Type.
Inductive UnionType {InterfaceType: Type} {Var: Type}: Type
 := utvariable (variable: Var)
  | utbottom
  | utunion (ltype rtype: UnionType)
  | utinterface (type: InterfaceType).
Arguments UnionType : clear implicits.
#[projections(primitive)]
CoInductive InterfaceType {Var: Type}: Type
 := { itinterface                                : Interface;
      itargument (parameter: Arity itinterface) : UnionType InterfaceType Var }.
Arguments InterfaceType : clear implicits.
Definition ExtendedType (Var: Type): Type
 := UnionType (InterfaceType Var) Var. (* Rocq accepts this definition. *)

Definition utsubst {InterfaceType InterfaceType': Type} {Var Var': Type}
                   (itsubst: InterfaceType → InterfaceType')
                   (f: Var → UnionType InterfaceType' Var')
                 : UnionType InterfaceType Var → UnionType InterfaceType' Var'
 := fix utsubst (type: UnionType InterfaceType Var): UnionType InterfaceType' Var'
     := match type with
        | utvariable variable ⇒ f variable
        | utbottom            ⇒ utbottom
        | utunion ltype rtype ⇒ utunion (utsubst ltype) (utsubst rtype)
        | utinterface type    ⇒ utinterface (itsubst type)
        end.
Definition itsubst {Var Var': Type} (f: Var → ExtendedType Var')
                 : InterfaceType Var → InterfaceType Var'
 := cofix itsubst (itype: InterfaceType Var): InterfaceType Var'
     := {| itinterface          := itype.(itinterface);
           itargument parameter := utsubst itsubst f (itype.(itargument) parameter) |}.
Definition etsubst {Var Var': Type} (f: Var → ExtendedType Var')
                 : ExtendedType Var → ExtendedType Var'
 := utsubst (itsubst f) f. (* Rocq rejects this definition—specifically that of itsubst. *)
```

Fig. 5. Encoding of the extended type grammar (`ExtendedType`) and substitution thereon (`etsubst`) in Rocq—the latter is unnecessarily rejected due to lack of support for mixing corecursion and recursion [Pédrot 2015]

would actually be written by users could remain non-recursive, raises concerns for the long-term practicality of this approach.

4.2 Metatheoretic Limitations in Mechanized Proofs

We discovered that extending the grammar was particularly problematic for mechanical verification. In the extended grammar, we want to be able to apply `Foo` to itself recursively, but we do not want to be able to apply a union to itself recursively. That is, all equirecursion should be guarded by an interface application. But in between such recursions one needs to be able to have an arbitrary (finite) number of union applications.

This is most naturally represented using mutually-defined coinductive and inductive types. Rocq does not directly support this, but it can encode such mutual recursion, as given by the definition of `ExtendedType` in Figure 5. However, Rocq fails to support the corresponding mutual recursion required for defining substitution over this grammar, given by `etsubst` in Figure 5. In particular,

the `cofix` fails to recognize that the recursive call to `itsubst` in `utsubst` is guarded; this failure is caused by the `fix` enclosing the call [Pédrot 2015], which Rocq lacks relevant rules for.

Even if Rocq adds support for this feature, its necessity indicates that this proof technique is heavy-handed for the goal at hand. Furthermore, we found these constructions uninsightful. We came to wonder if there were a more direct way to prove safety, one that would be simpler (or at least less demanding of the metatheory) and would better capture why the algorithm is effective.

5 Constraint-Consistency as a Proof of Safety

Rather than viewing constraint-derivation as a means to make constraints easier to solve, we view constraint-derivation as directly constructing a proof that the program is safe. For example, deriving constraints from transitivity through unknowns is conceptually simply connecting in-flows to out-flows. Checking that constraints between interface types have compatible interfaces is simply checking that produced values are being consumed in compatible ways. Deriving constraints between their respective type arguments is then capturing the indirect flows stemming from that potential interaction. Many have observed the close relationship between flow-analysis and type-inference and that the former can often safely admit more programs than the latter [Palsberg and Schwartzbach 1992, 1995]. But rather than extending the type grammar to make type-inference equivalent to flow-analysis [Palsberg and O'Keefe 1995], we prove safety directly from this connection to flow.

To that end, we consider the type variables introduced during constraint-collection *not* as unknown types but rather as *abstract* types, with the collected constraints describing the key semantic properties of these abstract types. In other words, constraint-collection is essentially constructing a customized *type space* in which that particular program type-checks. For example, `bar2` in Figure 2 would type-check in a grammar of types extended with type-constants `V1` and ℓ_1`.Foo.X` so long as subtyping were also extended with rules for those constants corresponding to the collected constraints. Of course, extending subtyping rules arbitrarily can lead to unsafety. Constraint-derivation, then, is ensuring that this custom type space is *consistent* with the operational expectations of the execution semantics and therefore *safe* to use.

From this perspective, the extended grammar with fixed points and unions is merely attempting to establish a *uniform* type space that all such consistent type spaces embed into. Thus, the extended grammar's metatheoretic complexity stems from its need to be all-encompassing. By formalizing a means for each program to use its own type space, we will be able to directly use the consistency of a program's derived constraints as its proof of safety and bypass this complex extended grammar. In our experience, making this connection more direct has provided insights into how we can safely extend a language with new features.

6 A Minimal Calculus

Now we finally discuss our contributions formally. To that end, we introduce a calculus, one made extremely small in order to focus on the key concepts behind type-consistency and type-outference.

6.1 Grammar

The grammar of our calculus is presented in Figure 6. It consists of three sections.

The first section specifies the grammar of *user* types τ, which are the types denotable by the user. A user type is either a type variable α, the type Any, or an interface I applied to type arguments. The notation $\overline{\tau}$ indicates the grammar for a list of types. Note that there are no top, bottom, union, intersection, or fixed-point types, and at no point will we define an extended grammar adding such types. In particular, the role of Any is *not* to provide a top type or to make constraint-solving easier;

Type Variable	α	
Interface	I	
User Type	τ	$::= \alpha \mid \mathsf{Any} \mid I\langle\overline{\tau}\rangle$
Variance	$\pm$	$::= + \mid -$
Kind Context	Θ	$::= \langle\pm\alpha\rangle_\alpha$
Method	m	
User Method Signature	$\mathcal{M}$	$::= \tau \to \tau$
User Body Signature	$\mathcal{B}$	$::= \{m : \mathcal{M}\}_m$
Hierarchy	$\mathcal{H}$	$::= (I\langle\Theta\rangle : \mathcal{B})_I$
Variable	x	
Expression	e	$::= x \mid e.m(e) \mid I\{b\}$
Body	b	$::= \{m(x) := e\}_m$
Program	$\mathcal{P}$	$::= \mathcal{H} \Vdash e : \tau$

Fig. 6. Grammar—$\overline{\cdot}$ denotes lists, and $\langle\cdot\rangle_\alpha$, $\{\cdot\}_m$ and $(\cdot)_I$ denote ordered finite maps with respective keys

$$\text{Value} \quad v ::= I\{b\}$$

$$\boxed{e \to e} \qquad \frac{e \to e'}{e.m(e'') \to e'.m(e'')} \qquad \frac{m \in b \qquad \text{(where } [x \mapsto e] \text{ denotes capture-avoiding substitution)}}{I\{b\}.m(e) \to e^b_m[x^b_m \mapsto e]}$$

Fig. 7. Operational semantics

it simply exists to enable non-trivial subtyping relationships between our user types and to support our running example.

The second section specifies the grammar of a hierarchy $\mathcal{H}$. Each interface I in a hierarchy $\mathcal{H} = (I\langle\Theta^{\mathcal{H}}_I\rangle : \mathcal{B}^{\mathcal{H}}_I)_{I\in\mathcal{H}}$ has a corresponding kind context $\Theta^{\mathcal{H}}_I$. This kind context specifies the type parameters of the interface and their corresponding variance (+ for covariant or − for contravariant); the notation $\langle\pm\alpha\rangle_\alpha$ denotes a list (so order matters) of $\pm\alpha$ elements wherein any α occurs in this list at most once (thereby also encoding a finite map). Lastly, each interface has a user-expressible body signature $\mathcal{B}^{\mathcal{H}}_I$. Such a body signature specifies the methods m declared by that interface, along with a user-expressible method signature $\mathcal{M}$ (which is simply an input type and a result type) for each such method. More generally, when grammatical elements (such as $\mathcal{H}$, Θ, and $\mathcal{B}$) have an obvious interpretation as records, we will use it as a superscript to denote the obvious projection, sometimes also with a subscript when that projection is indexed (such as in $\Theta^{\mathcal{H}}_I$, $\pm^{\Theta}_\alpha$, and $\mathcal{M}^{\mathcal{B}}_m$).

The third section specifies the grammar of programs $\mathcal{P}$. A program $\mathcal{P} = \mathcal{H}^{\mathcal{P}} \Vdash e^{\mathcal{P}} : \tau^{\mathcal{P}}$ is a (closed) expression along with an expected (closed) type within a specified hierarchy. An expression e is either a (program) variable, a method invocation, or an object instantiation. An object instantiation specifies the interface I being instantiated and a body b comprised of the method implementations of the object. Note that there are no types within an object instantiation; both the input type and the result type of all method implementations are missing. Note also that no type arguments are given for the interface. Thus, only the interface *name* is given, and everything else needs to be outferred.

$$\boxed{\mathcal{H} \mid \Theta \vdash \tau : \pm} \qquad \frac{\alpha \in \Theta}{\mathcal{H} \mid \Theta \vdash \alpha : \pm^{\Theta}_{\alpha}} \qquad \frac{}{\mathcal{H} \mid \Theta \vdash \mathsf{Any}} \qquad \frac{\text{for all } \pm'\alpha \in \Theta^{\mathcal{H}}_{I}, \quad \mathcal{H} \mid \Theta \vdash \tau_{\alpha} : \pm \cdot \pm'}{\mathcal{H} \mid \Theta \vdash I\langle \tau_{\alpha} \rangle_{\alpha \in \Theta^{\mathcal{H}}_{I}} : \pm}$$

Fig. 8. Type validity, where $\pm \cdot \pm$ is notation for $+ \cdot + = +$ and $+ \cdot - = -$ and $- \cdot + = -$ and $- \cdot - = +$

$$\boxed{\vdash \mathcal{H}} \qquad \frac{\text{for all } I \in \mathcal{H}, \quad \mathcal{H} \mid \Theta^{\mathcal{H}}_{I} \vdash \mathcal{B}^{\mathcal{H}}_{I}}{\vdash \mathcal{H}} \qquad \boxed{\mathcal{H} \mid \Theta \vdash \mathcal{B}} \qquad \frac{\text{for all } m \in \mathcal{B}, \quad \mathcal{H} \mid \Theta \vdash \mathcal{M}^{\mathcal{B}}_{m}}{\mathcal{H} \mid \Theta \vdash \mathcal{B}} \qquad \boxed{\mathcal{H} \mid \Theta \vdash \mathcal{M}} \qquad \frac{\mathcal{H} \mid \Theta \vdash \tau : - \quad \mathcal{H} \mid \Theta \vdash \tau' : +}{\mathcal{H} \mid \Theta \vdash \tau \to \tau'}$$

Fig. 9. Hierarchy validity

6.2 Operational Semantics

The operational semantics of our calculus is provided in Figure 7. A value v is simply an arbitrary object instantiation. Method invocation simply looks up the corresponding method implementation (if it exists) in the object body and substitutes its input with the argument. (We use a lazy semantics simply because it is simpler to mechanically formalize.)

6.3 Type Validity

Figure 8 specifies when a type has the appropriate variance with respect to a kind context Θ (and its type variables are within scope). For example, the rule for type variables specifies that a type variable α has the variance $\pm^{\Theta}_{\alpha}$ assigned to it by the kind context Θ (supposing there is any such assignment). Note that the rule for interface types uses variance multiplication, which is defined in the caption of Figure 8.

Lemma 6.1. *Given a hierarchy $\mathcal{H}$, a kind context Θ, a type τ, and a variance $\pm$, $\mathcal{H} \mid \Theta \vdash \tau : \pm$ is decidable.*

Proof. The decision procedure is a standard syntactic process. □

6.4 Hierarchy Validity

The hierarchy of the program must respect standard scope and variance, as formalized in Figure 9. In particular, each interface's body signature in the hierarchy must respect that interface's kind context (because a hierarchy $\mathcal{H}$ is a finite map, the notation $I \in \mathcal{H}$ indicates I is in the domain of this map). This holds if each method signature in the body signature respects that kind context. Specifically, each input type must be contravariant, and each result type must be covariant.

Lemma 6.2. *Given a hierarchy $\mathcal{H}$, $\vdash \mathcal{H}$ is decidable.*

Proof. The decision procedure is a standard syntactic process. □

7 Type-Consistency

The last step of formalizing our calculus is program validity. To this end, we introduce a novel notion of program validity: *type-consistency*. Rather than using a fixed grammar of types for all programs, type-consistency permits each program to be validated using its own space of types. However, in order to ensure safety, such a *type space* must satisfy an additional *consistency* property.

$$\boxed{\mathcal{H} \vdash \tau <: \tau}$$

$$\frac{}{\mathcal{H} \vdash \alpha <: \alpha} \qquad \frac{}{\mathcal{H} \vdash \text{Any} <: \text{Any}} \qquad \frac{\text{for all } \pm\alpha \in \Theta_I^{\mathcal{H}}, \quad \mathcal{H} \vdash \tau_\alpha <:_\pm \tau'_\alpha}{\mathcal{H} \vdash I\langle\tau_\alpha\rangle_{\alpha\in\Theta_I^{\mathcal{H}}} <: I\langle\tau'_\alpha\rangle_{\alpha\in\Theta_I^{\mathcal{H}}}} \qquad \frac{}{\mathcal{H} \vdash I\langle\overline{\tau}\rangle <: \text{Any}}$$

Fig. 10. User subtyping, where $R_\pm$ is notation for $a\ R_+\ b = a\ R\ b$ and $a\ R_-\ b = b\ R\ a$

7.1 Type Space

We need a way to reason about types without restricting ourselves to a specific grammar of types. To this end, we introduce *type spaces* (using the notation $R_\pm$ defined in the caption of Figure 10).

Definition 7.1. A type space $\mathcal{T}$ in a hierarchy $\mathcal{H}$ consists of

- a set of "types" (whose elements are referenced by $T \in \mathcal{T}$),
- a type $\text{Any}^{\mathcal{T}} \in \mathcal{T}$,
- for each interface $I \in \mathcal{H}$ and $\Theta_I^{\mathcal{H}}$-sized tuple of types $\overline{T}$, a type $I\langle\overline{T}\rangle^{\mathcal{T}} \in \mathcal{T}$,
- a preorder $<:^{\mathcal{T}}$ on types satisfying, for all interfaces $I \in \mathcal{H}$ and type arguments $\langle T_\alpha\rangle_{\alpha\in\Theta_I^{\mathcal{H}}}$,

$$I\langle T_\alpha\rangle^{\mathcal{T}}_{\alpha\in\Theta_I^{\mathcal{H}}} <:^{\mathcal{T}} \text{Any}^{\mathcal{T}} \text{ and } \forall\langle T'_\alpha\rangle_{\alpha\in\Theta_I^{\mathcal{H}}}.\ \left(\forall\pm\alpha \in \Theta_I^{\mathcal{H}}.\ T_\alpha <:^{\mathcal{T}}_\pm T'_\alpha\right) \implies I\langle T_\alpha\rangle^{\mathcal{T}}_{\alpha\in\Theta_I^{\mathcal{H}}} <:^{\mathcal{T}} I\langle T'_\alpha\rangle^{\mathcal{T}}_{\alpha\in\Theta_I^{\mathcal{H}}}$$

- and a predicate $\lessdot^{\mathcal{T}}$ on types satisfying

$$\forall T, T' \in \mathcal{T}.\quad T <:^{\mathcal{T}} T' \wedge T' \lessdot^{\mathcal{T}} \implies T \lessdot^{\mathcal{T}}$$

In other words, a type space is an *algebra*, both on its set of types and on its preorder $<:^{\mathcal{T}}$ and predicate $\lessdot^{\mathcal{T}}$. This means we can know there are at least the types and relationships between them that one would expect, but that a type space is not limited to just the constructs we have conceived of so far. User types are simply the *free* type space—given any user type $\mathcal{H} \mid \Theta \vdash \tau$ and assignment of its free type variables $\Theta \mapsto \overline{T}$ to types in a type space $\mathcal{T}$, we can construct an interpretation $\tau[\Theta \mapsto \overline{T}]^{\mathcal{T}}$ of that user type within $\mathcal{T}$ in the obvious manner; furthermore, this mapping maps user-subtypes, defined in Figure 10, to $\mathcal{T}$-subtypes due to the algebraic properties required of $<:^{\mathcal{T}}$. Interestingly, user subtyping is not directly used by our novel definition of program validity; however, because user types and user subtyping form a type space (the user type space), user subtyping provides a useful baseline for how users can expect program components to compose.

The predicate $\lessdot^{\mathcal{T}}$ of a type space will be used to indicate that a type provides any method with any method signature. It is used to handle the situation where the receiver of a method is unconstrained. A bottom type does not necessarily satisfy this property in a nominal setting, nor does a type satisfying this property necessarily need to be a bottom type. Nonetheless, many systems do have a bottom type satisfying this property.

7.2 Type-Checking with a Type Space

Given a type space, we can define when an expression type-checks within that space. Figure 11 provides the four rules for type-checking. The first rule states that subtyping is subsumptive. The second rule is simply variable lookup. (Note that type contexts Γ are defined with respect to arbitrary types in $\mathcal{T}$, not just user types). The third and fourth rules respectively use judgements for method invocation and for object instantiation, which we discuss in more detail.

The judgement for method invocation $\mathcal{H} \mid \mathcal{T} \vdash T.m : M$ states that T's signature for m is M (again, using arbitrary types in $\mathcal{T}$, not just user types). It has two cases. The first case incorporates the key typing behavior of the $\lessdot^{\mathcal{T}}$ predicate; the expectation is that this will only be applicable when the receiver is unconstrained, implying the invocation site is unreachable. The second case is

$$\begin{array}{lll} \text{Type Contexts} & \Gamma & ::= (x : T)_x \\ \text{Method Signature} & M & ::= T \to T \\ \text{Body Signature} & B & ::= \{m : M\}_m \end{array}$$

$\boxed{\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e : T}$

$$\frac{\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e : T \qquad T <:^{\mathcal{T}} T'}{\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e : T'} \qquad \frac{}{\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash x : T_x^{\Gamma}}$$

$$\frac{\begin{array}{c}\mathcal{H} \mid \mathcal{T} \vdash T.m : T' \to T'' \\ \mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e : T \\ \mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e' : T'\end{array}}{\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e.m(e') : T''} \qquad \frac{\begin{array}{c}\mathcal{H} \mid \mathcal{T} \vdash \{m : T'_m \to T''_m\}_{m \in b} \sqsubseteq I\langle \overline{T} \rangle \\ \text{for all } m \in b, \quad x_m^b \notin \Gamma \\ \text{for all } m \in b, \quad \mathcal{H} \mid \mathcal{T} \mid \Gamma, x_m^b : T'_m \vdash e_m^b : T''_m\end{array}}{\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash I\{b\} : I\langle \overline{T} \rangle^{\mathcal{T}}}$$

$\boxed{\mathcal{H} \mid \mathcal{T} \vdash T.m : M}$

$$\frac{T \lessdot^{\mathcal{T}}}{\mathcal{H} \mid \mathcal{T} \vdash T.m : M} \qquad \frac{}{\mathcal{H} \mid \mathcal{T} \vdash I\langle \overline{T} \rangle^{\mathcal{T}}.m : \mathcal{M}_m^{\mathcal{B}_I^{\mathcal{H}}}[\Theta_I^{\mathcal{H}} \mapsto \overline{T}]^{\mathcal{T}}}$$

$\boxed{\mathcal{H} \mid \mathcal{T} \vdash B \sqsubseteq I\langle \overline{T} \rangle}$

$$\frac{\text{for all } m : \mathcal{M} \in \mathcal{B}_I^{\mathcal{H}}, \quad \mathcal{H} \mid \mathcal{T} \vdash M_m^B <: \mathcal{M}[\Theta_I^{\mathcal{H}} \mapsto \overline{T}]^{\mathcal{T}}}{\mathcal{H} \mid \mathcal{T} \vdash B \sqsubseteq I\langle \overline{T} \rangle}$$

$\boxed{\mathcal{H} \mid \mathcal{T} \vdash M <: M}$

$$\frac{T'' <:^{\mathcal{T}} T \qquad T' <:^{\mathcal{T}} T'''}{\mathcal{H} \mid \mathcal{T} \vdash T \to T' <: T'' \to T'''}$$

Fig. 11. Type-checking with respect to a type space $\mathcal{T}$ in a hierarchy $\mathcal{H}$

standard: an interface provides any method it declares with the explicit signature appropriately substituted (unrolling the notation $\mathcal{M}_m^{\mathcal{B}_I^{\mathcal{H}}}$, one gets the user method signature for method m in the user body signature for I given by the hierarchy $\mathcal{H}$).

The judgement used for object instantiation $\mathcal{H} \mid \mathcal{T} \vdash B \sqsubseteq I\langle \overline{T} \rangle$ states that the collection of method signatures in B satisfies the requirements of $I\langle \overline{T} \rangle$. It is also straightforward, simply requiring that each method declared by the interface has a corresponding signature in the body signature that refines the substitution of the declared signature.

7.3 Consistent Type Spaces and Safety

On its own, type-checking with respect to a type space is not sufficient to ensure safety. For example, the expression `Foo{}.bar(Unit{})` type-checks in any type space stating that an empty interface `Foo` is a subtype of some other interface that has a method `bar : Unit -> Unit`. The issue is that, while a type space ensures the expected subtypings are present, it does not exclude any unexpected subtypings.

To address this issue, we introduce the notion of a *consistent* type space $\mathcal{H} \vdash \mathcal{T}$, defined in Figure 12. The definition first states that, because this calculus does not support inheritance, interface types can be subtypes *only* when the interfaces are the same and the type arguments are subtypes or supertypes according to the variance of the interface. Thus, in a consistent type space, the empty interface `Foo` cannot be a subtype of an interface with a non-empty signature, causing consistent type spaces to reject the expression `Foo{}.bar(Unit{})` as one would expect. The definition secondly states that Any cannot satisfy $\lessdot^{\mathcal{T}}$. Lastly, the definition states that no interface type can either be a supertype of Any or satisfy $\lessdot^{\mathcal{T}}$.

In other words, a consistent type space also has a *coalgebraic* structure on its subtyping and predicate, but *without* requiring any coalgebraic structure on its set of types. The latter permits a

$$\boxed{\mathcal{H} \vdash \mathcal{T}} \qquad \frac{\left(\begin{array}{c}\text{for all } I \in \mathcal{H},\, I' \in \mathcal{H},\, \langle T_\alpha \rangle_{\alpha \in \Theta^{\mathcal{H}}_I}, \text{ and } \langle T'_\alpha \rangle_{\alpha \in \Theta^{\mathcal{H}}_{I'}}, \\ I\langle T_\alpha \rangle^{\mathcal{T}}_{\alpha \in \Theta^{\mathcal{H}}_I} <:^{\mathcal{T}} I'\langle T'_\alpha \rangle^{\mathcal{T}}_{\alpha \in \Theta^{\mathcal{H}}_{I'}} \text{ implies } I = I' \text{ and for all } \pm\alpha \in \Theta^{\mathcal{H}}_{I'},\, T_\alpha <:^{\mathcal{T}}_{\pm} T'_\alpha\end{array}\right) \quad \mathsf{Any}^{\mathcal{T}} \lessdot^{\mathcal{T}} \text{ does not hold} \quad \text{for all } I \in \mathcal{H}, \quad \text{no } \overline{T} \text{ satisfies either } \mathsf{Any}^{\mathcal{T}} <:^{\mathcal{T}} I\langle \overline{T} \rangle^{\mathcal{T}} \text{ or } I\langle \overline{T} \rangle^{\mathcal{T}} \lessdot^{\mathcal{T}}}{\mathcal{H} \vdash \mathcal{T}}$$

Fig. 12. Type-space consistency

type space to arbitrarily extend the grammar of types, while the former ensures the extension still respects the relationships expected between user-expressible constructs. The user type space is consistent, and within many proofs of type safety for existing works lie proofs that user subtyping has consistent coalgebraic structure.

Theorem 7.2. *The space of closed user types with user subtyping and with false as $\lessdot$ is consistent.*

Yet, by combining the algebraic and coalgebraic structure of consistent type spaces, we are able to prove progress and preservation for expressions checked using *any* consistent type space.

Theorem 7.3. *Given a hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, a type space $\mathcal{T}$ satisfying $\mathcal{H} \vdash \mathcal{T}$, and an expression e and type $T \in \mathcal{T}$ satisfying $\mathcal{H} \mid \mathcal{T} \mid \varnothing \vdash e : T$, the following properties hold:*

Progress *Either e is a value v or there exists an e' satisfying $e \longrightarrow e'$.*

Preservation *Any expression e' satisfying $e \longrightarrow e'$ also satisfies $\mathcal{H} \mid \mathcal{T} \mid \varnothing \vdash e' : T$.*

Proof. In addition to straightforward lemmas on substitution and the like following from the *algebraic* structure of arbitrary type spaces, progress and preservation rely on the following facts:

(1) No value can have a type satisfying $\lessdot^{\mathcal{T}}$.
(2) If a value $I\{b\}$ has type $I'\langle \overline{T} \rangle^{\mathcal{T}}$, then there is a body signature $\{m : T'_m \longrightarrow T''_m\}_{m \in b}$ satisfying both $\forall m \in b.\ \mathcal{H} \mid \mathcal{T} \mid x^b_m : T'_m \vdash e^b_m : T''_m$ and $\mathcal{H} \mid \mathcal{T} \vdash \{m : T'_m \longrightarrow T''_m\}_{m \in b} \sqsubseteq I'\langle \overline{T} \rangle^{\mathcal{T}}$.

Both of these facts follow from the *coalgebraic* structure of consistent type spaces. Consider the latter fact as an example. The proof that the object type-checks must be an application of object instantiation (using type arguments $\overline{T'''}$ and body signature $\{m : T'_m \longrightarrow T''_m\}_{m \in b}$) followed by a series of subsumptions. By transitivity of subtyping in type spaces, this means $I\langle \overline{T'''} \rangle^{\mathcal{T}}$ must be a subtype of $I'\langle \overline{T} \rangle^{\mathcal{T}}$. Thus, by the coalgebraic structure ensured by the assumed consistency of $\mathcal{T}$, we know that I' must be I and that $\overline{T}$ must be supertypes (modulo variance) of the type arguments $\overline{T'''}$ used to validate the instantiation. Putting this together with basic properties ensured by requiring body signatures to respect variance, one can easily derive $\mathcal{H} \mid \mathcal{T} \vdash \{m : T'_m \longrightarrow T''_m\}_{m \in b} \sqsubseteq I'\langle \overline{T} \rangle^{\mathcal{T}}$.

Using the above two facts, the proofs of progress and preservation are made trivial. □

Shortly we will connect consistency of type spaces to consistency of constraint sets. Eifrig et al. [1995a] also observed that constraint-consistency ensures safety, but their exploration of the topic is incomplete. For example, their definition of program validity includes a many-part *Simplification* algorithm, and the abbreviated proof of safety provided in the extended paper [Eifrig et al. 1995b] never mentions how consistency of the constraints contributes to the safety of reduction. Our observation is that safety follows from constraint-consistency ensuring that a corresponding type space has a coalgebraic structure that aligns with the operational semantics of the language. For example, the coalgebraic structure for interface types guarantees that whenever a well-typed production of an interface type (i.e. an object) flows into a well-typed consumption of an interface

$$\boxed{\mathcal{H} \mid \mathcal{T} \vdash e : \tau} \qquad \boxed{\mathcal{H} \vdash e : \tau} \qquad \boxed{\vdash \mathcal{P}}$$

$$\frac{\mathcal{H} \mid \mathcal{T} \mid \varnothing \vdash e : \tau[\varnothing \mapsto \varnothing]^{\mathcal{T}}}{\mathcal{H} \mid \mathcal{T} \vdash e : \tau} \qquad \frac{\mathcal{H} \vdash \mathcal{T} \qquad \mathcal{H} \mid \mathcal{T} \vdash e : \tau}{\mathcal{H} \vdash e : \tau} \qquad \frac{\vdash \mathcal{H} \qquad \mathcal{H} \mid \varnothing \vdash \tau : + \qquad \mathcal{H} \vdash e : \tau}{\vdash \mathcal{H} \Vdash e : \tau}$$

Fig. 13. Type-consistency

type (i.e. a method invocation), then the resulting reduction is necessarily type-preserving. Thus this work can be seen as a revival and elaboration of Eifrig et al.'s approach to design and proofs.

7.4 Type-Consistency and Program Validity

By Theorem 7.3, our novel declarative definition of type-consistency, given in Figure 13, accepts only safe expressions and programs. The judgement $\mathcal{H} \mid \mathcal{T} \vdash e : \tau$ states that the (arbitrary) type space $\mathcal{T}$ checks that expression e (expected to be closed) has (closed) user type τ when it can be checked to have the interpretation of τ within that type space $\mathcal{T}$, i.e. $\tau[\varnothing \mapsto \varnothing]^{\mathcal{T}}$. The judgement $\mathcal{H} \vdash e : \tau$ states that an expression is type-consistent with user type τ under hierarchy $\mathcal{H}$ when there is a *consistent* type space $\mathcal{T}$ in $\mathcal{H}$ that can type-check e against user type τ. Because user types form a consistent type space, this definition accepts all expressions that are checkable using only user types. So if users were advised to plan their expressions around user types, we would accept all expressions they would expect to work. They might be surprised to discover we can accept even more expressions, but they should never be particularly surprised by an expression being rejected.

The judgement $\vdash \mathcal{P}$ defines when we consider a program to be valid. The hierarchy $\mathcal{H}^{\mathcal{P}}$ is required to be valid, the user type $\tau^{\mathcal{P}}$ is required to be closed, and the expression $e^{\mathcal{P}}$ is required to be type-consistent with respect to $\tau^{\mathcal{P}}$. Due to Theorem 7.3, this notion of program validity ensures safety even though it does not ensure type-inferability. On the other hand, any sound notion of type-inferability ensures type-consistency; the (extended) grammar of types that inference is performed within defines a consistent type space, and type-inference verifies that the program type-checks within that space, thereby demonstrating type-consistency. By using type-consistency rather than type-inferability, we can add new type constructs in order to allow users to express more programs *without* any concern that such additions will change the (in)validity of any existing programs (or the run-time behavior of any existing overload resolutions). Thus type-consistency offers a permissive and stable notion of program validity even in the presence of language evolution.

8 Type-Outference

Now we formalize our type-outference algorithm for deciding type-consistency. The first stage of the algorithm is to construct a finite graphical core representing the program at hand. The configurations of the label-listeners in this graphical core represent the various ways in which its method invocations can be resolved. In particular, each configuration of the graphical core effectively defines a type space in which the program type-checks. The second stage of the algorithm, then, is to find a configuration—if any—whose corresponding type space is consistent, thereby deciding type-consistency of the program.

8.1 Graphical Cores

While the end goal is to validate a program, we primarily operate on a graphical core of components and constraints representing what is needed in order to type-check the program. The following is a formal description of the structure we informally visualized in Figure 4.

Definition 8.1. A graphical core C in a hierarchy $\mathcal{H}$ consists of

- a finite set of "abstract types" t,
- a finite set of "nodes" n, each of which has
  - an interface I^n (i.e. its "label")
  - and for each type parameter $\alpha \in \Theta^{\mathcal{H}}_{I^n}$, an "argument" member μ^n_α,
  - where a *member* μ is either an abstract type, a node, or Any,
- a finite set of "label-listeners" ℓ,
- a finite set of "events" ε, each of which has
  - a label-listener ℓ^ε,
  - an interface I^ε,
  - and for each type parameter $\alpha \in \Theta^{\mathcal{H}}_{I^\varepsilon}$, an "argument" member μ^ε_α,
- and a finite set of "constraints" κ, each of which has
  - an optional event ε^κ (using $\varepsilon^\kappa = \varnothing$ to denote when the event is absent),
  - a "lower" member $\mu^\kappa_\triangledown$,
  - and an "upper" component $\xi^\kappa_\triangle$,
  - where a *component* ξ is either a member or a label-listener.

The abstract types t represent the unknowns that traditionally would be inferred; being abstract, they have no structure themselves, but they can occur elsewhere as node arguments and constraint bounds. The nodes n represent interface types; given a user type, one can encode its syntax tree as nodes (and members) of a graphical core with the obvious labeling and arguments. The label-listeners ℓ are used both for events and as upper bounds in constraints. The events ε specify their firing condition and initial effect: an event ε fires when its label-listener ℓ^ε acquires a lower bound labeled by its interface I^ε, at which point each of its argument members μ^ε_α is constrained to be bound by the corresponding argument of the lower bound. The constraints κ indicate that their lower bounds must be subtypes of their upper bounds, sometimes conditioned upon a specified event firing; only the upper bound of a constraint is allowed to be a label-listener.

The label-listeners, events, and optional constraint events are our stateless formalization of the stateful tentative resolutions we used in Figure 4. They are combined with a *configuration* in order to represent a particular snapshot of stateful tentative resolutions.

Definition 8.2. A configuration $\sigma ::= (\ell \to I)_{\ell \in C}$ of a graphical core C is a (partial) finite map of the label-listeners ℓ of C to an interface I^σ_ℓ.

When a configuration has no entry for a given label-listener (denoted $I^\sigma_\ell = \varnothing$), it represents resolving the corresponding invocation site as unreachable.

In order to support resolution of a method m, one creates a label-listener ℓ, two abstract types t and t', and the following for each interface I directly declaring a signature $\tau \to \tau'$ for m:

- an abstract type t''_α for each type parameter $\alpha \in \Theta^{\mathcal{H}}_I$,
- an event ε with label-listener ℓ and interface I, using the newly created t''_α abstract types as its member arguments corresponding to the type parameters of I,
- nodes so that $\tau[\alpha \mapsto t''_\alpha]_{\alpha \in \Theta^{\mathcal{H}}_I}$ is represented by a member μ,
- nodes so that $\tau'[\alpha \mapsto t''_\alpha]_{\alpha \in \Theta^{\mathcal{H}}_I}$ is represented by a member μ',
- and two constraints conditioned upon ε and respectively requiring $t <: \mu$ and $\mu' <: t'$.

The effect of this is to dynamically constrain the abstract input (t) and result (t') types by an interface's method signature only when the receiver is determined to belong to that interface.

In our mechanical formalization [Tate 2025], we actually define graphical cores recursively. Rather than events, we allow a graphical core to associate combinations of label-listeners and

$$\boxed{\mathcal{H} \mid C \vdash_\sigma \varepsilon} \qquad \frac{I^\sigma_{\ell^\varepsilon} = I^\varepsilon}{\mathcal{H} \mid C \vdash_\sigma \varepsilon} \qquad \boxed{\mathcal{H} \mid C \vdash_\sigma \kappa} \qquad \frac{\varepsilon^\kappa = \varnothing}{\mathcal{H} \mid C \vdash_\sigma \kappa} \qquad \frac{\mathcal{H} \mid C \vdash_\sigma \varepsilon^\kappa}{\mathcal{H} \mid C \vdash_\sigma \kappa}$$

$$\boxed{\mathcal{H} \mid C \vdash_\sigma \mu \preceq \xi} \qquad \frac{\mathcal{H} \mid C \vdash_\sigma \kappa}{\mathcal{H} \mid C \vdash_\sigma \mu^\kappa_\triangledown \preceq \xi^\kappa_\triangle} \qquad \frac{\mathcal{H} \mid C \vdash_\sigma \mu \preceq t' \qquad \mathcal{H} \mid C \vdash_\sigma t' \preceq \xi''}{\mathcal{H} \mid C \vdash_\sigma \mu \preceq \xi''}$$

$$\frac{\mathcal{H} \mid C \vdash_\sigma n \preceq n' \qquad I^n = I^{n'} \qquad \Theta = \Theta^{\mathcal{H}}_{I^{n'}}}{\mathcal{H} \mid C \vdash_\sigma \mu^n_\alpha \preceq^\ominus_{\pm\alpha} \mu^{n'}_\alpha} \qquad \frac{\mathcal{H} \mid C \vdash_\sigma \varepsilon \qquad \mathcal{H} \mid C \vdash_\sigma n \preceq \ell^\varepsilon \qquad I^n = I^\varepsilon \qquad \Theta = \Theta^{\mathcal{H}}_{I^\varepsilon}}{\mathcal{H} \mid C \vdash_\sigma \mu^n_\alpha \preceq^\ominus_{\pm\alpha} \mu^\varepsilon_\alpha}$$

Fig. 14. Derived constraints of a graphical core

interfaces with graphical cores to be incorporated only in appropriate configurations. Thus we can dynamically add more abstract types, nodes, label-listeners, and constraints as needed (so long as the recursion is well-founded). However, while this recursive definition is both mechanically simpler and more powerful, it makes the subsequent definitions more complicated, so here we use a flattened version—specialized to the limited needs at hand—in order to simplify the subsequent presentation.

8.2 Deriving Constraints

The graphical core of a program only describes the constraints required for the program to type-check in some type space. That type space should be consistent, so from these constraints we can derive additional constraints that must also hold in any consistent type space. Figure 14 formalizes how we can derive such constraints.

The judgement $\mathcal{H} \mid C \vdash_\sigma \varepsilon$ specifies that an event ε is activated in a configuration σ when its label-listener ℓ^ε is configured by σ to have the interface I^ε expected by the event. The judgement $\mathcal{H} \mid C \vdash_\sigma \kappa$ specifies that a constraint κ is activated in a configuration σ when either it has no required event or its required event ε^κ is activated.

The judgement $\mathcal{H} \mid C \vdash_\sigma \mu \preceq \xi$ specifies when one can derive that the member μ must be a subtype of the component ξ in a configuration σ. Again, only the upper bound is allowed to be a label-listener. For this reason, we make sure we use the notation $\mathcal{H} \mid C \vdash_\sigma \mu \preceq_\pm \mu'$ only when both sides are necessarily members.

There are four ways in which constraints can be derived:

(1) By assumption, the lower and upper bound of any activated constraint κ must be subtypes.
(2) By transitivity, any lower and upper bound on a common abstract type t' must be subtypes.
(3) By consistency, if we can derive that two nodes with the same interface must be subtypes, then their arguments (for any type parameter α of the common interface) must be subtypes or supertypes according to the declared variance of α.
(4) Similarly by consistency, if we can derive that a node must be a subtype of the label-listener of an activated event with the same interface, then the node's argument μ^n_α (for any type parameter α of the common interface) must be a subtype or supertype of the event's corresponding member μ^ε_α according to the declared variance of α.

One might wonder why transitivity only applies to abstract types even though we need transitivity to hold for all types. From an algorithmic perspective, we want to need only track the lower and upper bounds of abstract types, rather than all components. From a proof perspective, nodes

$$\boxed{\mathcal{H} \vdash_\sigma C} \qquad \frac{\begin{array}{c}\text{for all } n \text{ and } n' \in C, \mathcal{H} \mid C \vdash_\sigma n \preceq n' \text{ implies } I^n = I^{n'} \\ \text{for all } n \text{ and } \ell \in C, \mathcal{H} \mid C \vdash_\sigma n \preceq \ell \text{ implies } I^n = I^\sigma_\ell \\ \text{no } n \in C \text{ satisfies } \mathcal{H} \mid C \vdash_\sigma \text{Any} \preceq n \\ \text{no } \ell \in C \text{ satisfies } \mathcal{H} \mid C \vdash_\sigma \text{Any} \preceq \ell\end{array}}{\mathcal{H} \vdash_\sigma C} \qquad \boxed{\mathcal{H} \vdash C} \qquad \frac{\mathcal{H} \vdash_\sigma C}{\mathcal{H} \vdash C}$$

Fig. 15. Graphical-core consistency

represent interface types, so we will be able to prove transitivity using the subtyping properties of the interface types they represent. As for label-listeners, the grammar of constraints is restricted so that they can never have a derived upper bound, so transitivity will never be applicable.

If one reviews the walkthrough of Figure 4, one will notice that nearly every step was simply determining more derived constraints according to the rules in Figure 14. That is, the algorithm we employed was primarily just a worklist-based means of computing the complete set of derived constraints that hold given the activated constraints.

Lemma 8.3. *For any hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, and for any graphical core C in $\mathcal{H}$ and any configuration σ of C, the set of members μ and components ξ satisfying $\mathcal{H} \mid C \vdash_\sigma \mu \preceq \xi$ is finite and computable.*

Proof. Because the sets of nodes, abstract types, and label-listeners of C are all finite, the set of potentially derivable constraints is necessarily finite. Given a finite subset of derived constraints, one can compute which instantiations of the rules in Figure 14 can be applied to derive more constraints. Because the total set of potentially derivable constraints is finite, repeatedly adding constraints to such a (initially empty) subset until it becomes closed under the rules in Figure 14 necessarily terminates. □

Thus, a key to termination is keeping the set of potentially derivable constraints finite, which we do by keeping the set of things that we can derive constraints on finite and ensuring derivations remain within this finite set. This is in contrast with the extension of structural algorithms to arbitrarily-recursive interface signatures, which adds types each time a new lower bound is added to the receiver of a method invocation, which is why it fails to validate our running examples.

8.3 Constraint-Consistency

As many have observed before, many algorithms work by reducing program validity (traditionally defined as type-inferability) to constraint-consistency [Aiken and Wimmers 1993; Aiken et al. 1994; Binder et al. 2022; Eifrig et al. 1995a,b; Jim and Palsberg 1999; Parreaux 2020; Parreaux and Chau 2022; Pottier 1998a,b; Trifonov and Smith 1996]. We do the same, albeit incorporating state into the process. Consistency of a graphical core ($\mathcal{H} \vdash C$) is defined in Figure 15 using consistency of a particular configuration ($\mathcal{H} \vdash_\sigma C$). In particular, if one can derive constrains between a node and a node or a label-listener, then their respective interfaces (in the given configuration) must be equal. Furthermore, one must not be able to derive Any as a lower bound for either a node or a label-listener. Thus, taken in combination with the rules for constraint-derivation, whenever we process a new derived constraint we should often be able to either derive further constraints or determine that the current configuration of the graphical core is inconsistent. As such, we have a decision procedure proving the following.

Theorem 8.4. *For any hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, and for any graphical core C in $\mathcal{H}$, $\mathcal{H} \vdash C$ is decidable.*

$$\boxed{\mathcal{H} \mid \mathcal{T} \models_\sigma C} \quad \frac{\begin{array}{c} f \text{ is a mapping of the components of } C \text{ to the types of } \mathcal{T} \\ \text{for all } \kappa \in C, \quad \mathcal{H} \mid C \vdash_\sigma \kappa \text{ implies } f(\mu^\kappa_\triangledown) <:^{\mathcal{T}} f(\xi^\kappa_\triangle) \\ f(\mathsf{Any}) = \mathsf{Any}^{\mathcal{T}} \\ \text{for all } n \in C, \quad I^n\langle f(\mu^n_\alpha)\rangle^{\mathcal{T}}_{\alpha \in \Theta^{\mathcal{H}}_{I^n}} <:^{\mathcal{T}} f(n) \text{ and } f(n) <:^{\mathcal{T}} I^n\langle f(\mu^n_\alpha)\rangle^{\mathcal{T}}_{\alpha \in \Theta^{\mathcal{H}}_{I^n}} \\ \text{for all } \varepsilon \in C, \quad \mathcal{H} \mid C \vdash_\sigma \varepsilon \text{ implies } f(\ell^\varepsilon) <:^{\mathcal{T}} I^\varepsilon\langle f(\mu^\varepsilon_\alpha)\rangle^{\mathcal{T}}_{\alpha \in \Theta^{\mathcal{H}}_{I^\varepsilon}} \\ \text{for all } \ell \in C, \quad I^\sigma_\ell = \varnothing \text{ implies } f(\ell) \lessdot^{\mathcal{T}} \end{array}}{\mathcal{H} \mid \mathcal{T} \models_\sigma C} \qquad \boxed{\mathcal{H} \mid \mathcal{T} \models C} \quad \frac{\mathcal{H} \mid \mathcal{T} \models_\sigma C}{\mathcal{H} \mid \mathcal{T} \models C}$$

Fig. 16. Modeling graphical cores with type spaces

Proof. Initialize σ to be the empty configuration. We trivially know that any consistent configuration must be a refinement (i.e. maps more label-listeners to actual interfaces) of σ, an invariant we will maintain as we mutate σ. We can easily prove by induction that any constraints derivable within σ are also derivable within any refinement of σ. As such, the set of derivable constraints will simply grow as we refine σ.

By Lemma 8.3, we can compute the complete finite set of derivable constraints within σ. And by the above observation, we can proceed knowing that they are also derivable for all consistent configurations. If there are any newly derived constraints upper-bounding Any by a node or a label-listener, we know that no consistent configuration exists. Otherwise, we enumerate all newly derived constraints between nodes. If the corresponding interfaces fail to be equal, we know that no consistent configuration exists. Otherwise, we proceed to enumerate all newly derived constraints between nodes and label-listeners. If σ does not specify an interface for the label-listener, then we know any consistent configuration must configure that label-listener to use specifically the node's interface, and so we mutate σ accordingly and return to deriving constraints. Otherwise, if the node's interface fails to equal σ's interface for the label-listener, then we know that no consistent configuration exists. Otherwise, we proceed to the next derived constraint. If there are no more derived constraints to consider, then σ satisfies $\mathcal{H} \vdash_\sigma C$.

Because the set of label-listeners is finite, strict refinement of configurations is well-founded, guaranteeing this stateful process terminates. □

Thus type-outference works by deciding consistency of the graphical core of a program. Of course, to prove that this decides type-consistency for the program, we need to connect graphical cores to type spaces and type-consistency.

9 Deciding Type-Consistency

So far we have a notion of program validity that ensures safety using type spaces, and we have an algorithm for deciding consistency of graphical cores, but we have not formally connected the two. To that end, we define when a type space models a graphical core, use that to establish how graphical cores capture type-checking within type spaces, and prove that modeling connects our various notions of consistency to ensure decidability.

9.1 Modeling

Type spaces are a semantic concept, and graphical cores are an algorithmic concept. Here we connect them by defining the former as models of the latter. This definition is provided in Figure 16.

A type space models a graphical core ($\mathcal{H} \mid \mathcal{T} \models C$) if it models a configuration of that graphical core ($\mathcal{H} \mid \mathcal{T} \models_\sigma C$). To do so, it must provide an interpretation f of the components of the graphical

$$\boxed{\mathcal{H} \vdash e}$$

$$\frac{}{\mathcal{H} \vdash x} \qquad \frac{\mathcal{H} \vdash e \qquad \mathcal{H} \vdash e'}{\mathcal{H} \vdash e.m(e')} \qquad \frac{\text{for all } m \in b, \quad \mathcal{H} \vdash e^b_m \qquad \text{for all } m \in \mathcal{B}^{\mathcal{H}}_I, \quad m \in b}{\mathcal{H} \vdash I\{b\}}$$

Fig. 17. Type-checkability

core as types in the type space. This interpretation must satisfy all of the constraints activated in the configuration at hand. It must interpret Any as the type space does. It must interpret nodes as equivalent to the interface types they represent. It must interpret the label-listeners of activated events to have at least the structure required by the event. And it must interpret any unresolved label-listeners with types satisfying $\preccurlyeq^{\mathcal{T}}$. In other words, a type space models a graphical core if its types provide a solution for the system of constraints specified by the graphical core.

9.2 Extraction

We can use this connection between type spaces and graphical cores to establish how graphical cores effectively capture the validation problem for expressions.

Lemma 9.1. *For any hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, for any expression e satisfying $\mathcal{H} \vdash e$, and for any user type τ satisfying $\mathcal{H} \mid \varnothing \vdash \tau : +$, there exists a graphical core C such that, for any type space $\mathcal{T}$ in $\mathcal{H}$, $\mathcal{H} \mid \mathcal{T} \vdash e : \tau$ holds if any only if $\mathcal{H} \mid \mathcal{T} \models C$ holds.*

Proof. The construction of C is tedious and bares no surprises or insights beyond the utilization of label-listeners already described in Section 8.1, so we leave this as an exercise for the reader (though it can be found in our mechanization [Tate 2025]). □

Lemma 9.1 states that, for expressions e satisfying $\mathcal{H} \vdash e$ (which we discuss next), we can reduce—without loss of generality—the question of whether an expression type-checks within a given type space $\mathcal{T}$ to whether $\mathcal{T}$ models a graphical core C we can construct from the expression.

This statement only holds for expressions e satisfying $\mathcal{H} \vdash e$, which introduces a new concept: type-checkability. Type-checkability is defined in Figure 17. It simply asserts that all objects in the expression have at least some implementation for each method of the corresponding interface (regardless of whether that implementation type-checks). This is necessary for the expression to type-check in some type space, regardless of whether that type space is consistent.

Lemma 9.2. *For any hierarchy $\mathcal{H}$, for any type space $\mathcal{T}$ in $\mathcal{H}$, for any type context Γ in $\mathcal{T}$, for any expression e, and for any type $T \in \mathcal{T}$, $\mathcal{H} \mid \mathcal{T} \mid \Gamma \vdash e : T$ implies $\mathcal{H} \vdash e$.*

Proof. Trivial induction. □

Graphical cores are designed to focus on the cases in which an expression is at least type-checkable (even if not necessarily in a consistent type space), so they rely on the simpler aspects of validation being performed beforehand.

Lemma 9.3. *For any hierarchy $\mathcal{H}$ and expression e, $\mathcal{H} \vdash e$ is decidable.*

Proof. This is a straightforward syntactic process. □

9.3 Soundness

Now that we can extract a representative graphical core C from an expression e, we demonstrate that deciding C's consistency is a sound and complete decision procedure for e's type-consistency. The first step is soundness. To establish soundness, we need to show that if C is consistent then

$$\text{Type} \quad T ::= \xi \mid I\langle \overline{T} \rangle$$

$\boxed{\mathcal{H} \mid C \vdash_\sigma T <: T}$

$$\frac{\mathcal{H} \mid C \vdash_\sigma \mu \preceq \xi}{\mathcal{H} \mid C \vdash_\sigma \mu <: \xi}$$

$$\frac{}{\mathcal{H} \mid C \vdash_\sigma T <: T} \quad \frac{\mathcal{H} \mid C \vdash_\sigma T <: T'' \quad \mathcal{H} \mid C \vdash_\sigma T' <: T''}{\mathcal{H} \mid C \vdash_\sigma T <: T''} \quad \frac{\text{for all } \pm\alpha \in \Theta_I^{\mathcal{H}},\ \mathcal{H} \mid C \vdash_\sigma T_\alpha <:_\pm T'_\alpha}{\mathcal{H} \mid C \vdash_\sigma I\langle T_\alpha \rangle_{\alpha \in \Theta_I^{\mathcal{H}}} <: I\langle T'_\alpha \rangle_{\alpha \in \Theta_I^{\mathcal{H}}}} \quad \frac{}{\mathcal{H} \mid C \vdash_\sigma I\langle \overline{T} \rangle <: \mathsf{Any}}$$

$$\frac{}{\mathcal{H} \mid C \vdash_\sigma I^n\langle \mu_\alpha^n \rangle_{\alpha \in \Theta_{In}^{\mathcal{H}}} <: n} \qquad \frac{}{\mathcal{H} \mid C \vdash_\sigma n <: I^n\langle \mu_\alpha^n \rangle_{\alpha \in \Theta_{In}^{\mathcal{H}}}} \qquad \frac{\mathcal{H} \mid C \vdash_\sigma \varepsilon}{\mathcal{H} \mid C \vdash_\sigma \ell^\varepsilon <: I^\varepsilon\langle \mu_\alpha^\varepsilon \rangle_{\alpha \in \Theta_{I^\varepsilon}^{\mathcal{H}}}}$$

$\boxed{\mathcal{H} \mid C \vdash_\sigma T \lessdot}$

$$\frac{\mathcal{H} \mid C \vdash_\sigma T <: T' \qquad \mathcal{H} \mid C \vdash_\sigma T' \lessdot}{\mathcal{H} \mid C \vdash_\sigma T \lessdot} \qquad \frac{I_\ell^\sigma = \varnothing}{\mathcal{H} \mid C \vdash_\sigma \ell \lessdot}$$

Fig. 18. Canonical type space $\mathcal{T}_{\mathcal{H},C,\sigma}$ of a graphical core C with a configuration σ in hierarchy $\mathcal{H}$

there is a type space that models C (and therefore, by Lemma 9.1, type-checks e) and is consistent. To that end, we view a graphical core as defining a custom grammar of types with a custom configuration-dependent subtyping relation, thereby specifying a type space for each configuration. An expression's graphical core, then, specifies the custom type spaces it type-checks within.

Using derived constraints, we define the canonical type space $\mathcal{T}_{\mathcal{H},C,\sigma}$ of a graphical core C with configuration σ in Figure 18. Its types are the components of the graphical core along with interface types generated therefrom. Similarly, subtyping in the type space (judgement $\mathcal{H} \mid C \vdash_\sigma T <: T$) is primarily generated from the derived constraints of the graphical core (the top rule) and the required algebraic structure of type-space subtyping (the middle line of rules). The bottom line of rules connects nodes (and label-listeners) to the types they represent (in the configuration σ) as required by any model of C. Note that there is no rule for abstract types; they are simply abstractions whose semantic behavior is entirely captured by the constraints declared (and derived) upon them.

The predicate $\lessdot^{\mathcal{T}_{\mathcal{H},C,\sigma}}$ is defined inductively via the judgement $\mathcal{H} \mid C \vdash_\sigma T \lessdot$. The first rule directly ensures the algebraic structure required of type spaces. The second rule makes any label-listener that is unresolved in the configuration σ satisfy $\lessdot$, as required by any model of σ.

The following two properties of this construction capture why it is the canonical model of σ.

Lemma 9.4. *For any hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, and for any graphical core C in $\mathcal{H}$ and configuration σ of C, $\mathcal{T}_{\mathcal{H},C,\sigma}$ is a type space in $\mathcal{H}$, and $\mathcal{H} \mid \mathcal{T}_{\mathcal{H},C,\sigma} \models_\sigma C$ holds.*

Proof. Trivial from the definition of $\mathcal{T}_{\mathcal{H},C,\sigma}$. □

Lemma 9.5. *For any hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, and for any graphical core C in $\mathcal{H}$ and configuration σ of C, $\mathcal{H} \vdash_\sigma C$ implies $\mathcal{H} \vdash \mathcal{T}_{\mathcal{H},C,\sigma}$.*

Proof. We prove that subtyping between interface types necessarily implies the expected interface equality and type-argument relationships; the other proofs follow the same reasoning.

Given a proof of subtyping between two interface types, one can easily construct a chain of subtyping proofs (and intermediating types) with no immediate applications of reflexivity or

transitivity. We will transform this chain through various steps into a chain of applications of specifically the interface rule.

First, suppose this chain contains an abstract type. Because the end types are interface types, this abstract type must occur between two proofs in the chain. Because the only applicable rule for abstract types is the one incorporating derived constraints, we know these two proofs must be derived constraints. And because we can derive transitive constraints for abstract types, we can replace these two proofs with a single proof incorporating that derived constraint, thereby removing the abstract type from the chain. Thus we can repeat this process to produce a chain of proofs without intermediating abstract types.

Now suppose this chain contains an Any. Because the right end of the chain cannot be Any, there must be some proof after this Any. Consider the rightmost proof immediately following an Any. The only possible proof is a derived constraint, thereby providing a derived upper bound on Any. By the assumed consistency of C, this upper bound cannot be a node or a label-listener. The upper bound also cannot be an abstract type because we just removed all abstract types from the chain. And it cannot be Any because then this would not be the rightmost proof. Therefore, no such derived upper bound can exist, and consequently the chain must already be free of Any.

Now suppose this chain contains a node. This node must be contained within a chain of proofs starting with a proof relating a node to its interface type (as a supertype thereof), followed by a (possibly empty) chain of derived constraints, followed by a rule relating a node (or a label-listener) to its interface type (in configuration σ through some activated event ε). Due to the earlier eliminations, and due to our syntactic restriction on derived constraints, the intermediating types in the chain of derived constraints must all be nodes. By the assumed consistency of C, the interfaces on the ends and the interfaces of all the intermediating nodes must all be equal. Consequently, for each type parameter of that common interface, we can construct chains of derived constraints between the respective arguments of these nodes (and event ε). Thus, we can replace this entire chain—including the proofs connecting the ends to their interface types—with a single proof applying the interface rule, using the reflexive and transitive composition of these chains for each argument. After repeatedly doing so, we are left with not only no nodes in the chain, but even no label-listeners in the chain due to our syntactic restriction on derived constraints.

Thus we now have a chain of proofs connected by interface types, so all proofs must be applications of the interface rule. This means that all interfaces in these applications must be the same, as required for the type space to be consistent. Furthermore, the chain of proofs between the arguments for each given type parameter of the common interface can be combined into a single proof using reflexivity and transitivity, ensuring the second requirement for consistency. □

Thus, if we furthermore incorporate Lemma 9.1, $\mathcal{T}_{\mathcal{H},C,\sigma}$ establishes type-consistency of the expression that C was extracted from if C is decided to be consistent in some configuration σ.

9.4 Completeness

With soundness established, we move on to completeness.

Lemma 9.6. *For any hierarchy $\mathcal{H}$ satisfying $\vdash \mathcal{H}$, for any graphical core C in $\mathcal{H}$, and for any type space $\mathcal{T}$ in $\mathcal{H}$ satisfying $\mathcal{H} \mid \mathcal{T} \models C$, $\mathcal{H} \vdash \mathcal{T}$ implies $\mathcal{H} \vdash C$.*

Proof. Let σ and f be the configuration and mapping evidencing that $\mathcal{T}$ models C. Given nodes n and n' satisfying $\mathcal{H} \mid C \vdash_\sigma n \preceq n'$, modeling and consistency are easily shown to imply that $f(n)$ is a subtype of $f(n')$. Furthermore, modeling implies that $f(n)$ is a supertype of $I^n\langle \mu_\alpha^n \rangle_{\alpha \in \Theta_{I^n}^{\mathcal{H}}}$ and that $f(n')$ is a subtype of $I^{n'}\langle \mu_\alpha^{n'} \rangle_{\alpha \in \Theta_{I^{n'}}^{\mathcal{H}}}$. By transitivity and consistency of $\mathcal{T}$, I^n and $I^{n'}$ must then be equal, as required for consistency of C. Similar reasoning proves the remaining requirements. □

Thus, if C were decided to be inconsistent, no type space that can type-check the expression that C was extracted from can be consistent, ensuring the expression is type-inconsistent.

9.5 Decidability

At this point, we have all the pieces necessary to decide type-consistency of a program.

Theorem 9.7. *Given a program $\mathcal{P}$, $\vdash \mathcal{P}$ is decidable.*

Proof. By Lemma 6.2, we can decide whether or not $\vdash \mathcal{H}^{\mathcal{P}}$ holds, rejecting $\mathcal{P}$ if not. By Lemma 6.1, we can decide whether or not $\mathcal{H}^{\mathcal{P}} \mid \varnothing \vdash \tau^{\mathcal{P}} : +$ holds, rejecting $\mathcal{P}$ if not. By Lemma 9.3, we can decide whether or not $\mathcal{H}^{\mathcal{P}} \vdash e^{\mathcal{P}}$ holds, rejecting $\mathcal{P}$ if not by Lemma 9.2. Thus, by Lemma 9.1, we can extract a representative graphical core C from $e^{\mathcal{P}}$. Lastly, by Theorem 8.4, we can decide whether or not $\mathcal{H}^{\mathcal{P}} \vdash C$ holds. Consider each case as follows.

Suppose $\mathcal{H}^{\mathcal{P}} \vdash C$ holds, necessarily due to some configuration σ. Then by Lemma 9.5 the type space $\mathcal{T}_{\mathcal{H}^{\mathcal{P}},C,\sigma}$ is consistent. Furthermore, by Lemma 9.4, $\mathcal{T}_{\mathcal{H}^{\mathcal{P}},C,\sigma}$ models σ. Consequently, by the property of C ensured by Lemma 9.1, $\mathcal{H}^{\mathcal{P}} \mid \mathcal{T}_{\mathcal{H}^{\mathcal{P}},C,\sigma} \vdash e^{\mathcal{P}} : \tau^{\mathcal{P}}$ holds, and so $\mathcal{T}_{\mathcal{H}^{\mathcal{P}},C,\sigma}$'s consistency demonstrates $\mathcal{P}$'s type-consistency.

Suppose $\mathcal{H}^{\mathcal{P}} \vdash C$ does not hold. Assume $\mathcal{P}$ is type-consistent, as demonstrated by some type space $\mathcal{T}$ satisfying $\mathcal{H}^{\mathcal{P}} \vdash \mathcal{T}$ and $\mathcal{H}^{\mathcal{P}} \mid \mathcal{T} \vdash e^{\mathcal{P}} : \tau^{\mathcal{P}}$. By the property of C ensured by Lemma 9.1, the latter implies $\mathcal{H}^{\mathcal{P}} \mid \mathcal{T} \models C$ must also hold. Thus, by Lemma 9.6, the consistency of $\mathcal{T}$ implies consistency of C. This contradicts our supposition. Therefore, $\mathcal{P}$ must be type-inconsistent. □

Thus, by Theorem 9.7, our novel event-driven type-outference algorithm decides type-consistency, our novel notion of program validity that ensures safety—by Theorem 7.3—without ensuring type-inferability. Reviewing the algorithm, we can strengthen our decidability result: the algorithm is polynomial time with respect to the size of the program.[2] This is ensured by the fact that constraint-derivation is *directed*: we always derive a *conjunction* of constraints (or an inconsistency)—never a disjunction—which avoids the need for backtracking. So, by taking unknown types *out* of program validation and constraint reasoning, rather than *inferring* them to reduce constraints or type-check the program, we made program validation not only decidable but efficiently so. Furthermore, by using type-consistency rather than type-inferability, we were able to formalize and verify our definitions and claims in Rocq [Tate 2025] despite Rocq's metatheoretic limitations [Pédrot 2015].

10 Conclusion

We have presented the first sound and complete decision procedure capable of validating nominal object-oriented programs without type annotations in expressions (beyond the interface name that each object instantiates) wherein generic interfaces can have irregularly-recursive signatures. Such interface signatures are exemplary of the challenges that an algorithm must overcome in order to scale to the needs of modern major typed object-oriented languages. We did so by incorporating event-driven label-listeners into constraint-derivation, enabling more complex aspects of the validation problem at hand to be generated on demand while still ensuring termination. Furthermore, we mechanically verified our algorithm. We did so by defining type-consistency as a novel declarative notion of program validity that removes the need to construct complex solutions for our graphical cores even after we validated their consistency. As a consequence, our algorithm does not infer types, so we call it a type-outference algorithm. These concepts and techniques—type-consistency, type-outference, and label-listeners—are the foundations upon which we aim to make Kotlin—which this work is intended for—decidable. Obviously there are many more features we need to address; fortunately, preliminary investigations suggest that we can adapt them cleanly and efficiently.

[2]We have made no attempt to mechanically verify this claim.

Data-Availability Statement

Our mechanical formalization and verification of our definitions and theorems is provided in a single Rocq file available online through the ACM DL [Tate 2025].

Acknowledgments

This work was funded in full by JetBrains as consultation for the Kotlin team. I owe much to this team for skillfully illustrating the inadequacies of my prior approaches and supporting me in exploring riskier—but ultimately fruitful—avenues. I am grateful to Jens Palsberg for his historical insights into various early works in this space when this project first began; speaking as someone with a reading disability, when it can take an entire day to realize a paper is not what you were looking for, having someone direct you towards the most relevant works saves weeks of wasted effort. I greatly appreciate Fabian Muehlboeck's many excellent suggestions for improving the presentation of early renditions of this work. I am thankful to the programming-languages research group at the University of California, San Diego for helping me pin down intuitions for conveying key concepts. Finally, I would like to recognize the substantial time and effort the review committee put into contributing to the exposition of this paper.

References

Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 31–41. doi:10.1145/165180.165188

Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. 1994. Soft Typing with Conditional Types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. Association for Computing Machinery, New York, NY, USA, 163–173. doi:10.1145/174675.177847

David Binder, Ingo Skupin, David Läwen, and Klaus Ostermann. 2022. Structural Refinement Types. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2022)*. Association for Computing Machinery, New York, NY, USA, 15–27. doi:10.1145/3546196.3550163

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyễn. 2016. Set-Theoretic Types for Polymorphic Variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 378–391. doi:10.1145/2951913.2951928

Henry DeYoung, Andreia Mordido, Frank Pfenning, and Ankush Das. 2024. Parametric Subtyping for Structural Parametric Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 90 (Jan. 2024), 31 pages. doi:10.1145/3632932

Stephen Dolan. 2017. *Algebraic Subtyping*. Ph. D. Dissertation. University of Cambridge.

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 60–72. doi:10.1145/3009837.3009882

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995a. Sound Polymorphic Type Inference for Objects. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*. Association for Computing Machinery, New York, NY, USA, 169–184. doi:10.1145/217838.217858

Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995b. Type Inference for Recursively Constrained Types and its Application to OOP. *Electronic Notes in Theoretical Computer Science* 1 (1995), 132–153. doi:10.1016/S1571-0661(04)80008-2 MFPS XI, Mathematical Foundations of Programming Semantics, Eleventh Annual Conference.

You-Chin Fuh and Prateek Mishra. 1988. Type Inference with Subtypes. In *ESOP '88*. Springer Berlin Heidelberg, Berlin, Heidelberg, 94–114. doi:10.1007/3-540-19027-9_7

You-Chin Fuh and Prateek Mishra. 1989. Polymorphic Subtype Inference: Closing the Theory-Practice Gap. In *TAPSOFT '89*. Springer Berlin Heidelberg, Berlin, Heidelberg, 167–183. doi:10.1007/3-540-50940-2_35

You-Chin Fuh and Prateek Mishra. 1990. Type Inference with Subtypes. *Theoretical Computer Science* 73, 2 (1990), 155–175. doi:10.1016/0304-3975(90)90144-7

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded Polymorphism into Shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 89–99. doi:10.1145/2594291.2594308

Trevor Jim and Jens Palsberg. 1999. Type inference in systems of recursive types with subtyping. (1999). https://web.cs.ucla.edu/~palsberg/draft/jim-palsberg99.pdf

Stefan Kaes. 1992. Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, New York, NY, USA, 193–204. doi:10.1145/141471.141540

Andrew Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. (Jan. 2007). https://www.microsoft.com/en-us/research/publication/on-decidability-of-nominal-subtyping-with-variance/

Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1994. Efficient Inference of Partial Types. *J. Comput. System Sci.* 49, 2 (1994), 306–324. doi:10.1016/S0022-0000(05)80051-0

John C. Mitchell. 1984. Coercion and Type Inference. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. Association for Computing Machinery, New York, NY, USA, 175–185. doi:10.1145/800017.800529

Jens Palsberg and Patrick O'Keefe. 1995. A Type System Equivalent to Flow Analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 367–378. doi:10.1145/199448.199533

Jens Palsberg and Michael I. Schwartzbach. 1992. Safety Analysis versus Type Inference for Partial Types. *Inform. Process. Lett.* 43, 4 (1992), 175–180. doi:10.1016/0020-0190(92)90196-3

Jens Palsberg and Michael I. Schwartzbach. 1995. Safety Analysis versus Type Inference. *Information and Computation* 118, 1 (1995), 128–141. doi:10.1006/inco.1995.1058

Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. 1997. Type Inference with Non-structural Subtyping. *Form. Asp. Comput.* 9, 1 (Jan. 1997), 49–67. doi:10.1007/BF01212524

Lionel Parreaux. 2020. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. doi:10.1145/3409006

Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 141 (Oct. 2022), 30 pages. doi:10.1145/3563304

Pierre-Marie Pédrot. 2015. Issue coq/coq#4261: Nested coinductive types are useless. (June 2015). https://github.com/coq/coq/issues/4261

Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. Association for Computing Machinery, New York, NY, USA, 305–315. doi:10.1145/143165.143228

François Pottier. 1998a. A Framework for Type Inference with Subtyping. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 228–238. doi:10.1145/289423.289448

François Pottier. 1998b. *Type Inference in the Presence of Subtyping: From Theory to Practice*. Ph. D. Dissertation. Institut National de Recherche en Informatique et en Automatique. https://inria.hal.science/inria-00073205

Tatsurou Sekiguchi and Akinori Yonezawa. 1994. A Complete Type Inference System for Subtyped Recursive Types. In *Theoretical Aspects of Computer Software*. Springer Berlin Heidelberg, Berlin, Heidelberg, 667–686.

Geoffrey Seward Smith. 1991. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. Ph. D. Dissertation. Cornell University. https://hdl.handle.net/1813/7070

Ryan Stansifer. 1988. Type Inference with Subtypes. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 88–97. doi:10.1145/73560.73568

Ross Tate. 2025. *Rocq Formalization and Verification for the OOPSLA 2025 Article 'Type-Outference with Label-Listeners: Foundations for Decidable Type-Consistency for Nominal Object-Oriented Generics'*. doi:10.1145/3747411

Jerzy Tiuryn and Mitchell Wand. 1993. Type Reconstruction with Recursive Types and Atomic Subtyping. In *TAPSOFT'93: Theory and Practice of Software Development*. Springer Berlin Heidelberg, Berlin, Heidelberg, 686–701.

Valery Trifonov and Scott Smith. 1996. Subtyping Constrained Types. In *Static Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 349–365. doi:10.1007/3-540-61739-6_52

Received 2025-03-25; accepted 2025-08-12