When Subtyping Constraints Liberate*†

A Novel Type Inference Approach for First-Class Polymorphism

LIONEL PARREAUX, HKUST, Hong Kong, China
ALEKSANDER BORUCH-GRUSZECKI, EPFL, Switzerland
ANDONG FAN, HKUST, Hong Kong, China
CHUN YIN CHAU, HKUST, Hong Kong, China

Type inference in the presence of *first-class* or *"impredicative" second-order polymorphism* à la System F has been an active research area for several decades, with original works dating back to the end of the 80s. Yet, until now many basic problems remain open, such as how to type check expressions like $(\lambda x.\ (x\ 123,\ x\ \mathsf{True}))\ \mathsf{id}$ reliably. We show that a type inference approach based on *multi-bounded polymorphism*, a form of implicit polymorphic subtyping with multiple lower and upper bounds, can help us resolve most of these problems in a uniquely simple and regular way. We define $\mathrm{F}_{\{\leq\}}$, a declarative type system derived from the existing theory of implicit coercions by Cretin and Rémy, and we introduce SuperF, a novel algorithm to infer polymorphic multi-bounded $\mathrm{F}_{\{\leq\}}$ types while checking user type annotations written in the syntax of System F. We use a recursion-avoiding heuristic to guarantee termination of type inference at the cost of rejecting some valid programs, which thankfully rarely triggers in practice. We show that SuperF is vastly more powerful than all first-class-polymorphic type inference systems proposed so far, significantly advancing the state of the art in type inference for general-purpose programming languages.

CCS Concepts: • **Theory of computation** → **Type theory**; • **Software and its engineering** → *Functional languages*; *Polymorphism*.

Additional Key Words and Phrases: type inference, first-class polymorphism, subtyping, constraint solving

1 INTRODUCTION

Consider the function `foo` defined as: `foo f = (f 123, f True)`.

For the past several decades, researchers have been unable to decide which type to infer for this function, should one wish to go beyond the classical Hindley-Milner type inference discipline of ML languages (which yields a unification error here), and should one want to support some form of first-class polymorphism. In particular, no system was proposed where such a term could be given a "satisfactory" type. A satisfactory type for `foo` should let us type check expressions like `foo (fun x → x)` at type `(Int, Bool)` and `foo (fun x → Some x)` at type `(Option Int, Option Bool)`.[1]

In this paper, we propose SuperF, a system which can infer the following type for `foo`:

```
foo : ∀ a b. ((Int → a) ∧ (Bool → b)) → (a, b)
```

which is really just syntax sugar for the following polymorphic type that incorporates several bounds for its extra type parameter `c`:

```
foo : ∀ a b c {c ≤ Int → a, c ≤ Bool → b}. c → (a, b)
```

This type is *satisfactory* in that it lets us type check the function call examples above at the desired respective types `(Int, Bool)` and `(Option Int, Option Bool)`.

*The title of this paper took inspiration from that of a talk by Runar Bjarnason: *Constraints Liberate, Liberties Constrain.*
†This is a technical report version of the conference paper that appeared in POPL 2024 [Parreaux et al. 2024].
[1]MLsub, which uses Algebraic Subtyping [Dolan and Mycroft 2017], would infer types `(Int ∨ Bool, Int ∨ Bool)` and `(Option (Int ∨ Bool), Option (Int ∨ Bool))`, which are imprecise and thus unsatisfactory.

Authors' addresses: Lionel Parreaux, parreaux@ust.hk, HKUST, Hong Kong, China; Aleksander Boruch-Gruszecki, aleksander.boruch-gruszecki@epfl.ch, EPFL, Switzerland; Andong Fan, afanab@cse.ust.hk, HKUST, Hong Kong, China; Chun Yin Chau, cychauab@connect.ust.hk, HKUST, Hong Kong, China.

Our approach is reminiscent of *bounded polymorphism*, but our type system $F_{\{\leq\}}$ diverges from most research on this topic in the following crucial respects:

- We assume *implicit*, *erased* polymorphism, meaning that polymorphic types do not need to be introduced and eliminated in terms and do not participate in the term's runtime semantics. Consequently, we can have subtyping relationships such as $\forall\alpha.\,\alpha \to \alpha \leq \mathsf{Int} \to \mathsf{Int}$, so that any term with the left-hand side type is known to *also* have the right-hand side type.
- We allow lower bounds *as well as* upper bounds, rather than only upper bounds (like in System $F_{<:}$) or only lower bounds (like in ML^F), making our system symmetric, but introducing possible inconsistencies between bounds, which need to be dealt with carefully.
- We allow associating an *arbitrary number of bounds* to the same type variable, instead of just one,[2] reminiscent of the old idea of *constrained types* [Odersky et al. 1999], which was previously studied mostly in the context of ML-style (i.e., not *first-class*) polymorphism.

Fortunately, this combination of type system features was already formally described as part of System F_{cc} by Cretin and Rémy [2014] in their groundbreaking work laying the foundations for general implicit polymorphism with subtyping. This allows us to define $F_{\{\leq\}}$ by translation to F_{cc} and focus on the *algorithmic* aspects of the system in this paper, noting that Cretin and Rémy only presented a *declarative* type system for System F_{cc} and did not investigate type inference.

While SuperF crucially relies on multi-bounded types internally, these types are non-denotable: they are not accessible to users, who are limited to writing type annotations in the syntax of System F, i.e., where polymorphic types cannot have bounds on their quantified type variables. Indeed, to make subtype constraint solving tractable, SuperF internally follows a *polarized* type syntax, whereby bounds are only allowed in *positive* polymorphic types. Annotated types cannot use bounds because these types are both positive *and* negative, since they *provide* a type for the annotated expression but also are *checked* against the expression's inferred type This is an atypical property of our system: it means that users cannot always provide explicit type signatures that are as general as the types that are internally inferred by the system. Nevertheless, we argue that in practice users rarely want to use bounds in their type signatures anyway, and that System F types are sufficient for a large number of practical functional programming use cases, including a majority of use cases studied in previous work.

Subtyping for System F types was historically realized in System F_η, which relates polymorphic types by subtyping based on their specificity. F_η has an "η rule" to subtype function types covariantly in their results and contravariantly in their arguments, which makes System F typing complete with respect to η conversion. Subtyping in F_η is known to be undecidable [Chrząszcz 1998; Tiuryn and Urzyczyn 1996], and $F_{\{\leq\}}$ is a strict generalization of F_η (i.e., all well-typed System F and F_η terms are well-typed $F_{\{\leq\}}$ terms, but the converse does not hold), so it is natural to conjecture that $F_{\{\leq\}}$ subtyping is also undecidable, making type inference necessarily incomplete.

SuperF is a *terminating* but *incomplete* type inference algorithm for $F_{\{\leq\}}$ that relies on a simple recursion-detection heuristic to raise errors in tricky recursive-looking cases. Without this check, type inference would diverge on terms that exhibit indirectly-recursive structures, such as the Ω combinator `(fun x → x x) (fun x → x x)`. Thankfully, we find that this check rarely triggers in practical, real-world examples. On the other hand, even when it succeeds, SuperF does not always infer principal types. This is mainly because it always distributes polymorphic types over function types even though that can sometimes lead to worse outcomes (an example is given as G14 in

[2]Intuitively, this is equivalent in power to allowing at most one upper bound and at most one lower bound on each type variable together with allowing intersection types in negative positions and union types in positive positions [Parreaux 2020], which is why we could represent the type of `foo` more concisely via an intersection in the example above.

Section 5.5). Whether we always infer principal types in a restricted system without distributivity is currently an open question.

From the programmer's perspective, the main limitation of SuperF type inference can be understood as: *SuperF never assumes parametrically-polymorphic types for function parameters.* Type annotations must be used when such polymorphic parameters are needed. Nevertheless, SuperF is appreciably more powerful than all other existing systems; we show that it can seamlessly handle most examples previously considered in the literature (a first), and that it does so *without requiring any annotations whatsoever.* Indeed, multi-bounded polymorphic subtyping already covers a lot of additional ground compared to existing unification-based approaches, allowing programs like the one at the beginning of this introduction to type check without having to resort to higher-rank parametric polymorphism where other systems would have to.

Our specific contributions are as follows:

- We describe the main problems of first-class-polymorphic type inference as well as our main ideas to address these problems (Section 2). To the best of our knowledge, although they are quite natural in retrospect, these ideas are novel and have not been developed before.
- We formalize $F_{\{\leq\}}$, a lambda calculus with first-class, implicit, erased, and what we call *multi-bounded* polymorphism (Section 3). We show that the soundness of this system can be obtained by translation into the existing System F_{cc}, whose semantic soundness was mechanically verified in Coq by Cretin [2014].
- We formalize our SuperF type inference system, emphasizing the novel and crucial concept of *subtype extrusion and avoidance* (Section 4). We prove that this type inference system is sound for $F_{\{\leq\}}$ and terminating.
- We present a practical implementation of SuperF (Section 5), which can be tried in a web demo available at https://hkust-taco.github.io/superf/. This implementation was evaluated on all examples described in previous work as well as many new and less trivial ones, demonstrating that our system is considerably more powerful than previously proposed approaches. We also show that SuperF infers precise yet concise types for existing ML programs by porting the `List` module from OCaml's standard library to SuperF.

2 PRESENTATION

We first present our approach from a high-level point of view focused on intuition.

2.1 Motivation

Before we delve into technical details, it is worth considering *why* one might care about first-class polymorphism and why impredicative polymorphic type inference is a useful addition to a functional programming language's type system.

Early on, Peyton Jones et al. [2007] presented many convincing use cases for *higher-rank polymorphism*, a restricted form of first-class polymorphism. These examples included data structure fusion, encapsulation of state and other effects, existential type encodings, generic programming, folding and mapping functions respecting data structure invariants, internal type class desugaring, etc. Nowadays, higher-rank polymorphism is used pervasively throughout the Haskell ecosystem.

Peyton Jones et al. [2007, §3.4] also presented several use cases for *impredicative* polymorphic type inference (i.e., unrestricted first-class polymorphism, as studied in this paper). Their examples rely on the observation that without impredicative polymorphism, several abstractions that work

well in the monomorphic case start failing in the polymorphic case, requiring tedious manual specialization. For instance, consider the following definitions:

```
revapp : ∀ a. a → (a → b) → b      |     poly : (∀ v. v → v) → (Int, Bool)
revapp x f = f x                   |     poly f = (f 3, f True)
```

While both of these functions fit into classical higher-rank polymorphism, applying one to the other, as in `(revapp (fun x → x) poly)`, requires *impredicatively* instantiating the type variable `a` to type `∀ v. v → v`. A similar thing happens with the fix-point combinator, whose usual type, `fix : (a → a) → a`, cannot be applied to a polymorphic function without impredicative polymorphism.

As another example, recent versions of the Haskell language started supporting abstracting over record fields through a special type class so that, for example, it became possible to write functions working over any input type that contains a field named `"size"` of type `a`. Allowing this useful abstraction infrastructure to extend to the setting of records with *polymorphic* fields requires impredicative polymorphism, as it requires instantiating `a` in our example with a polymorphic type.

Altogether, supporting first-class polymorphism in its "full glory" has many important real-world benefits, which also explains why the Haskell community has spent over a decade trying to add such generalized support to the Glasgow Haskell Compiler (GHC) [Serrano et al. 2020, 2018; Vytiniotis et al. 2006] and why even languages like Scala 3 are starting to add the feature.[3]

The solution we propose in this paper is a general one: SuperF supports higher-rank polymorphism as a side effect of its support for unrestricted first-class polymorphism, and all classical examples of predicative higher-rank polymorphism are still supported out of the box in SuperF.

2.2 Parametricity and Subtyping

Recall the definition of function `foo` shown in the introduction: `foo f = (f 123, f True)`

To be polymorphic or not to be. Clearly, function parameter `f` should be polymorphic one way or another in order for `foo` to be well-typed, because `f` is applied to arguments of completely different types `Int` and `Bool`. So an intuitive type for `foo` could have the following form:

```
foo : (∀ a. a → ?) → (?, ?)
```

This leads us to a first problem: what type should the parameter function `f` be expected to *return*? The simplest answer that comes to mind would be to have it return the same type as its input:

```
foo : (∀ a. a → a) → (Int, Bool)
```

While this is possible, it is obviously not general enough — what if the caller wanted to pass to `foo` the function `fun x → Some x`? So far, most previous state-of-the-art approaches to first-class polymorphism already throw in the towel and ask the user to provide an explicit type annotation for `f`. The user could annotate `f` as `∀ a. a → a` or they could annotate it as `∀ a. a → Option a`, or as `∀ a. a → List a`, etc., etc., depending on their intended use of `foo`. Moreover, these parametric polymorphism considerations are not *stable* in the face of small changes. Consider:

```
foo1 f = (f E0, f E1)
```

where `E0` and `E1` are two arbitrary expressions. If `E0` and `E1` happen to be typeable at the same type `T`, `f` no longer needs to be polymorphic, which allows calling `foo1` with *less powerful* non-polymorphic function arguments of type `T → R` for some `R`. This may in turn be needed later, depending on the uses of `foo1`. For example, taking `E0 = E1 = fun x → x`, we have:

```
foo2 f = (f (fun x → x), f (fun x → x))
foo2 : ∀ a b. ((a → a) → b) → (b, b)
```

[3] See *polymorphic function types*: https://docs.scala-lang.org/scala3/reference/new-types/polymorphic-function-types.html.

This typing of `foo2` would let us write expressions like `foo2 (fun f → f 1 > 0)`, of type `(Bool, Bool)`. But the parameter-polymorphic type is still on the table, and is neither more nor less general:

```
foo2 : (∀ a. a → a) → ((∀ a. a → a), (∀ a. a → a))
```

which would allow us to type `foo2 id` as `((∀ a. a → a), (∀ a. a → a))`. The fundamental *instability* of this kind of reasoning gets even worse whenever we pass to `f` argument that *may or may not* be typed identically depending on polymorphism decisions made earlier in type inference, as in:

```
foo3 f = (f (fun (x, y) → (y, x)), f (fun (x, y) → (x, y)))
```

Above, we could type both arguments to `f` at the same type `∀ a. (a, a) → (a, a)`, or we could type them at unrelated types `∀ a b. (a, b) → (b, a)` and `∀ a b. (a, b) → (a, b)` respectively, and again there is no most general choice. Many types seem possible, and there is no obvious way to proceed. This leads us to formulate another major problem: when *should* inferred types be polymorphic and when should they *not* be? In practice, these two problems make the job of a type inference engine for System F unpleasantly cumbersome, and the need for a form of subtyping is felt [Rémy 2005].

Subtype polymorphism and intersection types. Our failure to find a most general type for `foo` was due to a belief that `f` should be assigned a *parametric* polymorphic type. Parametric polymorphism is a form of *infinitary* polymorphism [Aiken and Wimmers 1993; Leivant 1990] — that is, a function `f` of type `∀ a. a → a` can be seen as a function having at the same time all the types `f : Int → Int`; `f : Bool → Bool`; `f : Option Int → Option Int`; etc. In type theory, we can express such *overloaded* function types through *intersection types* (∧), as in:

```
f : (Int → Int) ∧ (Bool → Bool) ∧ (Option Int → Option Int) ∧ ...
```

It should now become clear that assigning to the `f` parameter of `foo` an infinitary intersection type is definitely "overkill". After all, `f` is only applied to `Int` and `Bool` arguments, so a type such as `(Int → S) ∧ (Bool → T)` should suffice. Since there are no constraints on `S` and `T`, these can be taken to be type variables of `foo`, yielding the type:

```
foo : ∀ a b. ((Int → a) ∧ (Bool → b)) → (a, b)
```

This type (call it τ_0) happens to be a *principal* or *most general* type of `foo`,[4] meaning that all other types that can be assigned to `foo` can be obtained by widening τ_0 through subtyping.

Proof. Assume that `foo` has type τ. Since `foo` is a function, τ must be a supertype of $\tau_1 \to \tau_2$ for some τ_1, τ_2, i.e., $\tau_1 \to \tau_2 \leq \tau$ (τ is not necessarily a function type; for instance, it could be $\top$). Since `foo` returns a pair, we must have $(\sigma_1, \sigma_2) \leq \tau_2$ for some σ_1, σ_2. Since `f` is applied to an argument of type `Int`, resulting in type σ_1, we must have $\tau_1 \leq \mathsf{Int} \to \sigma_1$. Similarly, we must have $\tau_1 \leq \mathsf{Bool} \to \sigma_2$. We need to show that $\tau_0 = \forall\, \alpha\, \beta.\ ((\mathsf{Int} \to \alpha) \wedge (\mathsf{Bool} \to \beta)) \to (\alpha, \beta) \leq \tau$, a relationship which is implied by $\tau_0 \leq \tau_1 \to \tau_2$. To show the latter, instantiate the left-hand side by taking $\alpha = \sigma_1$ and $\beta = \sigma_2$, resulting in $((\mathsf{Int} \to \sigma_1) \wedge (\mathsf{Bool} \to \sigma_2)) \to (\sigma_1, \sigma_2) \leq \tau_1 \to \tau_2$, which works because that is equivalent to: • $\tau_1 \leq (\mathsf{Int} \to \sigma_1) \wedge (\mathsf{Bool} \to \sigma_2)$, i.e., $\tau_1 \leq \mathsf{Int} \to \sigma_1$ and $\tau_1 \leq \mathsf{Bool} \to \sigma_2$; and • $(\sigma_1, \sigma_2) \leq \tau_2 = (\sigma_1, \sigma_2)$. □

Notice that τ_0 does not even involve any higher-rank parametric polymorphism! (Instead, it has a higher-rank intersection type.) Yet, a function of this type can be called with parametrically-polymorphic arguments, such as functions of type $\tau_{\mathsf{id}} = \forall\, \mathtt{c}.\ \mathtt{c} \to \mathtt{c}$ which should not come as a surprise, as we saw that τ_{id} stands for `(Int → Int) ∧ (Bool → Bool) ∧ ...`, which is clearly a subtype of `(Int → Int) ∧ (Bool → Bool)`. Similarly, we can now also call `foo` with argument `fun x → Some x`, of type `∀ c. c → Option c`, which yields a result of type `(Option Int, Option Bool)`. As a less semantic and more syntactic way to see why the latter works, instantiate `a` to `Option Int` and `b` to `Option Bool`

[4] SuperF does not always infer principal types when they exist, but it does so in many simple cases like this.

in the type of `foo`, and notice that `∀ c. c → Option c` is indeed a subtype of `(Int → Option Int) ∧ (Bool → Option Bool)` because that is the same as saying that `∀ c. c → Option c` is a subtype of `Int → Option Int` *and also* a subtype of `Bool → Option Bool`, which is true as in each cases we can instantiate the polymorphic type by substituting `c` accordingly. Furthermore, a function of type τ_0 can *also* be called with monomorphic arguments, of types such as `⊤ → Str` (where `Int ≤ ⊤` and `Bool ≤ ⊤`). So it is more general than all System F types we have previously considered for it.

This idea generalizes smoothly to the abstracted example `foo1` mentioned earlier:

```
foo1 f = (f E0, f E1)      |      foo1 : ((T0 → a) ∧ (T1 → b)) → (a, b)
```

where `T0` and `T1` are the inferred types of respectively `E0` and `E1`. Thanks to subtyping, whether the types chosen for `E0` and `E1` happen to be compatible or not has become irrelevant. This effectively removes the "discontinuity" in reasoning between these two very similar cases: we no longer have to reason about awkward cases where different, incomparable types suddenly become possible due to small decisions made when inferring the types of subterms.

All in all, picking a *parametrically*-polymorphic type for `f` makes it more powerful than strictly necessary, and this is a problem because that type occurs *negatively* (i.e., in *input* position), meaning that its excessive power leads to a *weaker* overall type, preventing the discovery of a sufficiently general one. All we had to do was to abandon the infinitary context of parametric polymorphism and assume the finitary context of intersection-based subtype polymorphism instead.

Union types. A dual phenomenon happens with union types in positive positions. Unions give us a natural least upper bound between arbitrary types, even when these types have very different shapes and structures, while other approaches would have to approximate each type to a common and often less precise super type, a process that may not always have good solutions. Consider:

```
bar x = if <condition> then x else id
```

where `id` has type `∀ a. a → a`. Because `x` and `id` flow together into the result type, most previous approaches would try to *unify* their respective types into a unique result type. But there is no best way of performing such unification: we could make `x` have the same polymorphic type as `id`, but this would prevent legitimate uses of `bar` such as `bar succ 0` (where `succ: Int → Int`), which could be typed at `Int` if we instantiated the type of `id` to `Int → Int` while typing `bar`. In SuperF, we have:

```
bar : ∀ a. a → (a ∨ ∀ b. b → b)
```

which our implementation automatically simplifies by distributing the inner quantifier out:

```
bar : ∀ a b. a → (a ∨ (b → b))
```

and which does not impose any premature decisions on the polymorphism of the `x` parameter and on the instantiation of `id`'s type. SuperF correctly infers type `∀ b. (Int → Int) ∨ (b → b)` for `bar succ`, which our implementation simplifies to `Int → Int` (because `∀ b. b → b` is a subtype of `Int → Int`).

Going beyond System F. System F only allows expressing the infinitary form of polymorphism (i.e., parametric polymorphism). We prefer inferring types in a slightly more powerful language which smoothes out the rugged edges and discontinuities of System F's type language. The idea is not new: it goes back at least to ML^F, which used bounded quantification to allow a notion of subsumption (referred to as *generic instance*) to be used in inferred types, so as to abstract over the different possible generalization choices made by users of polymorphic functions. Yet, ML^F still required type annotations in many places, notably for all parameters used polymorphically, and it could not type check the `foo` function variations shown above (except for `foo2`) because it only allowed *one lower bound* per type variable, instead of multiple lower and upper bounds. So we argue that ML^F did not go far enough, and that what we actually need is full-blown polymorphic subtyping and full-blown multi-bounded polymorphism. Because it is a *superset* of System F, we call the type inference system presented in this paper *SuperF*.

On the fluidity of subtyping. Our declarative type system $F_{\{\leq\}}$ allows specifying bounded polymorphic types and unions in arbitrary positions, while the SuperF subset of $F_{\{\leq\}}$, used for type inference, only allows such types in positive positions. Moreover, SuperF types themselves are a superset of System F types. Here is an analogy to justify this design and to explain the ease of our approach compared to the difficulty of inferring pure System F types: one can think of System F types as integer numbers, SuperF types as real numbers, and $F_{\{\leq\}}$ types as imaginary numbers. It is much easier to find *real* solutions to polynomials than it is to find *integer* solutions to them, i.e., diophantine equations. Making an analogy between the problem of finding valid type assignments and the problem of finding polynomial roots, the ruggedness of System F types (integer numbers) forces one into making awkward discrete choices, whereas the fluidity of SuperF subtyping (real numbers) smoothes over these choices, letting one pick most general solutions to sub-problem in a way that composes well. But the best way of *explaining* how to find SuperF type solutions to type inference problems is through the more general framework of $F_{\{\leq\}}$ (imaginary numbers), which generalizes the subtyping relation (numeric domain) further yet.

2.3 SuperF and Implicit Multi-Bounded Polymorphism

We saw that intersections were better at expressing *requirements* on the types of *inputs*, since making these requirements as weak as possible conversely strengthens the overall type. On the other hand, in this paper we are not interested in using intersections in *positive* (output) position, although such intersections can be used to represent various features, such as ad-hoc function overloading. For example, in some programming languages like TypeScript and in semantic subtyping approaches [Frisch et al. 2002], one can write a definition like `dup x = (x, x)` and assign it *several* type signatures, such as `Int → (Int, Int)` *and* `Bool → (Bool, Bool)`, with the effect of giving `dup` the combined intersection type `(Int → (Int, Int)) ∧ (Bool → (Bool, Bool))`. By contrast, in SuperF, we only assign `dup` the more general type `∀ a. a → (a, a)`, which is fine to do in output position as this makes strictly *more* programs type check than picking a specific intersection.[5]

Similarly, SuperF does not support unions in negative positions, though these can be used to represent a structural form of algebraic data types [Castagna et al. 2016; Parreaux and Chau 2022].

Polarized unions & intersection desugaring. When unions are restricted to positive positions and intersections to negative ones, subtype inference remains simple and tractable [Dolan and Mycroft 2017]. In fact, this restricted use of unions and intersections turn out to be equivalent [Parreaux 2020] to a form of bounded polymorphism with both upper and lower bounds on type variables (*multi-bounded polymorphism*). Consider `baz` below and two *equivalent* types for it:

```
bar f x = if f x then f else fun x → x
-- using unions and intersections:
bar : ∀ a b. (a ∧ (b → Bool)) → b → (a ∨ ∀ d. d → d)              --  (A)
-- using multi-bounded polymorphism:
bar : ∀ a b c {a ≤ b → Bool, c ≥ a, c ≥ ∀ d. d → d}. a → b → c    --  (B)
```

Proof. To show that `(A)` and `(B)` are equivalent, we show subtyping in both directions. To show subtyping between two polymorphic types, we: (1) assume the right-hand side type variable bounds; (2) instantiate the type variables on the left-hand side type to types that satisfy the left-hand side type variable bounds; and (3) show subtyping between the underlying bodies of the two polymorphic types. In our case, to show `(A) ≤ (B)`, we pick `a = a` and `b = b` and thus have to show `(a ∧ (b → Bool)) → b → (a ∨ ∀ d. d → d) ≤ a → b → c` assuming `a ≤ b → Bool`, `c ≥ a`, and `c ≥ ∀ d. d → d`, which decomposes by function co- and contravariance to: • `a ≤ a ∧ (b → Bool)`, i.e., ○ `a ≤ a`

[5] Note that not all intersection-overloaded function type signatures can be represented using pure parametric polymorphism. For instance, in TypeScript it is possible to define functions with types such as `(Int → Bool) ∧ (Bool → Int)`.

(immediate); and ∘ `a` ≤ `b` → `Bool` (by assumption); • `b` ≤ `b` (immediate) • `a` ∨ ∀ `d. d` → `d` ≤ `c`, i.e., ∘ `a` ≤ `c` (by assumption); and ∘ ∀ `d. d` → `d` ≤ `c` (by assumption). To show the other direction `(B)` ≤ `(A)`, we pick `a = a` ∧ `(b` → `Bool)`, `b = b`, and `c = a` ∨ ∀ `d. d` → `d`, which satisfies all the left-hand side type variable bounds and syntactically results in the desired type. □

Using bounds internally is more convenient than directly using unions and intersections because (1) it facilitates both the implementation and formal specification of type inference; and (2) it allows avoiding some repetition in inferred types, as identical requirements attached to different occurrences of a type variables can be shared as part of a single bound. (For example, ∀ `c {c` ≤ ∀ `a. a` → `a}. (c, c)` → `c` is more concise than the equivalent ∀ `c. (c` ∧ `(`∀ `a. a` → `a), c` ∧ `(`∀ `a. a` → `a))` → `c`.) So in this paper we focus on *pure* multi-bounded polymorphism and regard unions and intersections as *syntactic sugar* for the former. Nevertheless, it is often useful to *think of* and *describe* bounded polymorphic types in terms of unions and intersections, which are often more intuitive.

Expressiveness. Extending System F with multi-bounded polymorphism is not insignificant and does bring additional expressiveness to the language. Appendix A.2 presents an example program which is not typeable in System F but is typeable in F_ω, and for which SuperF infers a principal type without the help of any annotations. Note that SuperF is not limited in the ranks of the polymorphic types it infers. For instance, Appendix A.3 exemplifies SuperF type inference for rank 3.

2.4 Type Inference Approach

We now describe type inference in SuperF.

Subtyping over unification. It was historically recognized that subtype inference is notoriously harder than traditional unification-based type inference as in ML. Therefore, it was natural to expect that the difficulty of subtype inference would *compound* with that of first-class-polymorphic type inference, which is probably why the vast majority of previous work stuck with unification — indeed, even the venerable ML^F system, which features *lower-bounded* polymorphism and therefore has a core notion of subtyping (based on generic instances), uses first-order *unification* as the workhorse of its type inference engine. We believe that this is fundamentally a mistake, and that subtype inference in fact *alleviates* most of the intricacies of unification-based approaches to first-class-polymorphic type inference. Indeed, subtyping is better at tracking the *flow of values* through a program, which becomes important in this context.[6] We argue that ML can get away with approximating data flows in a way that creates undue relationships between unrelated types because its core type system is truly simplistic (essentially simply-typed lambda calculus). But as soon as we move to the first-class-polymorphic setting, such approximations induce all sorts of problems and situations where one needs to make premature polymorphism choices.

Main ideas of SuperF type inference. Generally, our main ideas are as follows:

- Never *infer* parametrically-polymorphic types in negative position, since it is often enough to infer intersection types there.[7]
- Segregate the syntax of types between positive and negative types, similar to the approaches of Dolan and Mycroft [2017]; Pottier [1998], among others.
- Let users provide type annotations that include System F-style polymorphic types, and make sure we can *check* our inferred types *against* these types.

[6]The usefulness of combining subtype-based flow analysis with polymorphism was already noticed by the turn of the century in the context of control-flow analysis, e.g., by Faxén [1997]; Rehof and Fähndrich [2001]; Smith and Wang [2000].
[7]This insight was also leveraged in earlier work by Jim [2000], in a different type inference setting.

- Only infer polymorphic types for lambda abstractions, as we are mostly interested in polymorphic functions, and subtyping with distributivity usually alleviates the need for other forms of polymorphic values in a call-by-value system.[8] (This point is minor and could easily be relaxed.)
- Offload generated subtyping constraints to a dedicated subtype constraint solver (described in Section 2.5); solve these constraints down to type variable bounds upon generalizing the types of lambda expressions.

The approach described above essentially allows us to infer types for many well-typed System F terms commonly encountered in functional programs, as well as for some terms that are well-typed in our system but ill-typed in System F.

Checking restricted user annotations. While SuperF internally supports multi-bounded polymorphism in positive positions, which allows type inference to use subtyping to delay polymorphism choices, we believe most programmers rarely think in terms of such bounded types. To make checking programmer type annotations against inferred types practical, we *restrict the language of type annotations* to System F types, meaning that users may specify type signatures containing first-class-polymorphic types, but these may *not* specify bounds. Indeed, if we allowed bounds on user-provided type signatures, we would encounter much more difficult constraints to solve — e.g., consider `foo (add : (Int → Int) ∧ (Str → Str)) : τ = ...` where the type annotations are equivalent to giving `foo` type `∀ a {a ≤ Int → Int, a ≤ Str → Str}. a → τ`. While type checking the body of `foo` and its uses of the `add` parameter, we would need to deal with constraints of the form `a ≤ σ` where `a` is a rigid type variable with upper bounds; which is equally as hard as dealing with overloading constrains like `(Int → Int) ∧ (Str → Str) ≤ σ`. We are not aware of any constraint-solving approaches able to deal with such types without having to either approximate them[9] or rely on backtracking, which we argue makes type inference impractical in a real-world context.

Subtype Extrusion. A major difficulty of inferring subtypes in the context of *nested* polymorphism — whether first-class or ML-style — is to avoid mixing up type variables coming from different polymorphism levels. For example, consider typing an expression of the form `fun x → E (fun y → x (y, y))` or equivalently `fun x → let f y = x (y, y) in E f`, where E is some arbitrary subexpression. The variable `y` introduced by the inner argument function will have type β, for some fresh type variable β, which will be generalized locally as part of the overall type of the inner function, yielding a type of the form $\forall\beta \ldots \{\ldots\}.\, \beta \to \tau$, assuming `x (y, y)` returns type τ. Because β is generalized in this nested local function, it is *more polymorphic* than the type of `x` (call it α), so leaking β into a bound of α as in $\alpha \le (\beta, \beta) \to \tau$ would be unsound. To see why, consider that β may later be *instantiated* into several unrelated types as part of its use in the body of E, by which point we would lose the connection between these specific instances and α.

We solve this problem by *extruding* types that are *too polymorphic*, like (β, β), into some less polymorphic types like (β', β'), where β' is added to the same quantifier as α. Crucially, the extrusion is made to be an *approximation* of the original type by adding the bound $\beta \le \beta'$ onto the original β.[10] This way, whenever β is later instantiated in E — say, successively to Int and Bool — then by transitivity we will get constraints $\mathsf{Int} \le \beta'$ and $\mathsf{Bool} \le \beta'$, which will ensure that

[8] An example program that is rejected because we only generalize lambdas is given in a Section 2.7 footnote.

[9] MLsub by Dolan and Mycroft [2017] achieves subsumption checking in the presence of such constraints by *under-approximating intersection types*, so for example it treats $(\tau_1 \to \tau_2) \wedge (\sigma_1 \to \sigma_2)$ as equivalent to $(\tau_1 \vee \sigma_1) \to (\tau_2 \wedge \sigma_2)$. This simplifying assumption is *unsound* in the presence of first-class polymorphism, so their technique cannot be used in SuperF. For instance, consider that `id` can be typed as $\forall\alpha.\alpha \to \alpha$, which is a subtype of $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Str} \to \mathsf{Str})$, which according to MLsub is a subtype of $(\mathsf{Int} \vee \mathsf{Str}) \to (\mathsf{Int} \wedge \mathsf{Str})$ – but this type is clearly not a valid type for `id`.

[10] If β also occurred negatively in the extruded type, we would also add a *lower* approximant β'' where $\beta'' \le \beta$ and $\beta'' \le \beta'$.

the final type inferred for x will be essentially $(\mathsf{Int} \vee \mathsf{Bool},\ \mathsf{Int} \vee \mathsf{Bool}) \to \tau$, the type of a function that accounts for the fact that *both* integers *and* booleans may flow into its parameters.

The original idea of *subtype extrusion* is due to Parreaux [2020], who informally presented a level-based algorithm for subtype inference with nested polymorphism. This approach was itself inspired by the older level-based unification algorithm implemented for the Caml and later OCaml compilers [Kiselyov 2013; Pottier and Rémy 2005], which finds its origin in the work of Rémy [1992], who initially called these polymorphism levels *ranks*, or "degrés" in French [Rémy 1990]. In this paper, we formalize a simplified version of Parreaux's extrusion algorithm, which does not use explicit levels but can be implemented with levels as an optimization (our SuperF implementation does so).

2.5 Subtype Constraint Solving

The constraint-solving approach of SuperF is relatively straightforward, which we consider to be one of the main contributions of this paper.

First, we decompose concrete type constraints following the structure of types, for example decomposing $(\tau_1, \tau_2) \le (\sigma_1, \sigma_2)$ into $\tau_1 \le \sigma_1$ and $\tau_2 \le \sigma_2$, and decomposing $\tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2$, by following function parameter contravariance and function result covariance, into $\sigma_1 \le \tau_1$ and $\tau_2 \le \sigma_2$. At the same time, we compute the transitive closure of type variable bounds, meaning that when constraining $\alpha \le \tau$ we also constrain $\sigma \le \tau$ for each existing lower bound σ of α, and symmetrically for $\tau' \le \alpha$ and upper bounds. For polymorphic types appearing on the left, of the form '$\forall \overline{\alpha}\{\overline{B}\}.\ \tau \le \sigma$', all we need to do is (1) instantiate all quantified type variables $\overline{\alpha}$ to fresh type variables $\overline{\alpha'}$ in τ and B; (2) solve the bounds $[\alpha \mapsto \alpha']\overline{B}$ as constraints to make sure they hold; and (3) continue by constraining $[\alpha \mapsto \alpha']\tau \le \sigma$.

Type Avoidance. when polymorphic types are found on the *right*, as in $\tau \le \forall \alpha.\, \sigma$, we need to turn the α type variable into a so-called *skolem*, i.e., a type variable on which no bounds can be added (because we are not allowed to assume anything about this type). We continue by constraining $\tau \le \sigma$, during which we may end up trying to add to existing (outer) type variables bounds that refer to this α skolem, which makes no sense outside of the corresponding polymorphic type. Therefore, we need to *approximate* these bounds until they no longer refer to α. A simple solution, which we settle on in this paper, is to *widen* all positive occurrences of α to $\top$ and to *narrow* all its negative occurrences to $\bot$. As a concrete example, consider passing some argument of type $\alpha \to \beta$ to a function of type $(\forall \gamma.\ \gamma \to \gamma) \to \tau$, where α and β are some type variables coming from the outside. This leads to $\alpha \to \beta \le (\forall \gamma.\ \gamma \to \gamma)$ and to constraining $\gamma_{\text{sk.}} \le \alpha$ and $\beta \le \gamma_{\text{sk.}}$ where $\gamma_{\text{sk.}}$ is the skolem, resulting in overall inferred bounds $\{\top \le \alpha,\ \beta \le \bot\}$. The argument function, whose type must subtype $\alpha \to \beta$, would thus essentially be restricted to non-terminating computations (of type $\bot$) or delayed non-terminating computations like `fun _ → failwith "oops"` (of type $\top \to \bot$).

Delaying instantiation. Instantiating polymorphic types found on the left *too early* and those found on the right *too late* may lead to unnecessary failures [Zhao et al. 2019]. For example, in $\forall \alpha.\, \alpha \to \alpha \le \forall \beta.\, \beta \to \beta$, if we instantiate the left-hand-side first, we proceed to constraining $\alpha \to \alpha \le \beta_{\text{sk.}} \to \beta_{\text{sk.}}$, which leads to bounds $\beta_{\text{sk.}} \le \alpha \le \beta_{\text{sk.}}$, and we end up with type-avoided bounds $\top \le \alpha \le \bot$ on α, which are inconsistent (thus failing the process). But instantiating the right-hand-side first works out because since α is then a *locally-instantiated variable*, it is *allowed* to refer to $\beta_{\text{sk.}}$, and we can forget about both α and $\beta_{\text{sk.}}$ after the constraining is done — that is, in general, unless α leaked into an outer constraint while constraining, in which case we would have to keep an extruded version of it around, which would fail for the same reason as above.

Distributivity. It is sometimes possible to either *delay* the instantiation or *rush* the skolemization of polymorphic types by using the *distributivity* property, which states that $\forall \alpha.\, \tau \to \sigma$ is *equivalent*

by subtyping to $\tau \to \forall\alpha.\,\sigma$ when α does not occur in τ (this is called *deep skolemization* by Peyton Jones et al. [2007]). For example, to successfully constrain $\forall\alpha.\,\mathsf{Int} \to \alpha \to \alpha \le \mathsf{Int} \to \forall\beta.\,\beta \to \beta$, we can either distribute the left-hand side, resulting in the equivalent constraint $\mathsf{Int} \to \forall\alpha.\,\alpha \to \alpha \le \mathsf{Int} \to \forall\beta.\,\beta \to \beta$, which succeeds, or distribute the right-hand side, resulting in $\forall\alpha.\,\mathsf{Int} \to \alpha \to \alpha \le \forall\beta.\,\mathsf{Int} \to \beta \to \beta$, which also succeeds.[11] This is because in both cases, after distributing, we are able to skolemize the right-hand side before instantiating the left-hand side. Therefore, we try to make use of distributivity whenever possible in SuperF.

Cycle checking. Our implementation does not support recursive types, so it rejects cyclic constraints such as $\alpha \le \mathsf{Int} \to \alpha$, using a form of subtyping-aware "occurs-check". This check prevents typing some programs that could have meaningful types even without recursive types, such as `let rec f a = f`, which can be typed as $\top \to \top$, as well as $\top \to \top \to \top$, as well as $\top \to \top \to \top \to \top$, etc. and in general $\mu\alpha.\,(\top \to \alpha)$ where μ is the recursive type binder. However, the loss of expressiveness is not very significant; indeed, ML programmers are already used to "ill-formed" recursive definitions like these being rejected, and their use is of limited interest when more usual forms of type-level recursion can be achieved through standard data type definitions.

Unfortunately, even when we reject cyclic constraints, the subtype constraining process described thus far may still not terminate on inputs involving *indirect* forms of recursion, such as the standard $\Omega = (\lambda x.\,x\;x)\;(\lambda x.\,x\;x)$ term and other recursion combinators like $\mathrm{Y}\,f = (\lambda x.\,f\;(x\;x))\;(\lambda x.\,f\;(x\;x))$. Therefore, we propose the use of a *heuristic* which we refer to as the "*suspiciously recursive-looking criterion*" (SRLC). Constraints that fail the SRLC are simply rejected conservatively, terminating type inference with a failure.

2.6 Suspiciously Recursive-Looking Criterion (SRLC)

The core challenge for ensuring the termination of type inference is that constraint solving may end up comparing an infinite number of distinct types, which are generated by instantiating and skolemizing universal types, as well as copying types during subtype extrusion.

Going back to the roots. Our idea is to assign what we call a "*root*" to each type and to make sure that all types ever created during constraint solving only have a finite number of roots. We abort constraint solving with an error upon comparing a pair of types with the same roots as another pair of types currently being compared, which effectively bounds the depth of recursive constraining calls. The root of a type variable α is:

- α itself if α already existed at the start of the constraint-solving run;
- β if α was created by instantiating, skolemizing, or extruding a type variable with root β during the current constraint-solving run.

The roots of other types are formed by substituting all type variables with their respective roots.

Termination argument. Because the only way to create new types during constraint solving is to substitute the type variables of existing types with fresh type variables (which by construction yields a type with the same root as the original) or with $\top$ and $\bot$, it is clear that the number of roots we will ever reach is finite, ensuring the termination of constraint solving.

Practicality. We found that in practice, the SLRC seldom gets in the way of typing correct terms, the only exception being terms that make use of self-application (such as G9 in Table 3, presented in Section 5.5). Crucially, we assume monomorphic recursion as a primitive of the language, meaning that we type recursive definitions by first assigning them a type variable and then constraining

[11] Note that we do not always have a choice about which side to distribute, unlike here.

that type variable to be a supertype of the inferred result type.[12] In this context, the programs the SRLC rules out are mostly ones that use indirect recursion (in the style of the Y combinator).

2.7 Expressiveness and Limitations

SuperF is uniquely expressive, but it does naturally have limitations.

Expressiveness. HML [Leijen 2009] is a restriction of ML^F where type annotations are restricted to System F types and where all polymorphic parameters must be annotated (even when not used polymorphically within the function body). There are small differences between HML/ML^F and SuperF in the places where generalization occurs,[13] which are usually smoothed over by distributivity. Modulo these differences, we conjecture that SuperF subsumes HML, which in turn subsumes ML type inference. So SuperF subsumes ML type inference as well, and type annotations are not needed by SuperF in programs where ML type checking succeeds. While we have not yet proved it, we experimentally verified this claim by porting OCaml's `List` module to our SuperF implementation, requiring no type annotations (see Section 5.3). ML^F (even in its "shallow" variant [Le Botlan and Rémy 2009]) allows unrestricted type annotations and can therefore accept programs with annotated lower bounds that are not *syntactically* valid in SuperF. Conversely, SuperF accepts programs that do not type check in ML^F. Therefore, SuperF and ML^F are technically incomparable in terms of expressiveness. However, we conjecture that a syntactic restriction of ML^F source programs to using only System F type annotations would be subsumed by SuperF even without the additional polymorphism restriction of HML. Since functional programmers do not normally write type annotations with lower bounds (we are not aware of any functional programs using such annotations in the wild), one could argue that SuperF is more expressive than ML^F in practice.

Limitation: needed annotations. While it is enough to infer non-parametric intersection types in negative positions in many cases, it is not enough when a *parametrically*-polymorphic input type is needed. For instance, the following refactored version of `foo` no longer admits a precise type:

```
fooLet f = let g x = f x in (g 123, g True)
fooLet : ∀ a. ((Int ∨ Bool) → a) → (a, a)
```

This is because `fooLet` makes both arguments to `f` pass through the bottleneck of `g`'s `x` parameter, thereby merging their originally-separate data flows. It is not possible to recover the precise type of `foo` through type annotations, even in the declarative system.[14] On the other hand, type annotations *can* be used if a different type is desired, for example annotating `(f : ∀ a. a → a)`.

Limitation: restricted type syntax. As explained in the introduction, SuperF is atypical in that it infers types that cannot always be written down (inferred types are generally *non-denotable*) due to the polarity restriction of its internal type syntax. Users may only provide System F type signatures, in which polymorphic types do not have bounds, so that for example the principal type of `foo` cannot be used as an explicit type signature. However, any well-typed SuperF term admits a number of possible System F types. These types are often less precise than the SuperF type, and there is often not a best one to pick, but since it is the user who is providing the type signature, this is not a problem in practice. It appears that users seldom want to use bounds in their annotations anyway (indeed, much of the work following up on MLF was based on this very assumption). Moreover, in a typical program, a large number of functions, if not most of them, will not be associated with an explicit type signatures. For instance, even tough the standard practice in Haskell is to annotate exported top-level definitions with type signatures, typical Haskell

[12] Of course, we can also type check polymorphically-recursive functions as long as explicit type signatures are provided.
[13] For instance, `flip const 42` is generalized (to `∀ 'a . 'a → 'a`) in HML but not in SuperF, as it is not syntactically a function.
[14] Note that we *cannot* type `g` at `(Int → a) ∧ (Bool → b)`, the type of `f` in the original `foo`, because $F_{\{\leq\}}$ and a fortiori SuperF deliberately lack an intersection-introduction typing rule (which would make type inference intractable).

code contains many unannotated helper functions, notably in `where` clauses and in non-exported definitions, as well as lambda expressions passed as arguments to other functions or stored in data structures. When working with such local definitions, the full power of SuperF type inference is available, and previously-untypeable functions like `foo` can now be used. Therefore, our restricted type syntax should not be thought of chiefly as a limitation: the SuperF algorithm it enables is a strict improvement over most existing approaches for inferring System F-style first-class polymorphism.

2.8 Practical Considerations: Stability and Error Messages

We now briefly explain why we think SuperF is a useful and practical type inference system.

Robustness and Stability. An important design consideration for first-class-polymorphic type inference is *robustness* against innocuous changes to the program, i.e., type inference *stability* [Le Botlan and Rémy 2009, §4.5]. Fragile type inference leads to suboptimal user experience as it becomes harder for programmers to predict when type annotations will be needed. Thanks to subtyping, which faithfully encodes program data flows without having to approximate types through unification, SuperF achieves a high degree of stability. However, just like for expressiveness, SuperF is neither more nor less stable than MLF in general. Table 1 summarizes some of the important stability properties of ML^F and SuperF. While inferring precise data flow types generally improves stability, it also makes the system *too expressive* for supporting stability properties that more rigid systems like ML^F possess, as exemplified in M1. A counterexample showing that M1 does not hold in SuperF is the `fooLet` function mentioned in Section 2.7, whose let-reduced form has the same (more general) type as `foo`. The reason this is not a problem in ML^F is that ML^F is able to type check neither `foo` nor `fooLet`, nor any similar functions where unannotated parameters are used at distinct types in the function's body. We conjecture that M1 holds in SuperF if we make it a side condition that `x` be used *linearly* in `a2`, though proving this is quite challenging and we have not done so yet. Note that the side condition of MS5 is crucial, as neither ML^F nor SuperF are stable under general η expansion. Here, "`f` has function type" should be understood in a restrictive, syntactic sense: for instance, neither a type variable with function type bounds nor a polymorphic type with a function type body is a *function type* as required by the condition. The side condition that `x` should not appear under lambdas in S3 can be intuitively understood as follows: lambdas introduce nested polymorphic scopes, so for `x` to interact with the corresponding nested polymorphic type variables without leaking them and causing imprecision, `x` may need to be parametrically polymorphic, but we do not infer parametrically polymorphic types for function parameters.

Table 1. Example *typeability equivalences* demonstrating some stability properties of ML^F and SuperF. In each case, the left-hand side is typeable *if and only if* the right-hand side is, with equivalent types. Rows labeled with "MS$*$" show properties shared by both systems while those labeled with "M$*$" show properties enjoyed by ML^F but not SuperF and those labeled with "S$*$" show properties enjoyed by SuperF but not ML^F. Metavariables s, t, f, g range over terms while x, y range over variables.

M1	Let conversion	**let** x = s **in** t	$[x \mapsto s]t$	x occurs in t
MS2	Redefine application	$s\ t$	(**fun** $f\ x \to f\ x$) $s\ t$	
MS3	Reorder arguments	$s\ t_1\ t_2$	(**fun** $x\ y \to s\ y\ x$) $t_2\ t_1$	
MS4	Curryfication	$s\ (t_1, t_2)$	(**fun** $x\ y \to s\ (x, y)$) $t_1\ t_2$	
MS5	η conversion	f	**fun** $x \to f\ x$	f has function type
S1	Tail conversion	(**if** t **then** f **else** g) x	**if** t **then** $f\ x$ **else** $g\ x$	
S2	Head conversion	x (**if** s **then** t_1 **else** t_2)	**if** s **then** $x\ t_1$ **else** $x\ t_2$	
S3	Rename argument	(**fun** $x \to t$) y	$[x \mapsto y]t$	x does not occur under lambdas in t

Core syntax

$$\begin{array}{lrcl} \textit{Type} & \tau, \sigma & ::= & \alpha \mid \tau \to \tau \mid \forall V\{\Sigma\}.\, \tau \mid \langle \tau \rangle \mid \boxed{\tau}_x \mid \top \\ \textit{Bound} & B & ::= & \alpha \le \tau \mid \tau \le \alpha \\ \textit{Term} & s, t & ::= & x \mid \lambda x.\, t \mid t\ t \mid (\,) \mid t : \tau \mid \textbf{let}\ x = t\ \textbf{in}\ t \end{array}$$

Contexts

$$\begin{array}{lrcl} \textit{Typing ctx.} & \Gamma & ::= & \epsilon \mid \Gamma \cdot (x : \tau) \mid \Gamma \cdot B \\ \textit{Bounds ctx.} & \Xi, \Sigma & ::= & \epsilon \mid \Xi \cdot B \\ \textit{Variables} & V, W, X & ::= & \epsilon \mid V \cdot \alpha \end{array}$$

Fig. 1. Syntax of types, terms, and contexts.

User-friendly type error messages. A non-obvious advantage of type inference systems like SuperF based on subtyping (in the school of *algebraic subtyping* [Dolan 2017]) rather than unification is that they are easily adapted to track and propagate *type provenance information* relating all inferred types to the relevant source code locations from which these types arose. This information can in turn be used to produce precise and informative type error messages [Bhanuka et al. 2023]. In the first-class polymorphism setting, one has to additionally take care of the interaction between rigid variables and skolems, which can leak and cause problems down the line. SuperF is well-equipped to deal with these problems thanks to its fine-grained tracking of data flows.

3 DECLARATIVE TYPE SYSTEM

We now present the declarative $F_{\{\le\}}$ type system formally.

The syntax of $F_{\{\le\}}$ is presented in Figure 1. The only non-standard syntax is that of polymorphic types $\forall V\{\Sigma\}.\, \tau$, which include a set of bounds Σ on the quantified variables V, and the *box* $\boxed{\tau}_x$ and *unbox* $\langle \tau \rangle$ forms, which are explained next. For simplicity of presentation, we do not have special forms for concrete features such as product and sum types, as their addition is straightforward; the bare-bone features presented here are sufficient to demonstrate our main ideas.

Notations and Shorthands. Notation $\overline{E_i}^{\,i}$ denotes a repetition of $i = 0$ to n occurrences of E_i, and we omit i when it is unambiguous. We also make use of the following notations:

$$V \notin X \triangleq \overline{\alpha \notin X}^{\alpha \in V};\quad \mathcal{B}(\Gamma) \triangleq \overline{B}^{B \in \Gamma};\quad \bot \triangleq \forall \alpha.\, \alpha;\quad \tau_1 \vee \tau_2 \triangleq \bigvee\{\tau_1, \tau_2\};\quad \bigvee\{\overline{\tau_i}^{\,i}\} \triangleq$$

$$\forall \alpha\{\overline{\tau_i \le \alpha}^{\,i}\}.\, \alpha \quad (\alpha\ \textbf{\textit{fresh}})$$

3.1 Declarative Typing Rules

System $F_{\{\le\}}$ is a mostly standard polymorphic type system whose main distinctive features are *multi-bounded polymorphic types* and *boxed types*. The latter prevents unannotated function parameters from being used polymorphically, which reflects the limitations of the SuperF algorithm.

The declarative typing rules of $F_{\{\le\}}$ are presented in Figure 2. Rules T-Unit, T-Var, T-App, T-Subs, and T-Let are all standard. T-Abs extends the typing context with a *boxed* type $\boxed{\tau}_x$ for its parameter x, which prevents the parameter from being used polymorphically without an annotation, as we shall see next. T-Unbox allows erasing leftover boxes whose variables are no longer in scope, using the box erasure syntax $\tau \setminus x$ formally defined and exemplified in Appendix B.1. T-Asc expects the annotated expression to have type $\langle \tau \rangle$, which is a type used to *unbox* polymorphic types, making them available for use. Rule T-Forall allows *generalizing* the type τ of an arbitrary term t.[15] There are two requirements for generalization: first, the free type variables of τ must not occur in Γ, which is standard; second, we require that the quantification $\forall V\{\Sigma\}$ be *consistent* in Γ, written $\mathcal{B}(\Gamma) \vdash \forall V\{\Sigma\}$ ***cons.*** and read "in context Γ, the quantification of variables V with bounds Σ is consistent". We define consistency as the existence of a solution to Σ, i.e. a substitution of V such that the variable bounds Σ hold in $\mathcal{B}(\Gamma)$. This makes sure no parts of a typing derivation

[15] While SuperF only generalizes lambda expressions, it is simpler to present generalization as an orthogonal rule in $F_{\{\le\}}$.

$\boxed{\Gamma \vdash t : \tau}$

$$\textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \top} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{T-Abs}\ \frac{\Gamma \cdot \left(x : \boxed{\tau_1}_x\right) \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2} \qquad \textsc{T-Unbox}\ \frac{\Gamma \vdash t : \tau \quad x \notin dom(\Gamma)}{\Gamma \vdash t : (\tau \setminus x)}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash t_0 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash t_0\ t_1 : \tau_2} \qquad \textsc{T-Asc}\ \frac{\Gamma \vdash t : \langle \tau \rangle}{\Gamma \vdash (t : \tau) : \tau} \qquad \textsc{T-Subs}\ \frac{\Gamma \vdash t : \tau_1 \quad \mathcal{B}(\Gamma) \vdash \tau_1 \leq \tau_2}{\Gamma \vdash t : \tau_2}$$

$$\textsc{T-Forall}\ \frac{\Gamma \cdot \Sigma \vdash t : \tau \quad V \notin FV(\Gamma) \quad \mathcal{B}(\Gamma) \vdash \forall V\{\Sigma\}\ \textbf{\textit{cons.}}}{\Gamma \vdash t : \forall V\{\Sigma\}.\, \tau} \qquad \textsc{T-Let}\ \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \cdot (x : \tau_1) \vdash t_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : \tau_2}$$

Fig. 2. Declarative typing rules.

$\boxed{\Xi \vdash \tau \leq \tau}$

$$\textsc{S-Top}\ \frac{}{\Xi \vdash \tau \leq \top} \qquad \textsc{S-VarRefl}\ \frac{}{\Xi \vdash \alpha \leq \alpha} \qquad \textsc{S-Hyp}\ \frac{B \in \Xi}{\Xi \vdash B} \qquad \textsc{S-Unbox1}\ \frac{\Xi \vdash \tau \leq \sigma}{\Xi \vdash \boxed{\tau}_x \leq \langle \sigma \rangle} \qquad \textsc{S-Unbox2}\ \frac{}{\Xi \vdash \tau \leq \langle \tau \rangle} \qquad \textsc{S-CongBound}\ \frac{\Xi \vdash \alpha \leq \tau}{\Xi \vdash \boxed{\alpha}_x \leq \boxed{\tau}_x}$$

$$\textsc{S-Trans}\ \frac{\Xi \vdash \tau_0 \leq \tau_1 \quad \Xi \vdash \tau_1 \leq \tau_2}{\Xi \vdash \tau_0 \leq \tau_2} \qquad \textsc{S-Fun}\ \frac{\Xi \vdash \tau_0 \leq \tau_1 \quad \Xi \vdash \tau_2 \leq \tau_3}{\Xi \vdash \tau_1 \to \tau_2 \leq \tau_0 \to \tau_3} \qquad \textsc{S-CongFun}\ \frac{}{\Xi \vdash \boxed{\tau_1 \to \tau_2}_x \leq \tau_1 \to \boxed{\tau_2}_x} \qquad \textsc{S-Forall-R}\ \frac{V \notin FV(\tau) \quad \Xi \vdash \forall V\{\Sigma\}\ \textbf{\textit{cons.}}}{\Xi \vdash \tau \leq \forall V\{\Sigma\}.\, \tau}$$

$$\textsc{S-Forall-Cov}\ \frac{\Xi \vdash \forall V\{\Sigma\}\ \textbf{\textit{cons.}} \quad \Xi \cdot \Sigma \vdash \tau \leq \sigma}{\Xi \vdash \forall V\{\Sigma\}.\, \tau \leq \forall V\{\Sigma\}.\, \sigma} \qquad \textsc{S-Forall-L}\ \frac{\Xi \vdash [\overline{\alpha_i \mapsto \tau_i}^{\,i}]\Sigma}{\Xi \vdash \forall \overline{\alpha_i}^{\,i}\{\Sigma\}.\, \tau \leq [\overline{\alpha_i \mapsto \tau_i}^{\,i}]\tau} \qquad \textsc{S-Forall-Distr}\ \frac{\Xi \vdash \forall V\{\Sigma\}\ \textbf{\textit{cons.}} \quad V \notin FV(\tau_1)}{\Xi \vdash \forall V\{\Sigma\}.\, \tau_1 \to \tau_2 \leq \tau_1 \to \forall V\{\Sigma\}.\, \tau_2}$$

Fig. 3. Declarative subtyping rules.

can rely on unsound assumptions. In particular, subtyping assumptions cannot be used to carry subtyping proofs in the same way as dependent type systems and GADTs can be used to carry type equality or subtyping proofs [Boruch-Gruszecki et al. 2022; Parreaux et al. 2019; Scherer and Rémy 2015]. We disallow inconsistent constraints because these could be used to make the type system accept obviously ill-typed terms. For example, $\forall\alpha\{\mathsf{Nat} \leq \alpha \leq \mathsf{Int}\}.\, \alpha \to \alpha$ (a shorthand for $\forall\alpha\{\mathsf{Nat} \leq \alpha, \alpha \leq \mathsf{Int}\}.\, \alpha \to \alpha$) is consistent because $\mathsf{Nat} \leq \mathsf{Int}$. It defines an identity type that works on any type between Nat and Int. On the other hand, $\forall\alpha\beta\{\top \leq \alpha \leq \beta, \beta \leq \mathsf{Int}\}.\, \alpha \to \beta$ is *inconsistent* because the (transitive) bounding on α and β, $\top \leq \mathsf{Int}$, cannot be derived.

3.2 Declarative Subtyping Rules

The subtyping rules of $\mathrm{F}_{\{<:\}}$ are presented in Figure 3. Rules S-Top, S-VarRefl, S-Trans, and S-Fun are standard. S-Hyp is used to leverage any type variable bounds present in the typing context. S-Forall-R *introduces* polymorphic types in the middle of subtyping. Its premise makes sure that the added quantified type variables are *not* in the type being quantified — the idea is that these type variables will appear later through the combined use of S-Forall-Cov,[16] to widen a type under quantifiers, and S-Forall-L, to instantiate the type variables of a polymorphic type. This way, we can use the subtyping rules to rearrange the quantifiers of a type — for example, we can

[16] S-Forall-Cov should not be confused with "kernel" rules in the context of explicit polymorphism *à la* System $\mathrm{F}_{<:}$. Unlike these systems, $\mathrm{F}_{\{<:\}}$ allows *polymorphic subtyping*, so that relationships like $\forall\alpha\{\alpha \leq \mathsf{Int}\}.\, \tau \leq \forall\alpha\{\alpha \leq \mathsf{Nat}\}.\, \tau$ do hold.

derive subtyping relationships such as $\forall\alpha\gamma.\forall\beta\{\beta \le \alpha\}.\, \alpha \to \beta \to \gamma \le \forall\beta\alpha\{\beta \le \alpha\}.\, \alpha \to \beta \to \top$. S-Forall-Distr describes the distributivity of polymorphic types over arrow types; note that the other direction is already admissible by covariance of function results and uses of S-Forall-R/S-Forall-Cov/S-Forall-L. All these rules require that the corresponding quantifications be consistent, to avoid introducing bad types during subtyping. Finally, S-Unbox1 and S-Unbox2 are the rules that make type ascription work. The latter is used when the ascribed type needs not be unboxed, and the former is used to open a boxed function parameter type. Note that boxes only block polymorphic types and they congrue with type variables and functions:[17] S-CongBound allows *upcasting* a boxed type variable to one of its bounds and S-CongFun allows pushing a box into the result type.

Derived lattice types. As hinted previously, we can *encode* bottom and union types.

Theorem 3.1. *$\bot \triangleq \forall\alpha.\, \alpha$ is the bottom of $(\le)$, i.e., for all τ **wf**, we have $\bot \le \tau$.*

Theorem 3.2. *For all $\overline{\tau_i \ \textbf{wf}}^i$ where $\alpha \notin FV(\overline{\tau_i}^i)$, the type defined as $\bigvee\{\overline{\tau_i}^i\} \triangleq \forall\alpha\{\overline{\tau_i \le \alpha}^i\}.\, \alpha$ is the least upper bound of all $\overline{\tau_i}^i$, i.e.,*

- *(A) $\bigvee\{\overline{\tau_i}^i\}$ is an upper bound of $\overline{\tau_i}^i$, meaning that for all j we have $\tau_j \le \bigvee\{\overline{\tau_i}^i\}$; and*
- *(B) for all σ **wf** such that σ is an upper bound of $\overline{\tau_i}^i$, we have $\bigvee\{\overline{\tau_i}^i\} \le \sigma$.*

Remark 1. By contrast, we cannot encode intersections in the same way as unions, because that would require the use of existential quantification, which is not supported by $F_{\{\le\}}$ nor by the underlying System F_{cc}. On the other hand, using universal quantification, we can still encode intersections that occur *negatively* in some outer type, i.e., if α occurs negatively in $\tau[\alpha]$, then $\tau[\sigma_0 \wedge \sigma_1]$ can be encoded as $\forall\alpha\{\alpha \le \sigma_0,\, \alpha \le \sigma_1\}.\, \tau[\alpha]$, where $\alpha \notin FV(\tau) \cup BV(\tau[\cdot])$.

Example 3.3. Consider term $t = \lambda x.$ **let** $y = (\mathit{choose}\ x\ (\lambda z.\, z) : \forall\alpha.\, \alpha \to \alpha)$ **in** $(y\ 0,\, y\ \textbf{True})$ in context $\Xi = (\mathit{choose} : \forall\beta.\, \beta \to \beta \to \beta)$. As expected intuitively, t can be typed in $F_{\{\le\}}$ at type $(\forall\alpha.\, \alpha \to \alpha) \to (\mathsf{Int}, \mathsf{Bool})$. First note that $\lambda z.\, z$ can be typed at $\forall\alpha.\, \alpha \to \alpha$. We can also assign to parameter x the type $\forall\alpha.\, \alpha \to \alpha$, so x in the lambda body is typed at $\boxed{\forall\alpha.\, \alpha \to \alpha}_x$. We need to pass arguments of mismatched boxed and unboxed types to *choose*. This is no problem: instantiate *choose*'s type variable β to the union $\boxed{\forall\alpha.\, \alpha \to \alpha}_x \vee (\forall\alpha.\, \alpha \to \alpha)$, letting us type the call $\mathit{choose}\ x\ (\lambda z.\, z)$, which returns the same union type. We can then upcast the result by *instantiating* the union type to $\langle\forall\alpha.\, \alpha \to \alpha\rangle$ (since each side is a subtype of it) and use that type in T-Asc.

Example 3.4. Consider again the `fooLet` function from Section 2.7. Observe that we *cannot* type `g` polymorphically enough because it lacks a type annotation on `f` to that effect.

3.3 Metatheory

To ensure the soundness of $F_{\{\le\}}$, we translate our terms into valid System F_{cc} terms. System F_{cc} is a very general declarative type system of so-called *implicit coercions* (which are essentially polymorphic subtyping assumptions parameterized with a context), designed by Cretin [2014]; Cretin and Rémy [2012, 2014]. The soundness of our translation is stated in the following theorem:

Theorem 3.5. *If $\vdash t : \tau$, then $\vdash^{F_{cc}} \tilde{t} : [\![\tau]\!]$ for some System F_{cc} term $\tilde{t}$.*

The definition of type translation $[\![\tau]\!]$ and the proof of the above theorem are enclosed in Appendix C. While the theorem only allows translating terms well-typed in the empty context, a term well-typed in an arbitrary Γ can be translated after wrapping it in abstractions to capture its free variables.

[17] In an extended system with sum and product types, we would define similar rules for pushing boxes inside them.

$$\begin{array}{lll} \textit{Positive type} & \tau^+ ::= a \mid \tau^- \to \tau^+ \mid \forall \overline{\alpha}\{\Sigma\}.\,\tau^+ \mid \top & \textit{Bound} \quad B ::= a \le \tau^- \mid \tau^+ \le a \\ \textit{Negative type} & \tau^- ::= a \mid \tau^+ \to \tau^- \mid \forall \alpha.\,\tau^- \mid \top & \textit{Type variable} \quad a, b ::= \alpha \mid \alpha_\alpha \\ \textit{Polarity} & \pm ::= + \mid - & \textit{Constraint} \quad C ::= \tau^+ \le^{\phi} \tau^- \end{array}$$

$$\textit{Contexts} \quad \Gamma ::= \epsilon \mid \Gamma \cdot (x : \tau^+) \qquad \phi ::= \Xi \mid \Xi \cdot \rhd \phi \qquad \Xi^? ::= \Xi \mid \textbf{\textit{err}} \qquad \Delta ::= \epsilon \mid \Delta \cdot C$$

Fig. 4. Polarized syntax of SuperF. Notation: we write $\tau^+ \le \tau^-$ as a shorthand for $\tau^+ \le^{\epsilon} \tau^-$.

4 TYPE INFERENCE

We now formally describe the SuperF type inference algorithm.

4.1 Restricted Polar Syntax

The syntax of types used by SuperF is presented in Figure 4. We now separate *positive* types τ^+ from *negative* types τ^-. This is reminiscent of the way that Le Botlan and Rémy disallow polymorphic types with non-trivial bounds "below arrows" in Shallow ML^F [Le Botlan and Rémy 2009].[18] We require that all type variables quantified in negative positions be boundless.

Type variables a, b include *rooted variables* α_β. Each α_β is a distinct, standalone variable with two components: a name α and a root β. We have $FV(\alpha_\beta) = \{ \alpha_\beta \}$. We use the root as metadata to ensure type inference terminates (see Section 4.3). As a shorthand, we define $\alpha_{\beta_\gamma} \triangleq \alpha_\gamma$.[19]

4.2 Type Inference Rules

The type inference rules of SuperF are presented in Figure 5. Judgment $\Gamma \vdash t : \tau^+ \Rightarrow \Delta$ is read "under context Γ, term t can be typed at τ^+ by generating constraints Δ". The constraints in the output context Δ are not resolved yet and may thus be inconsistent. The *subtype constraining* judgment $V, W \vdash \Xi \gg \Delta \gg \Sigma^?$, read "under rigid variables V, skolems W, and assumptions Ξ, the constraints Δ produce a set of bounds or an error $\Sigma^?$" and presented in the next section, is used to ensure that they are not. Type inference derivations make use of *freshness* premises. We only consider *well-formed* derivations, which are those where 'α ***fresh***' occurs at most once for each α. We write $S_1 \# S_2$ to denote that the sets S_1 and S_2 are disjoint.

Rules I-Top, I-Var, I-Asc and I-Let are straightforward. Importantly, note that the $(t : \tau^-)$ term in rule I-Asc only permits τ^- which are syntactically both positive and negative types (i.e., System F types), since τ^- also occurs where the judgment syntactically requires a positive type. Rule I-App infers the types of both parts t_1 and t_2 of an application $t_1\ t_2$, *assumes* some function type $\tau_2^+ \to \alpha$ for the former (where α is a fresh type variable), and finally returns α as a result along with the constraint that the type inferred for t_1 indeed is a subtype of $\tau_2^+ \to \alpha$. Rule I-Abs is the most interesting. It type checks the body of a lambda in a context Γ extended by $(x : \alpha)$ where α is fresh, which so far is standard. Then, it *resolves* the inferred constraints Δ, reducing them down to some bounds Ξ. These comprise bounds on the type variable α and on type variables created during lambda body typing or while resolving the constraints, as well as bounds on *outer* type variables (bound by other lambdas). The goal is to generalize the resulting polymorphic type "as much as possible" without leaking references to locally-quantified type variables into the outer polymorphic context, as we explained in Section 2.4. The *rigid variable* set V contains all variables which should

[18] Since Shallow ML^F only supports type variable lower bounds (no upper bounds) and since lower bounds are positive positions, this syntactic restriction is similar to having a polarized type syntax similar to ours.

[19] The core syntax only supports type variables a of the forms α and α_β. This shorthand defines the meaning of syntax α_a. It reflects the fact that the *root* of the variable stays the same even as a fresh type variable is duplicated into a new type variable. Without the shorthand, we would need to explicitly destructure a, which quickly becomes too verbose.

$\boxed{\Gamma \vdash t : \tau^+ \Rightarrow \Delta}$

$$\text{I-Top}\ \frac{}{\Gamma \vdash ( ) : \top \Rightarrow \epsilon} \qquad \text{I-Var}\ \frac{\Gamma(x) = \tau^+}{\Gamma \vdash x : \tau^+ \Rightarrow \epsilon} \qquad \text{I-Asc}\ \frac{\Gamma \vdash t : \tau^+ \Rightarrow \Delta}{\Gamma \vdash (t : \tau^-) : \tau^- \Rightarrow \Delta \cdot (\tau^+ \le \tau^-)}$$

$$\text{I-Let}\ \frac{\Gamma \vdash t_1 : \tau_1^+ \Rightarrow \Delta_0 \quad \Gamma \cdot (x : \tau_1^+) \vdash t_2 : \tau_2^+ \Rightarrow \Delta_1}{\Gamma \vdash \textbf{let}\ x = t_1\ \textbf{in}\ t_2 : \tau_2^+ \Rightarrow \Delta_0 \cdot \Delta_1} \qquad \text{I-App}\ \frac{\alpha\ \textit{fresh} \quad \Gamma \vdash t_1 : \tau_1^+ \Rightarrow \Delta_1 \quad \Gamma \vdash t_2 : \tau_2^+ \Rightarrow \Delta_2}{\Gamma \vdash t_1\ t_2 : \alpha \Rightarrow \Delta_1 \cdot \Delta_2 \cdot (\tau_1^+ \le \tau_2^+ \to \alpha)}$$

$$\text{I-Abs}\ \frac{\begin{array}{c}\alpha\ \textit{fresh} \quad V \supseteq FV(\Gamma) \quad V \# FV(\Delta) \setminus FV(\Gamma) \\ \Gamma \cdot (x : \alpha) \vdash t : \tau^+ \Rightarrow \Delta \quad V, \epsilon \vdash \epsilon \gg \Delta \gg \Xi \quad \mathit{split}_V(\mathit{uproot}(\Xi)) = (\Xi_0, \overline{\gamma}, \Xi_1)\end{array}}{\Gamma \vdash \lambda x.\, t : \forall \alpha \overline{\gamma}\{\Xi_1\}.\, \alpha \to \tau^+ \Rightarrow \Xi_0}$$

Fig. 5. Type inference rules of SuperF.

not be bound in the abstraction's type, and which will be bound in some outer context. This includes all type variables already in Γ, as well as additional fresh variables which are used by extrusion as approximants (see Section 4.3). To ensure constraining terminates, some type variables in Ξ may have been ascribed with *roots*; these ascriptions are unnecessary after subtype constraining is done, so they can be removed by *uproot*.

Definition 4.1 (Uprooting). Let $\mathit{uproot}(\alpha_\beta) \triangleq \alpha$, which naturally extends to τ and Ξ by congruence.

Definition 4.2 (Context splitting). Let us generalize the set difference operator '$\setminus$' to contexts. We *split* a context between outer bounds (those that mention *only* outer variables) and the rest using:

$$\mathit{split}_V(\Xi) \triangleq (\mathit{outer}_V(\Xi),\ FV(\Xi) \setminus V,\ \mathit{inner}_V(\Xi))$$

$$\mathit{outer}_V(\Xi) \triangleq \overline{\tau^+ \le \sigma^-}^{(\tau^+ \le \sigma^-) \in \Xi,\, FV(\tau^+ \le \sigma^-) \subseteq V} \qquad \mathit{inner}_V(\Xi) \triangleq \Xi \setminus \mathit{outer}_V(\Xi)$$

4.3 Subtype Constraining

The subtype constraining rules are defined in Figure 6.

Constraining algorithm. We first describe how to interpret the subtype constraining rules as an algorithm. Our algorithm uses the first variable context both as an input and an output: we input rigid variables from the context (V) and get back approximants from extrusion (V'), combining both in the judgment as a single context ($V \cdot V'$). The former is an immutable set of rigid variables while the latter is a supply of rigid variables to be used as approximants in extrusion, formally represented by the subset premises in I-Abs and C-Forall-R. In an implementation of constraining, V' would be a mutable set. In a constraining derivation $(V \cdot V'), W \vdash \Xi \gg \Delta \gg \Xi^?, V'$ and $\Xi^?$ are the only output; everything else is an input. The order in which the constraining rules are applied does not affect the correctness of the corresponding constraining judgments, but our algorithm always applies the rules in the order given in the figure. (Crucially, C-Forall-R should be applied *before* C-Forall-L when possible, as explained in Section 2.5.) In order to algorithmically construct a derivation of constraining given the inputs (V, W, Ξ, Δ), we begin by attempting to use the first rule whose conclusion shape matches the inputs. If the matching rule has constraining premises, we recurse on them. If deriving any of them results in ***err***, the algorithm backtracks and uses C-Fail to construct the derivation. Otherwise, we construct a successful derivation. In particular, if the SRLC (premise of C-Flex-L/C-Flex-R) fails, the output is ***err***.

Acyclicity. For type inference to be sound, the constructed bounds graphs must remain acyclic, because F_{cc} does not support "full" recursive types (i.e., ones where recursive occurrences may appear in bounds). We define the semantic *acyclic* check for this purpose in Appendix B.2. That definition does not count two kinds of "harmless" cycles:

$\boxed{V, W \vdash \Xi \gg \Delta \gg \Xi^?}$ *Given rigid variables V, skolems W, and assuming bounds Ξ, the constraints in Δ can be solved by introducing new bounds $\Xi^?$ (if $\Xi^? \neq$ **err**)*

C-Empty
$$\frac{}{V, W \vdash \Xi \gg \epsilon \gg \epsilon}$$

C-Top
$$\frac{V, W \vdash \Xi \gg \Delta \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (\tau \leq^{\phi} \top) \gg \Xi'}$$

C-VarRefl
$$\frac{V, W \vdash \Xi \gg \Delta \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (a \leq^{\phi} a) \gg \Xi'}$$

C-Fun
$$\frac{V, W \vdash \Xi_1 \gg \Delta \cdot (\tau_3^+ \leq^{\rhd\phi} \tau_1^-) \cdot (\tau_2^+ \leq^{\rhd\phi} \tau_4^-) \gg \Xi_2}{V, W \vdash \Xi_1 \gg \Delta \cdot (\tau_1^- \to \tau_2^+ \leq^{\phi} \tau_3^+ \to \tau_4^-) \gg \Xi_2}$$

C-Skip
$$\frac{(\tau^+ \leq \tau^-) \in \phi \quad V, W \vdash \Xi \gg \Delta \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (\tau^+ \leq^{\phi} \tau^-) \gg \Xi'}$$

C-Flex-L
$$\frac{B = (a \leq \tau^-) \quad root(B) \notin roots(\phi) \quad acyclic(\Xi \cdot B) \quad a \notin V \cdot W \quad V, W \vdash \Xi \cdot B \gg \Delta \cdot \overline{(\tau^+ \leq^{B \cdot \phi} \tau^-)}^{(\tau^+ \leq a) \in \Xi} \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (a \leq^{\phi} \tau^-) \gg \Xi' \cdot B}$$

C-Flex-R
$$\frac{B = (\tau^+ \leq a) \quad root(B) \notin roots(\phi) \quad acyclic(\Xi \cdot B) \quad a \notin V \cdot W \quad V, W \vdash \Xi \cdot B \gg \Delta \cdot \overline{(\tau^+ \leq^{B \cdot \phi} \tau^-)}^{(a \leq \tau^-) \in \Xi} \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (\tau^+ \leq^{\phi} a) \gg \Xi' \cdot B}$$

C-Rigid-L
$$\frac{a \in V \quad V, W \vdash \tau^- \rightsquigarrow (\Sigma, \sigma^-) \quad V, W \vdash \Xi \gg \Delta \cdot \Sigma \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (a \leq^{\phi} \tau^-) \gg \Xi' \cdot (a \leq \sigma^-)}$$

C-Rigid-R
$$\frac{a \in V \quad V, W \vdash \tau^+ \rightsquigarrow (\Sigma, \sigma^+) \quad V, W \vdash \Xi \gg \Delta \cdot \Sigma \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (\tau^+ \leq^{\phi} a) \gg \Xi' \cdot (\sigma^+ \leq a)}$$

C-Forall-R
$$\frac{\beta\ \textbf{fresh} \quad \sigma^- = [\alpha \mapsto \beta_\alpha]\tau^- \quad V' \supseteq FV(\tau^+ \leq \tau^-) \setminus W \quad V' \# W \cdot \beta_\alpha \quad V', W \cdot \beta_\alpha \vdash \epsilon \gg \tau^+ \leq^{\rhd\phi} \sigma^- \gg \Xi \quad V, W \vdash \Xi_0 \gg \Delta \cdot outer_{V'}(\Xi)^{\phi} \gg \Xi_1}{V, W \vdash \Xi_0 \gg \Delta \cdot (\tau^+ \leq^{\phi} \forall \alpha.\, \tau^-) \gg \Xi_1}$$

C-Forall-L
$$\frac{\overline{\beta_\alpha\ \textbf{fresh}} \quad V, W \vdash \Xi_0 \gg \Delta \cdot (\overline{[\alpha \mapsto \beta_\alpha]B})^{\phi} \cdot (\overline{[\alpha \mapsto \beta_\alpha]}\tau^+ \leq^{\phi} \tau^-) \gg \Xi_1}{V, W \vdash \Xi_0 \gg \Delta \cdot (\forall \overline{\alpha}\{\overline{B}\}.\, \tau^+ \leq^{\phi} \tau^-) \gg \Xi_1}$$

C-Fail
$$\frac{}{V, W \vdash \Xi \gg \Delta \gg \textbf{err}}$$

$\boxed{V, W \vdash \tau^{\pm} \rightsquigarrow (\Sigma, \sigma^{\pm})}$ *Given rigid variables V and skolems W, the extrusion of $\tau^{\pm}$ is $\sigma^{\pm}$ in Σ.*

X-1
$$\frac{FV(\tau^{\pm}) \subseteq V}{V, W \vdash \tau^{\pm} \rightsquigarrow (\epsilon, \tau^{\pm})}$$

X-2
$$\frac{F = FV(\tau^{\pm}) \setminus (V \cdot W) \quad \overline{(\beta_a, a, \gamma_a)\ \textbf{X-fresh} \quad \{\beta_a, \gamma_a\} \subseteq V}^{a \in F} \quad \rho = [\overline{a- \mapsto \beta_a,\ a+ \mapsto \gamma_a}^{a \in F},\ \overline{b- \mapsto \bot,\ b+ \mapsto \top}^{b \in W}] \quad W \vdash \tau^{\pm}\ \textbf{X-ok}}{V, W \vdash \tau^{\pm} \rightsquigarrow (\overline{(\beta_a \leq a) \cdot (a \leq \gamma_a)}^{a \in F},\ \rho^{\pm}\tau^{\pm})}$$

Fig. 6. Subtype constraining and extrusion rules of SuperF.

- *Immediate* type variable bound cycles. Indeed, the variables involved in an immediate cycle are simply equivalent (and could be unified). For instance, the judgment $acyclic((\alpha \leq \beta) \cdot (\beta \leq \alpha))$ holds. A type like $\forall \alpha\beta\{\alpha \leq \beta, \beta \leq \alpha\}.\ (\alpha \to \beta) \to (\alpha \to \beta)$ is equivalent to $\forall \alpha.\ (\alpha \to \alpha) \to (\alpha \to \alpha)$ and does not contain a 'real' or *semantic* cycle.
- *Contravariant* cycles like in type $\forall \alpha\beta\{\alpha \leq \alpha \to \beta\}.\ \alpha \to \beta$, which would be inferred for term $\lambda x.\ x\ x$, or equivalently $\forall \alpha\beta.\ ((\alpha \to \beta) \wedge \alpha) \to \beta$, which is equivalent and also syntactically cyclic. Such syntactically cyclic bounds do not actually lead to cyclic reasoning because they do not induce cycles in the polar traversal of types. To understand this, the reader will want to refer to the definition of polar traversal in Appendix B.2 as the $reach^{\pm}$ dual functions.

Constraint annotations. Constraints have the form $\tau^+ \leq^{\phi} \tau^-$, where ϕ is a set of subtyping relationships, some possibly guarded by $\rhd$, which are currently in the process of being constrained.

This annotation has two uses: First, it allows catching, in C-Skip, constraints that go through *immediate* type variable cycles (consider what happens with, e.g., $\Delta = (\alpha \le \beta) \cdot (\beta \le \alpha) \cdot (\alpha \le \bot)$). Note that $\in$ does not look past $\triangleright$ in ϕ, so that C-Skip is prevented from triggering on previously seen constraints after going through a type constructor (see how C-Fun adds $\triangleright$ in its sub-constraints). Indeed, we do not wish the constraining algorithm to admit recursive types because our declarative system $\mathrm{F}_{\{\le\}}$ does not support them, but this would happen if we did not guard the constraints in $\triangleright$ while going into type constructors. The effect of $\triangleright$ is to prevent C-Skip from picking up constraints that are being solved at an outer constructor level. Second, it allows the SRLC to catch constraints applying on roots currently being constrained (as an example, see Appendix A.5). The reason why ϕ cannot be a simple *context* in the constraining judgment is that constraining is specified as a "tail-recursive" worklist algorithm, where subderivations do not necessarily correspond to implied constraints, since they also concern "sibling" constraints in Δ. We also write $(\overline{\tau^+ \le \sigma^-})^\phi$ as a shorthand for $\overline{\tau^+ \le^\phi \sigma^-}$.

Constraining rules. Rules C-Empty, C-Top, C-VarRefl, and C-Fun are straightforward. There are four nontrivial rules for type variables. C-Flex-L and C-Flex-R register new bounds on locally-flexible type variables and traverse the transitive consequences of these bounds. They perform a cyclicity check to ensure that cyclic bounds such as $\alpha \le \tau \to \alpha$ are rejected and a roots-check to guarantee termination. Moreover, they augment the sub-constraint annotations with the bound B currently being propagated. Finally, they output B as part of the result. C-Rigid-L and C-Rigid-R register *extruded* bounds on *locally-rigid* variables. Locally-rigid variables are those bound by outer lambdas as well as those currently rigidified by an outer application of C-Forall-R; extrusion ensures that inner variables do not leak through the bounds of outer ones. C-Forall-L *instantiates* the type variables and bounds of a polymorphic type found on the left. C-Forall-R skolemizes a quantified variable found on the right-hand side, as described in Section 2.5. We first compute the set of *free variables* V' involved in the subsequent $\tau^+ \le \tau^-$ comparison in order to locally rigidify them. We recurse on that comparison, which gives us the intermediate result Ξ. Finally, we continue constraining, adding the delayed rigid variable bounds $\mathit{outer}_{V'}(\Xi)$ as new constraints to solve. We can discard $\mathit{inner}_{V'}(\Xi)$ because these bounds only concern purely local variables that were created while constraining the skolemized right-hand-side type. For instance, when constraining $\forall\alpha.\, \alpha \to \alpha \le \forall\beta.\, \beta \to \beta$, these inner bounds will contain $\alpha'_\alpha \le \beta'_\beta$ and $\beta'_\beta \le \alpha'_\alpha$, which are irrelevant outside the subderivation. Note that there is no need to relate V' and V: it is enough to only have all free type variables in $\tau^+ \le \tau^-$. We suspend the constraining of any type variables *other than* those that are locally introduced while instantiating nested polymorphic types. In the second subderivation, we resume constraining the delayed bounds on V' induced by $\tau^+ \le \sigma^-$.

Polar substitutions. Notation $\rho = [a- \mapsto \tau^-,\ a+ \mapsto \tau^+]$ denotes a *polar substitution* ρ, which is a substitution that maps the *positive* occurrences of a type variable a to some negative type τ^- and the *negative* occurrences of a to a possibly different positive type τ^+. A polar substitution ρ is applied as either $\rho^+(\sigma^+)$ or $\rho^-(\sigma^-)$ depending on the assumed polarity of $\sigma^\pm$. (Since the syntaxes of positive and negative types overlap, it is not always possible to tell their polarity without context.)

Extrusion. Extrusion is a critical part of polymorphic type inference. Its task is to avoid adding to a type variable bounds that are "more polymorphic" than the type variable itself. We distinguish between *locally-rigid* type variables V and *skolems* W: the former are *less* polymorphic than the current polymorphism level, while the latter are *more* polymorphic. Skolems are introduced by C-Forall-R and represent arbitrary future instantiations; less polymorphic types should thus not be allowed to mention them. Rules C-Rigid-L and C-Rigid-R use extrusion to ensure that the bounds added to rigid type variables mention neither flexible variables nor skolems. Extrusion

itself is defined with two rules: X-1 and X-2. The first one simply filters out types that already only mention rigid variables. In X-2, set F is the list of flexible variables to extrude. Flexible variables are "more polymorphic" than rigid variables, thus flexible variables should not appear in the bounds of rigid variables. Thankfully, we are allowed to add new bounds to flexible variables, so that we can create less polymorphic *approximants* for them to replace their uses in the bounds of rigid variables. An approximant is related to the original polymorphic type variable through the latter's bounds. Concretely, each $a \in F$ is extruded into a lower approximant β_a (for the negative occurrences of a) and an upper approximant γ_a (for the positive occurrences of a), and we register the bounds $\beta_a \le a$, $a \le \gamma_a$ as constraints — these should be viewed as bounds registered on a, as the approximants cannot themselves refer to the more polymorphic a in their own bounds. $[a- \mapsto \beta_a,\ a+ \mapsto \gamma_a]$ denotes the corresponding polar substitution. The **X-fresh** premise denotes that each a is always approximated into the *same* approximants in the whole constraining derivation and that each approximant is an otherwise fresh rigid variable. We also use an **X-ok** premise to ensure that the bounds of universal types remain consistent after extrusion.[20] In contrast to flexible variables, skolems are "too polymorphic" and their occurrences can only be approximated to either $\top$ or $\bot$, depending on each occurrence's polarity. We cannot register approximants for skolems because we are not allowed to add bounds to skolems, which denote *unknown, arbitrary* types. Both skolems and flexible variables are approximated by extrusion such that the rigid variable bound resulting from C-Rigid-L and C-Rigid-R is at least as constraining as the corresponding type from the original constraint. For instance, if α is rigid and β is flexible, then the constraint $\alpha \le \top \to \beta$ results in a bound $\alpha \le \top \to \gamma_\beta$ such that $\gamma_\beta \le \beta$, where γ_β is the *lower approximant* of β, a special locally-rigid variable. The approximant is bound at the same polymorphism level as α, making it equally polymorphic as α and generalized alongside it, for example in I-Abs (see the description of I-Abs in Section 4.2). Approximants always end up being bounded by the opposite bounds of their root variable: the lower approximant of β inherits β's upper bounds, and the upper approximant of β inherits β's lower bounds, which results from adding the approximant bounds as *constraints* to be solved later (rather than as solved bounds). This ensures that bounding a flexible variable with its approximants yields consistent bounds, correctly making constraints like $\alpha \to \alpha \le \forall\beta.\, \beta \to \beta$ fail.

Example 4.3. Assuming $f : (\forall\alpha.\, \alpha \to \alpha) \to \beta$, notice that **let** $x = \lambda y.\, y$ **in** $f\ (\lambda z.\, x\ z)$ can be typed in SuperF because $\forall\gamma.\, \gamma \to \gamma$ is inferred as the type of x and subsequently used parametrically in $x\ z$. On the other hand, $(\lambda x.\, f\ (\lambda z.\, x\ z))\ (\lambda y.\, y)$ cannot be typed by SuperF because we never assume parametrically-polymorphic types for parameters. In this case, the type variable α assigned to z and the type variable γ assumed for the result of the $x\ z$ application are leaked outside the scope of the λz abstraction because they are involved in constraint $\beta \le \alpha \to \gamma$, where β is the type variable assigned to x. Because of this constraint, both α and γ need to be extruded to the lower polymorphic level of β. Therefore, $\lambda z.\, x\ z$, whose inferred type is, $\alpha \to \gamma$, cannot be assigned the polymorphic type required by f. Note also that after η-contracting $\lambda z.\, x\ z$ to x, $(\lambda x.\, f\ x)\ (\lambda y.\, y)$ become typeable in SuperF, which shows that SuperF, like $\mathrm{F}_{\{\le\}}$, is not stable under η expansion.

Termination and the SRLC. The Suspiciously Recursive-Looking Criterion (SRLC) is formally represented as the two *roots* premises in C-Flex-L and C-Flex-R. As explained in Section 2.5, the goal of the SRLC is to ensure that for all inputs, deriving subtype constraining always terminates. We provide an intuitive proof sketch for the termination theorem in Appendix D.1.

[20] This is currently implemented as a rather ad-hoc and incomplete check, given by Definition B.8 in Appendix B.3,which crudely ensures that polymorphic types will not get inconsistent bounds after extrusion by checking that the skolems being extruded are not reachable from both the upper bounds and lower bounds of the same type variable. If this check fails, type inference conservatively fails.

Definition 4.4 (Roots). Define $root(\alpha_\beta) \triangleq \beta$, $root(\alpha) \triangleq \alpha$ and, when τ is not a type variable, $root(\tau) \triangleq [\overline{a \mapsto root(a)}^{a \in FV(\tau)}]\tau$. In addition, define $root(\tau \le \sigma) = root(\tau) \le root(\sigma)$, $roots(\Xi) \triangleq \{root(\tau \le \sigma) \mid (\tau \le \sigma) \in \Xi\}$, and $roots(\Xi \cdot \rhd \phi) \triangleq roots(\Xi) \cup roots(\phi)$.

Order of constraining type variables. The fact that C-Flex-L and C-Flex-R are tried before C-Rigid-L and C-Rigid-R is important to avoid non-termination of the constraining algorithm. In a previous version of this paper [Parreaux et al. 2024], we mistakenly had the rules in the opposite order (contrary to what our implementation actually does), which is fixed in this version.

4.4 Distributivity

Distributivity, which is supported by $F_{\{\le\}}$, adds flexibility to the type inference system. The basic idea of using distributivity in type inference is described in the two following rules:

$$\text{C-Forall-Distr-R}\quad \frac{\alpha \notin FV(\tau_1^+) \qquad \Xi_1 \gg \Delta \cdot (\tau_1^+ \le^\phi \forall\alpha.\, \tau_1^+ \to \tau_2^-) \gg \Xi_2}{\Xi_1 \gg \Delta \cdot (\tau_1^+ \le^\phi \tau_1^+ \to \forall\alpha.\, \tau_2^-) \gg \Xi_2}$$

$$\text{C-Forall-Distr-L}\quad \frac{split_{FV(\tau_1^-)}(\Sigma) = (\Sigma_0, \_, \Sigma_1) \qquad V_0 = V \cap FV(\Sigma_0) \qquad V_1 = V \setminus V_0 \qquad \Xi_1 \gg \Delta \cdot (\forall V_0\{\Sigma_0\}.\, \tau_1^- \to \forall V_1\{\Sigma_1\}.\, \tau_2^+ \le^\phi \tau_3^-) \gg \Xi_2}{\Xi_1 \gg \Delta \cdot (\forall V\{\Sigma\}.\, \tau_1^- \to \tau_2^+ \le^\phi \tau_3^-) \gg \Xi_2}$$

C-Forall-Distr-R is straightforward: it simply pushes quantifiers out of arrow-type right-hand sides so as to make skolemization happen as early as possible. C-Forall-Distr-L does the reverse, pushing quantifiers into arrow-type right-hand sides to delay their instantiation. The latter is more complicated to specify because it has to deal with splitting multi-bound quantifiers, filtering out those type variables whose bounds transitively refer to the arrow type's left-hand side and which can therefore not be distributed. These rules are straightforwardly generalized to look deeply inside arrow types, so that they can push quantifiers across multiple nested arrow types; while this is implemented in our prototype, we omit the formal details here for the sake of brevity.

Limitations. Distributivity reasoning in SuperF is unlike the rest of the type inference system, which is why we describe it in its own subsection. The application of all other type inference rules is "uncontroversial", in the sense that there is no obvious better thing to do in the context where each rule applies. On the other hand, whether to apply distributivity is a choice, and the choice is non-obvious: while applying it is *usually* beneficial (so in practice, we always apply it eagerly whenever possible), it can occasionally lead to *worse* type inference outcomes, yielding errors when a type inference run without distributivity would have succeeded.

For example, consider the following program, which contains a type ascription on subterm `a`:

```
test f = let a () = f () in a : ⊤ → (∀ b. b → b)
```

When typing `f ()`, SuperF creates a fresh type variable α and constrains the type of `f` to be a subtype of $\top \to \alpha$. But because the type variable of `f` is *less polymorphic*, that type is *extruded* into $\top \to \beta_\alpha$ where $\alpha \ge \beta_\alpha$ and β_α is a fresh type variable at the same (outer) polymorphism level as `f`. Binding `a` is then assigned the generalized function type $\tau = \forall\alpha\{\beta_\alpha \le \alpha\}.\, \top \to \alpha$. Then, when typing the ascription in the body of the let binding, a constraint of the form $\tau \le \top \to \sigma_{\text{id}}$ is generated. SuperF finds that the left-hand side is a polymorphic type and the right-hand side is an arrow whose right-hand side is polymorphic, so it has two choices:

- Distribute the latter out (i.e., what the implementation does), resulting in $\tau \le \forall\gamma.\, \top \to \gamma \to \gamma$, which can be skolemized to $\tau \le \top \to \gamma^{\text{sk.}} \to \gamma^{\text{sk.}}$, after which the left-hand side is finally instantiated to $\top \to \alpha'$ where $\alpha' \ge \beta_\alpha$. So the constraint results in adding upper bound $\top \to \gamma^{\text{sk.}} \to \gamma^{\text{sk.}}$ to the local flexible variable α' and propagating that bound to β_α. But β_α is a less polymorphic (rigid) type variable, so the bound is extruded to $\top \to \top \to \bot$, an approximation that may later lead to type errors.

- Refrain from distributing and instead instantiate the LHS, resulting in $\top \to \alpha' \le \top \to \sigma_{\mathrm{id}}$ where $\alpha' \ge \beta_\alpha$, which decomposes to $\alpha' \le \sigma_{\mathrm{id}}$. This results in adding bound σ_{id} to α' and propagating that bound to β_α. Notice that this time, no extrusion/avoidance is performed, because we kept the right-hand side nested polymorphic type intact.

So while distributive reasoning is often useful, there are cases where it is in fact counter-productive. The "dummy" LHS polymorphic type τ, which just wraps a plain type variable with a single lower bound, leads SuperF to distribute the RHS although that is not necessary to solve the constraint successfully, which in turn leads to type extrusion. To remain predictable, SuperF does not perform any non-local reasoning on whether performing distributivity is likely to be beneficial or not, and so it fails in this example. It is an open question whether we can determine when distributivity is unnecessary, or at least whether we can design a heuristic to avoid with high accuracy the cases when it is likely harmful.

4.5 Metatheory

The following theorem captures the main correctness criterion for type inference:

Theorem 4.5 (Soundness of Type Inference). *If* $\vdash t : \tau^+ \Rightarrow \Delta$ *and* $\vdash \epsilon \gg \Delta \gg \Xi'$*, then we have* $\vdash t : \forall V\{\Xi\}.\, \tau^+$*, where* $\Xi = \mathit{uproot}(\Xi')$*, and* $V = FV(\Xi)$*.*

If type inference infers a type for a closed term, we obtain a typing that can be translated to System F_{cc}. We state the accompanying correctness theorem of constraining as follows:

Definition 4.6 (Constraint entailment). With $\Xi \vdash \Delta$ we denote that $\overline{\Xi \vdash \tau \le \sigma}^{(\tau \le \sigma) \in \Delta}$.

Theorem 4.7 (Soundness of Constraining). *Let* $V, \overline{\alpha} \vdash \epsilon \gg \Delta \gg \Sigma'$ *and* $\Sigma = \mathit{uproot}(\Sigma')$*. Then for all* $\overline{\tau}$ *where* $FV(\overline{\tau}) \# \overline{\beta}$ *and* $\overline{\beta} = FV(\Sigma) \setminus \{ V \cdot \overline{\alpha} \}$ *we have*

- $\mathit{outer}_V(\Sigma) \cdot \mathit{inner}_V([\overline{\alpha \mapsto \tau}]\Sigma) \vdash [\overline{\alpha \mapsto \tau}]\Delta$ *and*
- $\mathit{outer}_V(\Sigma) \vdash \forall \overline{\beta}\{\mathit{inner}_V([\overline{\alpha \mapsto \tau}]\Sigma)\}$ ***cons.***.

This theorem is only stated for derivations whose assumptions are empty ϵ; this matches how constraining is used in I-Abs and C-Forall-R. Intuitively, the results Σ of such derivations satisfy two properties: they entail (or resolve) the input constraints Δ, and their inner flexible variable bounds $\mathit{inner}_V(\Sigma)$ are consistent if we assume the outer rigid variable bounds $\mathit{outer}_V(\Sigma)$. These properties hold for all skolem replacement $[\overline{\alpha \mapsto \tau}]$, which is crucially important for C-Forall-R. We prove Theorems 4.5 and 4.7 in Appendix D.

5 PRACTICAL IMPLEMENTATION OF SUPERF

In this section, we describe MLscriptF, our implementation of SuperF in the existing MLscript language, and empirically evaluate it on existing test suites and examples from previous work literature. There are a few inessential differences between MLscriptF and SuperF as formalized in Section 4, the main ones being that MLscriptF:

- supports type checking recursive definitions, which is not shown in Section 4 but is straightforward: we bind the definition's name to a fresh type variable while typing the definition's body and then constrain that type variable to be a supertype of the inferred body type. We also refrain from generalizing the functions constituting the bodies of these recursive definitions, as that would lead to polymorphism extrusion and unnecessary failures.[21]

[21] Another way of understanding this is that we do not support type inference for polymorphically-recursive functions.

- does not generalize lambdas nested under other lambdas (as in curried functions); doing so is redundant since distributivity can always be used to push polymorphic types back inside arrow types when needed. More subtly, we also do not want to incur *too much* polymorphism in the presence of recursive terms, as that would degrade the type checker's performance.
- uses explicit *polymorphism levels* to track extrusion and type avoidance, as described in Section 2.4. This can be seen as an optimization (to minimize the number of type traversals) that does not affect the functional properties of the system.
- has top-level `def` bindings with call-by-name semantics which are always generalized.

5.1 Implementation and Validation on MLscript Test Suite

We have implemented our approach in MLscript, a new ML-family language currently in active development. The MLscript codebase already had a significant amount of regression tests written for it, amounting to about 8000 lines of test *code* (not counting comments nor blank lines), which runs (in parallel) in about 7 seconds on a recent x86 MacBook laptop.[22] Many of these tests produce statistics, such as the number of calls to the subtype constraining method, to check that the amount of work remains reasonable. About 70% of these tests were written before implementing first-class polymorphism (FCP); they test various aspects of the language through small functions constructed in it. This makes us confident that the addition of our FCP technique to an existing language is practical and does not introduce performance problems in the form of, for example, pathological cases that would blow up the time spent type checking a given file.

5.2 Validation on ML^F Test Suite

The test suite of ML^F was graciously provided to us by its authors Le Botlan and Rémy [2009]. It provided us with a lot of interesting and tricky examples, which SuperF all managed to type check. Because ML^F is the previously-unbeaten champion in FCP type inference expressiveness, we are quite satisfied that we could handle its test suite better than ML^F itself, which required manual type annotations in more places.

5.3 Validation on Existing OCaml Code

We have experimentally evaluated our claim that SuperF subsumes ML type inference by porting the `List` module implementation from OCaml's standard library (`list.ml`), which is about 500 lines of non-empty, non-comment lines of ML code. Our implementation

- did not require any type annotations;
- inferred relatively concise/compact and readable types;
- inferred types that were often more precise than the OCaml ones, thanks to subtyping; and
- inferred types which were successfully *checked against* the explicit type signatures provided in the OCaml standard library's `list.mli` interface file.

These claims are substantiated in supplementary material (see the file `OCamlList.md`).

5.4 Evaluation on Examples from The Literature

Table 2 presents all the examples that we could gather from previous work on FCP. (Most of them were already summarized by Emrich et al. [2020] for their paper on FreezeML.) We observe that

[22] All running times in this section include the time to actually *run* the tests (which is done via JavaScript code generation and executed through *nodejs*), as MLscript is a real programming language and not just a type checker.

Table 2. Typeability comparisons in different systems. '●' means the term type checks; '●*' means its inferred type is not as general as it could be (assuming SuperF subtyping); '◐' means the term type checks only after adding some type-free *polymorphism annotations* (e.g., FreezeML's freezing annotations); '○' means a type error is raised. SuperF−D is SuperF *without distributivity*, while SuperF has distributivity.

		MLF	FPH	HMF	HML	GI	FreezeML	SuperF−D	SuperF
A.	**Polymorphic instantiation**								
A1	λx. λy. y	●	●	●	●	●	●*	●	●
A2	choose id	●	●*	●*	●	●	●*	●	●
A3	choose nil ids	●	●	●	●	●	●	●	●
A4	λx. x x	○	○	○	○	○	○	●	●
A5	id auto	●	●	●	●	●	●	●	●
A6	id auto'	●	●	●	●	●	●*	●	●
A7	choose id auto	●	○	●	●	●	●	●	●
A8	choose id auto'	○	○	○	○	○	○	●	●
A9	f (choose id) ids	●	●	○	●	○	◐	●	●
	where $f : \forall\alpha.(\alpha \to \alpha) \to \mathsf{List}\ \alpha \to \alpha$								
A10	poly id	●	●	●	●	●	◐	●	●
A11	poly (λx. x)	●	●	●	●	●	◐	●	●
A12	id poly (λx. x)	●	●	●	●	●	◐	●	●
B.	**Inference with polymorphic arguments**								
B1	λf. (f 1, f True)	○	○	○	○	○	○	●	●
B2	λxs. poly (head xs)	●	○	○	○	○	○	●	●
C.	**Functions on polymorphic lists**								
C1	length ids	●	●	●	●	●	●	●	●
C2	tail ids	●	●	●	●	●	●	●	●
C3	head ids	●	●	●	●	●	●	●	●
C4	single id	●	●	●*	●	●	●*	●	●
C5	cons id ids	●	●	●	●	●	◐	●	●
C6	cons (λx. x) ids	●	●	●	●	●	◐	●	●
C7	append (single inc) (single id)	●	●	●	●	●	●	●	●
C8	g (single id) ids	●	●	○	●	○	◐	●	●
	where $g : \forall\alpha.\mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \to \alpha$								
C9	map poly (single id)	●	●	○	●	○	◐	●	●
C10	map head (single ids)	●	●	●	●	●	●	●	●
D.	**Application functions**								
D1	app poly id	●	●	●	●	●	◐	●	●
D2	revapp id poly	●	●	●	●	●	◐	●	●
D3	runST argST	●	●	●	●	●	◐	●	●
D4	app runST argST	●	●	●	●	●	◐	●	●
D5	revapp argST runST	●	●	●	●	●	◐	●	●
E.	**η-expansion**								
E1	k h l	●	○	○	○	○	○	●	●
E2	k (λx. h x) l	●	●	●	●	●	◐	●	●
	where $k : \forall\alpha.\alpha \to \mathsf{List}\ \alpha \to \alpha$ $h : \mathsf{Int} \to \forall\alpha.\alpha \to \alpha$ $l : \mathsf{List}\ (\forall\alpha.\mathsf{Int} \to \alpha \to \alpha)$								
E3	r (λx. λy. y)	●	○	○	●	○	◐	●	●
	where $r : (\forall\alpha.\alpha \to \forall\beta.\beta \to \beta) \to \mathsf{Int}$								
F.	**FreezeML paper additions**								
F5	auto id	●	●	●	●	●	◐	●	●
F6	cons (head ids) ids	●	○	●	●	●	●	●	●
F7	head ids 3	●	●	●	●	●	◐	●	●
F8	choose (head ids)	●	●	●	●	●	●	●	●
F9	**let** f = revapp id **in** f poly	●	○	○	●	○	◐	●	●
F10	choose id (λx. auto' x)	●	○	○	○	○	○	●	●

SuperF seamlessly handles *all* these previous examples without the need for any type or other polymorphism/freezing annotations, with or without distributivity.

$\text{head} : \forall\alpha.\, \text{List}\ \alpha \to \alpha$
$\text{tail} : \forall\alpha.\, \text{List}\ \alpha \to \text{List}\ \alpha$
$\text{nil} : \forall\alpha.\, \text{List}\ \alpha$
$\text{cons} : \forall\alpha.\, \alpha \to \text{List}\ \alpha \to \text{List}\ \alpha$
$\text{single} : \forall\alpha.\, \alpha \to \text{List}\ \alpha$
$\text{append} : \forall\alpha.\, \text{List}\ \alpha \to \text{List}\ \alpha \to \text{List}\ \alpha$
$\text{length} : \forall\alpha.\, \text{List}\ \alpha \to \text{Int}$
$\text{const} : \forall\alpha\,\beta.\, \alpha \to \beta \to \alpha$
$z = \lambda f.\, \lambda x.\, x$

$\text{id} : \forall\alpha.\, \alpha \to \alpha$
$\text{ids} : \text{List}\ (\forall\alpha.\, \alpha \to \alpha)$
$\text{inc} : \text{Int} \to \text{Int}$
$\text{choose} : \forall\alpha.\, \alpha \to \alpha \to \alpha$
$\text{poly} : (\forall\alpha.\, \alpha \to \alpha) \to (\text{Int}, \text{Bool})$
$\text{fst} : \forall\alpha\,\beta.\, (\alpha, \beta) \to \alpha$
$\text{auto} : (\forall\alpha.\, \alpha \to \alpha) \to (\forall\alpha.\, \alpha \to \alpha)$
$\text{auto}' : \forall\beta.(\forall\alpha.\, \alpha \to \alpha) \to (\beta \to \beta)$
$s = \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$

$\text{map} : \forall\alpha\,\beta.\, (\alpha \to \beta) \to \text{List}\ \alpha \to \text{List}\ \beta$
$\text{app} : \forall\alpha\,\beta.\, (\alpha \to \beta) \to \alpha \to \beta$
$\text{revapp} : \forall\alpha\,\beta.\, \alpha \to (\alpha \to \beta) \to \beta$
$\text{runST} : \forall\alpha.\, (\forall s.\, \text{ST}\ s\ \alpha) \to \alpha$
$\text{argST} : \forall s.\, \text{ST}\ s\ \text{Int}$
$\text{zero} : \text{ChurchInt}$
$\text{succ} : \text{ChurchInt} \to \text{ChurchInt}$
$\text{ChurchInt} = \forall\alpha.\, (\alpha \to \alpha) \to (\alpha \to \alpha)$
$n3 = s\ (s\ (s\ z))$

Fig. 7. Type signatures and definitions for terms used in Table 2 and Table 3.

Table 3. Typeability comparisons (continued). Legends are in Table 2; '×' means non-termination or crash.

		MLF	FPH	HMF	HML	GI	FreezeML	SuperF−D	SuperF
G.	**SuperF additions (this paper)**								
G1A	`z : ChurchInt`	●	●	●	●	●	●	●	●
G2	`s`	●*	●*	●*	●*	●*	●	●*	●
G2A	`s : ChurchInt → ChurchInt`	○	○	○	○	○	●	○	●
G3A	`n3 : ChurchInt`	●	●	●	●	○	●	●	●
G4A	`(fun () → n3 n3) : () → ChurchInt`	●	●	○	●	○	○	○	●
G5	`fst (fst (fst (n3 (λx. (x, 0)) 1)))`	○	○	○	○	○	○	●	●
G6	`(succ (succ zero)) (succ (succ zero))`	●	●	●	●	○	●	●	●
G7	`(s (s z)) (s (s z))`	●	●	●	●	○	●	●	●
G8	`let rec to_church n = if n == 0 then z else else s (to_church (n - 1))`	●*	●*	●*	●*	○	●	●*	●
G8A	`to_church : Int → ChurchInt`	○	○	○	○	○	●	○	●
G9	`let rec i x = if True then x else i i x`	○	○	○	○	×	○	○	○
G10	`(λx. x x) (λx. x x)`	○	○	○	○	○	○	○	○
G11	`auto (auto' id)`	●	●	●	●	●	○	○	●
G12	`(λy. let tmp = y id in y const) (λx. x x)`	○	○	○	×	○	○	●	●
G13	`(λk. (k (λx. x), k (λx. single x))) (λf. (f 1, f True))`	○	○	○	○	○	○	●	●
G14	`(λf. let a() = f id in (a() : Int → (∀'b. 'b → 'b))) (const (const id))`	●	○	○	○	○	○	●	○

5.5 New Examples

Because the examples presented in previous work were too simple to truly test the limits of SuperF, we created a number of more intricate expressions in Table 3, several of which come directly from the test suite of ML^F. Note that SuperF does not use bidirectional type checking, so for an annotated term `t : T` to type check, `t` itself must type check. Therefore, the annotation in, e.g., `s : ChurchInt → ChurchInt` does *not* help type inference — it only checks that the type inferred for `s` can be instantiated to the less precise type `ChurchInt → ChurchInt`.

SuperF cannot type check examples G2A, G4A, G8A, and G11 without distributivity. For example, in G11, distributivity pushes a quantifier into the result type of `auto'`, making it polymorphic, so it can be passed to `auto`. G4A is interesting because it is type checked successfully by MLF, FPH, and HML but not by SuperF without distributivity. This is due to SuperF only generalizing lambda abstractions, whereas these other systems generalize even applications (see also Section 2.7).

G14 is a contrived example manually constructed to demonstrate that distributivity in conjunction with subtype extrusion and type annotations sometimes leads to worse outcomes: SuperF distributes `Int → (∀'b. 'b → 'b)` into `∀'b. Int → ( 'b → 'b)`, leading to the premature skolemization of `'b` and to more extrusion than without distributing.

The definition in G9 uses itself recursively at different types and therefore needs polymorphic recursion, for which SuperF would require a type annotation — adding the explicit type signature

‘`i : ∀ a. a → a`’ makes it type check. Interestingly, it is alternatively possible to type check this example using recursive types, which our implementation also supports (behind a feature flag).

6 RELATED WORK

We now discuss some related work.

6.1 First-Class Polymorphism in ML-style languages

First-class polymorphism has been studied extensively in the setting of ML-style languages, where polymorphism is normally implicit.

Boxed polymorphism and related systems. *Poly-ML* [Garrigue and Rémy 1997] and its predecessors [Läufer and Odersky 1994; O'Toole and Gifford 1989; Rémy 1994] augment ML type schemes with boxed polytypes. In Poly-ML, boxed polytypes wrap type schemes into simple ML types using the syntax $[\sigma]$, preventing implicit instantiation. The introduction form of boxed polytypes requires explicit annotation of the type scheme ($[t : \sigma]$). Boxed polytypes must be eliminated explicitly at the use site with the syntax $\langle t \rangle$ to instantiate the type scheme. The approach never infers first-class-polymorphic types, but only propagate the polytypes provided by the user. However, this juxtaposition of ML-style type inference and System F-style type checking means that the order of the terms being processed can affect the result of type inference, since extra type information may be obtained from type inference. When a let-binding or an application is typed, it is possible that one of the two subterms must be typed before the other for the other to be typeable. Since there is no way to tell in advance if a type inference path would succeed, the algorithm must backtrack and explore the other paths when it fails. The number of such paths is usually exponential in the size of the program, leading to combinatorial explosion. For example, while typing $\lambda f.\ \langle f \rangle\ (g\ f)$, where $g = \lambda f : [\forall \alpha.\ \alpha \to \alpha].\ \langle f \rangle\ f$, typing the former $\langle f \rangle$ would fail since the type of f is a type variable, not a boxed polytype. However, typing the argument $(g\ f)$ first would force the type of f to be $[\forall \alpha.\ \alpha \to \alpha]$, which then allows $\langle f \rangle$ to be typed. To address this, the authors chose to reject programs where any of the paths may fail. This rejects a wide range of otherwise typeable programs and requires explicit type annotations on lambda parameters. The authors claim to have achieved complete type inference by incorporating the notion of "labels" into the declarative system, where each boxed polytype is associated with a label. A user-provided type annotation implicitly quantifies over all labels. A boxed polytype can only be opened when its label does not occur anywhere else, which implies that it must originate from an annotation. In SuperF, the unannotated version of g can be typed as $\lambda f.\ f\ f : \forall \alpha\beta\{\alpha \le \alpha \to \beta\}.\ \alpha \to \beta$, and the unannotated version of the problematic term above as $\lambda f.\ f\ (g\ f) : \forall \alpha\beta\gamma\{\alpha \le \alpha \to \beta,\ \alpha \le \beta \to \gamma\}.\ \alpha \to \gamma$, which is a subtype of the only annotated version typeable in Poly-ML, $\lambda f : [\forall \alpha.\ \alpha \to \alpha].\ \langle f \rangle\ (g\ f)$, with type $[\forall \alpha.\ \alpha \to \alpha] \to [\forall \alpha.\ \alpha \to \alpha]$.

ML^F. Le Botlan and Rémy [2003] introduced ML^F, which has been the "undefeated champion" of type inference for first-class polymorphism *until now*. ML^F uses a more flexible type language than System F in that it includes bounded quantification, whereby types are ordered based on their polymorphism "power", akin to our subtyping relation. ML^F takes inspiration from boxed polymorphism and attempts to alleviate the verbose introduction and elimination of boxes. This is achieved by introducing two forms of type variable bindings, namely rigid bindings ($\alpha \Rrightarrow \sigma$) and flexible bindings ($\alpha \ge \sigma$), which can appear in a context called the "prefix" and in binders for bounded polymorphism. The usual subsumption relation is split into an abstraction relation $\sqsubseteq$ and an instance relation $\sqsupseteq$. The $\sqsubseteq$ relation is used to implicitly abstract polytypes as type variables with a rigid binding in the prefix, which allows the polymorphism to propagate by preventing implicit instantiation (cf. boxed polymorphism in Poly-ML). For example, consider the function

$a = \lambda z.\ \omega^\dagger\ z$, where $\omega^\dagger = \lambda(x : \sigma_{\text{id}})\ x\ x$. It is clear that z must have a polymorphic type, but it is not used polymorphically in a, its polymorphism is merely passed to $\omega^\dagger$. Abstracting σ_{id} as α in the prefix allows the body of a to be typed by $(\alpha \Rightarrow \sigma_{\text{id}})\ z : \alpha \vdash \omega^\dagger\ z : \alpha$ without ever guessing polymorphism. Then a can be typed as $\forall(\alpha \Rightarrow \sigma_{\text{id}})\ \alpha \to \alpha$. Moreover, as seen in the type of a, rigid bindings keep track of sharing, which is crucial to type inference as it allows instantiation to be delayed. To reveal and instantiate an abstraction, the programmer must explicitly provide a type annotation that corresponds to the rigid binding. Contrary to the reversible $\sqsubseteq$ abstraction, the instance relation $\sqsubseteq$ is used to irreversibly instantiate a type scheme, subject to the flexible bindings. Intuitively, rigid bindings represent the types of terms that must be polymorphic, while flexible bindings represent the types that may be instantiated. For example, the principal type of $\omega^\dagger$ is $\forall(\alpha \Rightarrow \sigma_{\text{id}}, \alpha' \geq \sigma_{\text{id}})\ \alpha \to \alpha'$, where the parameter type α must be polymorphic since it is used polymorphically in the body, but the result type α' may be instantiated to any instance of σ_{id}. In ML$^{\text{F}}$, lambda parameters that are used polymorphically require annotations, since type inference does not guess second-order types. SuperF does not guess second-order polymorphic types either, but uses multiple bounds (which can be represented as intersections) to accommodate for finitarily-polymorphic uses of unannotated parameters.

Le Botlan and Rémy note that ML$^{\text{F}}$ infers principal types for types which are typeable in their system, but that typing some terms may require type annotations — and in fact, different annotations may lead to incomparable principal types. For example, they remark that $\lambda x.\ x\ x$ is not typeable in ML$^{\text{F}}$ while both $\lambda(x : \forall\alpha.\alpha)\ x\ x$ and $\lambda(x : \forall\alpha.\alpha \to \alpha)\ x\ x$ are, but neither has a more general type (their types are unrelated). On the other hand, $\lambda x.\ x\ x$ is typeable in SuperF, inferring type $\forall\alpha\beta.\ ((\alpha \to \beta) \wedge \alpha) \to \beta$ (i.e., $\forall\alpha\beta\gamma\{\gamma \leq \alpha \to \beta,\ \gamma \leq \alpha\}.\ \gamma \to \beta$ or equivalently $\forall\alpha\beta\{\alpha \leq \alpha \to \beta\}.\ \alpha \to \beta$), which is a subtype of *both* ML$^{\text{F}}$ types mentioned above.

The bindings in prefixes and quantifiers in ML$^{\text{F}}$ resemble constrained types in SuperF. However, a few distinctive features in SuperF contribute to its greater expressiveness over ML$^{\text{F}}$. Firstly, function types are contravariant in their parameter types in SuperF, whereas they are invariant in their parameter types in ML$^{\text{F}}$. Contravariance in the subtyping relation is crucial to type inference. For example, consider the two terms $f = \lambda(x : \sigma_{\text{id}} \to \sigma_{\text{auto}}).\ x$ and $g = \lambda(x : \sigma_{\text{auto}}).\ id$, where $id = \lambda x.\ x$, $\sigma_{\text{id}} = \forall\alpha.\ \alpha \to \alpha$ and $\sigma_{\text{auto}} = \sigma_{\text{id}} \to \sigma_{\text{id}}$. The application $f\ g$ is typeable in a straightforward manner in SuperF thanks to subtyping. The same term is, however, not typeable in ML$^{\text{F}}$. Secondly, as a result of the restriction to monotypes under type constructors, ML$^{\text{F}}$ requires the introduction of "administrative" type variable bindings even when these bindings are used once and do not represent any shared information. Let us consider the following example. The inferred type of id is the expected $\sigma_{\text{id}} = \forall\alpha.\ \alpha \to \alpha$ in both SuperF and ML$^{\text{F}}$. One may wish to instantiate α to some concrete type, say σ_{id}. In SuperF, id can be typed as $\sigma_{\text{id}} \to \sigma_{\text{id}}$, which is $\alpha \to \alpha$ with σ_{id} substituted for α. However, in ML$^{\text{F}}$, since only monotypes are allowed under an arrow, the type scheme σ_{id} must be abstracted in a binding as $\forall(\alpha \Rightarrow \sigma_{\text{id}})\ \alpha \to \alpha$. One may expect id to also be typeable as $\forall(\alpha \Rightarrow \sigma_{\text{id}})\ \forall(\beta \Rightarrow \sigma_{\text{id}})\ \alpha \to \beta$, or even the more general $\sigma_{\text{id}} \to \sigma_{\text{id}}$, which is a shorthand for $\forall(\alpha \Rightarrow \sigma_{\text{id}})\ \forall(\alpha' \geq \sigma_{\text{id}})\ \alpha \to \alpha'$. This is, however, not the case as ML$^{\text{F}}$ keeps track of sharing even after the $\alpha \geq \bot$ binding has been instantiated and rigidified to $\alpha \Rightarrow \sigma_{\text{id}}$. The arguably counterintuitive pseudo-subtyping relation in ML$^{\text{F}}$ imposes some burden on users, and may catch them by surprise. Finally, SuperF allows multiple bounds on the same variable, including multiple upper bounds and multiple lower bounds, whereas ML$^{\text{F}}$ only allows one lower or rigid bound on a variable. Supporting multiple bounds for each type variable allows us to effectively infer intersection types in negative positions (such as function parameters) and union types in positive positions (such as function results).

Variations on ML^F. Rémy and Yakobowski [2007] proposed a graphical representation of ML^F types that allows for an efficient unification algorithm. Le Botlan and Rémy further investigated a "shallow" version of ML^F where type annotations and rigid bounds are restricted to System F types [Le Botlan and Rémy 2009] which, while less expressive, still has interesting properties, and more importantly admits simpler semantics and metatheory. Finally, Rémy and Yakobowski [2012] designed xML^F, a "Church-style" intermediate language for MLF which would be amenable to intrinsically-typed compilation as in compilers like the Glasgow Haskell Compiler (GHC).

Restrictions on ML^F. *HML* [Leijen 2009] is a simplification of ML^F restricting type variable bindings to flexible bounds (discarding rigid bounds). HML retains the expressiveness of by requiring annotations on polymorphic function arguments, which is marginally higher than the requirement in ML^F. The author believes that this requirement to programmers is justified by the simpler types they work with. Like ML^F, HML is robust against most program transformations, including abstracting and inlining let-bindings, and abstraction with higher-order functions (most notably with `app` and `revapp`). An exception to this is η-expansion, which may require a type annotation on a polymorphic parameter type. The author believes that robustness is an important property as it forms the basis of equational reasoning. SuperF enjoys even more robustness thanks to there being no requirement to annotate polymorphic parameters, thanks to constrained types, which makes η-expansion robust as well as the other transformations.

HMF [Leijen 2008] is another, more drastic restriction of ML^F discarding flexible bindings, i.e., a simple extension of the Hindley-Milner type system with System F types, supporting type inference through a straightforward extension of algorithm W. The author believes that a simple type system is beneficial in that it eases the burden on programmers in understanding the types, and in simplifying the metatheory and the implementation of type inference. To achieve this, HMF always predicatively instantiates ambiguous applications eagerly. Annotations are thus required when impredicative instantiation is needed, in addition to the usual annotation requirement on function parameters that are used polymorphically. This has the adverse effect of introducing a dependency of typability on the order of parameter, which is alleviated in some cases by considering application chains all at once, instead of treating each of their constituent applications individually. This system has the major drawbacks that type inference is less stable than in ML^F-style systems.

FPH [Vytiniotis et al. 2008] is yet another proposed type inference system which uses System F types but can still be understood as a restriction of ML^F. *QML* [Russo and Vytiniotis 2009] extends ML types with universal and existential quantified types, which co-exist with type schemes. While type schemes are *implicitly* introduced by let-bindings and eliminated when applications happen, quantified types are *explicitly* introduced and eliminated by type annotations. For example, let $id = \lambda x.\ x$ in id will be typed as a type scheme $\Pi(\alpha)(\alpha \to \alpha)$. To have a first-class-polymorphic type, an explicit type annotation is necessary as let $pid = \{\forall\alpha.\ \alpha \to \alpha\}\ id$ in pid, here pid has the type $\Pi(\epsilon)(\forall\alpha.\ \alpha \to \alpha)$. On the other hand, it is also necessary to provide an explicit type annotation to eliminate this quantified type as $(pid\ \{\forall\alpha.\ \alpha \to \alpha\})\ 3$. Like many systems, QML also does not guess polymorphic types for function parameters. Parameters can only be polymorphic when they are annotated with quantified types. As an example, let $poly = \lambda f.$ let $y = f\ \{\forall\alpha.\ \alpha \to \alpha\}$ in $(y\ 1, y\ \texttt{true})$ in $poly$ has the type $(\forall\alpha.\ \alpha \to \alpha) \to (Int, Bool)$. The treatment of existentially-quantified types is similar, requiring explicit type annotations to introduce and eliminate these types.

FreezeML. *FreezeML* [Emrich et al. 2020, 2022] introduces a "*freeze*" operator to explicitly disable instantiation. Otherwise, like in ML, polymorphic types are implicitly instantiated. For example, *single id* where *single* : $\forall a.\ a \to \mathsf{List}\ a$ and $id : \forall a.\ a \to a$ is typed as $\mathsf{List}\ (a \to a)$ since id's type is implicitly instantiated. In contrast, with id frozen using syntax $\lceil id \rceil$, then *single* $\lceil id \rceil$ is typed

as List ($\forall a.\ a \to a$). With the frozen operator and some macro expansions, users may explicitly control whether the types of terms are generalized or instantiated. FreezeML's type inference algorithm extends the traditional algorithm W and achieves principal type inference of first-class polymorphism. However, it requires a significant number of annotations for polymorphic code: users need to manually control the explicit freezing, instantiation, and generalization of polymorphic types, *in addition to* the need of annotating the full type of polymorphic function parameters, which altogether amounts to a significant burden for programmers. It seems SuperF is more general than FreezeML — indeed, we conjecture that *most* (if not *all*) FreezeML programs can be typed by SuperF even after stripping all their freezing and instantiation annotations. For example, in FreezeML, the example E2 can only be typed with two explicit operators (as $k\ \$(\lambda x.(h\ x))@\ l$) while SuperF requires no annotations at all. Table 2 in appendix shows that such annotations are required for the majority of examples from the existing literature, although these examples are all quite simple. FreezeML annotations, while more concise than having to specify full types, are arguably not very intuitive. The E2 annotations above are needed to explicitly adapt the quantification of h from $Int \to \forall a.\ a \to a$ to $\forall a.\ Int \to a \to a$, so it becomes compatible with a type that only happens to *flow into* the same result variable. Users are effectively reduced to retyping by η expansion and adding freezing/instantiation in a way that may seem quite obscure to whoever reads the code afterward. Furthermore, FreezeML does not support reordering quantifiers implicitly. For example, $\forall \alpha\ \beta.\ \alpha \to \beta$ and $\forall \alpha\ \beta\ \gamma.\ \alpha \to \beta$ and $\forall \beta\ \alpha.\ \alpha \to \beta$ are all considered distinct, incompatible types [Emrich et al. 2022]. One may use explicit generalization to restore a canonical order of quantifiers, but this still results in a quite inflexible system where many valid programs fail due to syntactic details, rather than semantic ones. While FreezeML can type the `foo` examples from the introduction using various sets of annotations, it cannot assign a principal type to it, so there is no best set of annotations to use for that example. In particular, no set of annotations can let one type check both `foo (fun x → x)` and `foo (fun x → Some x)` with the expected result types, whereas our system type checks both "out of the box", without any annotations.

First-class polymorphism for Haskell. *Boxy types* [Vytiniotis et al. 2006] were used to implement GHC's original `ImpredicativeTypes` extension. The system incorporates bidirectional type inference by distinguishing types to be inferred and types to be checked using boxes. Boxy-types retains the idea of not guessing polymorphic types, but refines it by inferring polytypes for function arguments with locally known type information. The system as presented was very fragile against local transformations and thus difficult to use in practice, which is why it was eventually replaced by the more predictable *Quick Look*. For example, typability is not preserved under η-expansion.

Guarded instantiation (GI) [Serrano et al. 2018] focuses on simplicity and tries to balance between complexity, expressiveness, and annotation burden. In GI, polymorphic instantiation only happens on guarded type variables. The guardedness is decided in function applications by examining whether the type variable is guarded by type constructors in any parameter types, for example both type variables in $\forall a\ b.\ (a \to b) \to \mathsf{List}\ a \to \mathsf{List}\ b$ are guarded but neither in $\forall a\ b.\ a \to b$ is. Unguarded type variables may only be instantiated by monomorphic types or polymorphic types without top-level quantifiers, such as $Int \to \forall a.a$. Note that only the types of those parameters that are given corresponding arguments are considered. As an example, in *append id* where *append* : $\forall a.\ a \to \mathsf{List}\ a \to \mathsf{List}\ a$, though a is guarded in the second parameter type, in this partial application only the first parameter type is considered for the guardedness, so a cannot be polymorphicly instantiated. In *append id ids*, the second argument is now given so a is considered guarded and therefore may be instantiated polymorphicly. Type instantiation in GI takes no account of the context of the call and each argument in a call is treated independently between different function

applications. As usual, for lambdas, GI will not guess polymorphic parameters. Parameters may only be polymorphic when annotated.

Quick Look [Serrano et al. 2020] is a type inference system for Haskell that further simplifies the first-class-polymorphic type system. The authors claim that it is eminently practical for Haskell programming. However, it is quite inexpressive when compared to most systems mentioned previously, and in particulat it removes all forms of subtyping (so-called "deep subsumption") from Haskell, needing manual η expansions instead even to just reorder or distribute quantifiers. Since the official support of Quick Look in GHC around 2021, users of GHC have already complained about its inflexibility, as it makes them manually eta-expand code in many places where they did not need to before and where it appears "clear" that the compiler should be able to figure things out.[23]

Predicative higher-rank polymorphic type inference. Type inference can become much easier if one is to abandon the *impredicative* setting of first-class polymorphic type inference, i.e., if one separates the syntax of types between polymorphic and monomorphic types, only allowing the latter to be substituted for type variables. Odersky and Läufer [1996] demonstrated an example of this approach in their seminal work on type inference for higher-rank polymorphism. Many works have followed up on the idea, bringing various improvements [Dunfield and Krishnaswami 2013, 2019; Peyton Jones et al. 2007; Vytiniotis et al. 2006; Xie 2021; Xie and Oliveira 2018; Xue and Oliveira 2021] many of which relying on the idea of bidirectional typing [Dunfield and Krishnaswami 2021; Dunfield and Pfenning 2004].

Stability considerations. Le Botlan and Rémy [2009] originally investigated the stability of first-class polymorphic type inference in the context of ML^F. As Table 1 shows, we support most of their stability guarantees and some more. Bottu and Eisenberg [2021] recently tried to characterize such a notion of stability for Haskell. They developed 6 "similarity" properties and showed that in the context of GHC, all combinations of *shallow or deep* and *eager or lazy* instantiation break some of these similarities, though the *lazy and shallow* combination breaks the least. Of these similarities, one does not hold in SuperF due to the restricted language of annotations (similarity 2, about inferred types being usable as type signatures). The rest of these similarities trivially pass the stability criterion in SuperF, although SuperF is *deep* (and *not* shallow) as well as being lazy.[24]

6.2 Type Inference for System F

Boehm [1985] and Pfenning [1988] proposed partial type inference procedures based on second-order unification which is undecidable but has a practical semi-algorithm due to Huet [1975]. Their systems required placeholder annotations (without type information) for all type abstractions and type applications, which they called a *partial type inference* approach. Mitchell [1988] introduced the now-usual notion of *type containment* for System F, a notion of subtyping involving polymorphic types (what was known as *subtyping* at the time did not include polymorphic subtyping). He showed how to make System F complete with respect to η expansion in System F_η, meaning all terms that System F can type after η expanding some of their subterms are typeable in System F_η through subsumption. Type inference for System F_η was later shown undecidable, due to an undecidable type containment relation Wells [1996]. Rémy [2005] proposed a simpler partial type inference system for System F based on a similar notion of type containment, but the approach was not very satisfactory, pushing Rémy to investigate the more promising appraoch of ML^F.

[23] For example, see the following highly-upvoted reddit comment thread entitled "*Was simplified subsumption worth it for industry Haskell programmers?*" (accessed June 2022): https://redd.it/ujpzx3.

[24] The reason Bottu and Eisenberg feel the need to be *shallow* in their approach rather than *deep* can be attributed to the presence of Haskell-specific features in their system, such as explicit type applications, and to their lack of subtyping.

In parallel, there has been much work on systems like $F_{<:}$ which combined first-class polymorphism, type variable bounds, and subtyping, and even also sometimes included intersections [Pierce 1997]. However, these systems usually assume *explicit* polymorphism, whereby polymorphic types have to be introduced and eliminated in terms, and where subtyping between them is "structural," i.e., there are no relationships between types such as `∀ a. a → a` and `Int → Int`. This is by contrast to the traditional ML approach to polymorphism, which is implicit.

In this paper, we follow the ML tradition of implicit polymorphism, which is more appropriate as a foundation for generalizing the ML type system to first-class polymorphism. More specifically, we follow an approach closely inspired by *algebraic subtyping* [Dolan 2017], and its reformulation as *Simple-sub* [Parreaux 2020], including its novel approach to nested polymorphism with extrusion in the presence of subtyping.

6.3 Polyvariant Flow Analysis

Palsberg and Pavlopoulou [1998] showed that *polyvariant* analysis (also known as *context-sensitive* analysis) can be related formally to a subtyping system with union, intersection, and recursive types. Unions model sets of abstract values and intersections model each usage of an abstract value. Their system conspicuously does not feature polymorphism.

Faxén [1997]; Smith and Wang [2000] propose inferring polymorphic types rather than intersections for function definitions, which is more flexible and composable as it can process unrelated definitions separately, whereas the approach based solely on intersections is a global process. Smith and Wang develop a polymorphic system with subtype constraint solving designed for flow analysis. Their constraint closure rules are evocative of our C-Flex-L/C-Flex-R, C-Fun, and C-Forall-L. However, they have no rule analogous to C-Forall-R, nor do they need any notions of type avoidance and extrusion, which are proper to a *type inference* view of constraint solving (as opposed to the restrictive view of flow analysis). They give functions polymorphic types that contain all locally-inferred *unsolved subtyping constraints*.[25] The fact they do not solve constraints locally means they do not check for the consistency of inferred types (so they may fail to report type errors at definition sites) and it means their constraints are resolved over and over again every time the polymorphic types that capture them are instantiated. Similarly to us, their basic system is non-terminating and they introduce a termination condition which "detects a certain kind of self-referential flow in the constraints". They show that termination can be guaranteed by merging some instantiations in this case. Wang and Smith [2001] adapt this approach to object-oriented programming but abandon the first-class polymorphism part.

Rehof and Fähndrich [2001] also rely on polymorphic subtyping; they design a control-flow analysis that can be reduced to the problem of context-free language reachability. They represent constraint sets as substitutions to avoid copying constraints around and obtain better scalability. Whether this insight can be applied to SuperF remains to be determined.

6.4 Intersection and mixed type systems

Jim's polarized type system called "P" [Jim 2000] mixes universal quantification with intersection types in a way that strongly resembles ours, restricting the former to positive type positions and the latter to negative type positions. However, to retain decidability, the types of P are much more restricted syntactically. For example, they do not admit universal quantifiers nor intersections on the right of arrow types. P has special typing rules for variable and lambda applications. The former uses in its premise a *rewritten* lambda term (*not* a subterm of the term being typed), which makes

[25] Faxén proposes simplifying these constraints incrementally, but this does not change the problem fundamentally.

is difficult to explain and characterize well-typed terms on an intuitive level.[26] Moreover, since the syntax of negative types does not include polymorphic types, users cannot check their programs against provided System F type signatures.

6.5 Implicit Coercion Constraints

Motivated by the goal of designing a type system that could generalize all previous bounded quantification approaches (mainly ML-style constrained types [Odersky et al. 1999], System $F_{<:}$ [Pierce 1991], and ML^F [Le Botlan and Rémy 2003]), Cretin and Rémy [2014] developed a calculus of *implicit coercions* called System F_{cc}. Coercion constraints are very expressive and designed in a way that is more general than bounded quantification — notably, they do not require the use of a dedicated typing rule for generalization, lifting generalization to the coercion relation (which is a form of generalized subtyping). Thankfully, it is straightforward to encode our *multi-bounded polymorphism* type system into F_{cc}, as we demonstrate in Appendix C. Scherer and Rémy [2015] later investigated versions of coercion constraints allowing abstracting over possibly-inconsistent coercions, to support applications like GADTs [Xi et al. 2003].

7 CONCLUSION AND FUTURE WORK

We presented SuperF, a novel type inference algorithm for first-class polymorphism based on *multi-bounded polymorphism*. SuperF is uniquely expressive, stable under small program changes, and allows for understandable error messages to be reported. We also presented $F_{\{\leq\}}$, a simple type system that serves as the declarative basis of SuperF and that is founded on the theory of implicit coercions from the existing F_{cc}. SuperF was implemented as part of a real-world programming language and evaluated on previous test suites as well as examples from the literature.

SuperF could serve as a starting point for more advanced type systems features, such as higher-order subtyping, existentials and generalized algebraic data types (GADTs), as well as dependent types. For example, SuperF could provide an adequate algorithmic basis to the cDOT calculus of Boruch-Gruszecki et al. [2022] and to the $\lambda_I^\forall$ calculus of Xue and Oliveira [2021], who left algorithmic formulations of their systems for future work. Unlike SuperF, $\lambda_I^\forall$ uses *predicative* polymorphism. Restricting SuperF to predicative type instantiation should be straightforward and may allow dropping the SRLC. We would also like to support *unrestricted* type annotations in the future, which may be helped by integrating some form of bidirectional typing.

ACKNOWLEDGMENTS

We would like to thank Didier Rémy, Didier Le Botlan, Bruno Oliveira, Stephen Dolan, and the anonymous reviewers for their helpful feedback. This research was partially funded by Hong Kong Research Grant Council project number 26208821.

DATA AVAILABILITY STATEMENT

The artifact of this paper, which was granted the "Functional" and "Reusable" badges, is available on Zenodo at https://zenodo.org/records/8424750 [Parreaux et al. 2023]. The corresponding Github repository is available at https://github.com/hkust-taco/superf and the system can be tried in a web demo at https://hkust-taco.github.io/superf/.

[26] Generally, type inference can be greatly simplified if it is allowed to perform some steps of reduction in the program being considered. For example, ML type inference can be expressed as a simple type unification pass after reducing all let bindings. However, such *rewriting-based* approaches are not realistic and are not used in practice [Le Botlan and Rémy 2009].

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* (Copenhagen, Denmark) *(FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 31–41. https://doi.org/10.1145/165180.165188 ↪ page 5

Ishan Bhanuka, Lionel Parreaux, David Binder, and Jonathan Immanuel Brachthäuser. 2023. Getting into the Flow: Towards Better Type Error Messages for Constraint-Based Type Inference. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 237 (oct 2023), 29 pages. https://doi.org/10.1145/3622812 ↪ page 14

Richard S. Bird and Ross Paterson. 1999. De Bruijn Notation as a Nested Datatype. *J. Funct. Program.* 9, 1 (jan 1999), 77–91. https://doi.org/10.1017/S0956796899003366 ↪ page 38

Hans-J. Boehm. 1985. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. 339–345. https://doi.org/10.1109/SFCS.1985.44 ↪ page 31

Aleksander Boruch-Gruszecki, Radosław Waundefinedko, Yichen Xu, and Lionel Parreaux. 2022. A Case for DOT: Theoretical Foundations for Objects with Pattern Matching and GADT-Style Reasoning. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 179 (oct 2022), 30 pages. https://doi.org/10.1145/3563342 ↪ pages 15 and 33

Gert-Jan Bottu and Richard A. Eisenberg. 2021. Seeking Stability by Being Lazy and Shallow: Lazy and Shallow Instantiation is User Friendly. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell* (Virtual, Republic of Korea) *(Haskell 2021)*. Association for Computing Machinery, New York, NY, USA, 85–97. https://doi.org/10.1145/3471874.3472985 ↪ page 31

Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. 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, Nara, Japan, 378–391. https://doi.org/10.1145/2951913.2951928 ↪ page 7

Jacek Chrząszcz. 1998. Polymorphic subtyping without distributivity. In *Mathematical Foundations of Computer Science 1998*, Luboš Brim, Jozef Gruska, and Jiří Zlatuška (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 346–355. ↪ page 2

Julien Cretin. 2014. *Erasable coercions: a unified approach to type systems.* Theses. Université Paris-Diderot - Paris VII. https://tel.archives-ouvertes.fr/tel-00940511 ↪ pages 3 and 16

Julien Cretin and Didier Rémy. 2012. On the Power of Coercion Abstraction. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL 2012)*. Philadephia, PA, USA. To appear. ↪ page 16

Julien Cretin and Didier Rémy. 2014. System F with Coercion Constraints. In *Logics In Computer Science (LICS)*. ACM. ↪ pages 1, 2, 16, and 33

Stephen Dolan. 2017. *Algebraic subtyping*. Ph. D. Dissertation. ↪ pages 14 and 32

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. *ACM SIGPLAN Notices* 52, 1 (Jan. 2017), 60–72. https://doi.org/10.1145/3093333.3009882 ↪ pages 1, 7, 8, and 9

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (may 2021), 38 pages. https://doi.org/10.1145/3450952 ↪ page 31

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism. *SIGPLAN Not.* 48, 9 (Sept. 2013), 429–442. https://doi.org/10.1145/2544174.2500582 ↪ page 31

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290322 ↪ page 31

Jana Dunfield and Frank Pfenning. 2004. Tridirectional Typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/964001.964025 ↪ page 31

Frank Emrich, Sam Lindley, Jan Stolarek, James Cheney, and Jonathan Coates. 2020. FreezeML: Complete and Easy Type Inference for First-Class Polymorphism. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 423–437. https://doi.org/10.1145/3385412.3386003 ↪ pages 24 and 29

Frank Emrich, Jan Stolarek, James Cheney, and Sam Lindley. 2022. Constraint-Based Type Inference for FreezeML. *Proc. ACM Program. Lang.* 6, ICFP, Article 111 (aug 2022), 26 pages. https://doi.org/10.1145/3547642 ↪ pages 29 and 30

Karl-Filip Faxén. 1997. Polyvariance, polymorphism and flow analysis. In *Analysis and Verification of Multiple-Agent Languages*, Mads Dam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–278. ↪ pages 8 and 32

A. Frisch, G. Castagna, and V. Benzaken. 2002. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 137–146. https://doi.org/10.1109/LICS.2002.1029823 ↪ page 7

Jacques Garrigue and Didier Rémy. 1997. Extending ML with semi-explicit higher-order polymorphism. In *Theoretical Aspects of Computer Software*, Martín Abadi and Takayasu Ito (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–46. ↪ page 27

P. Giannini and S.R. Della Rocca. 1988. Characterization of typings in polymorphic type discipline. In *[1988] Proceedings. Third Annual Symposium on Logic in Computer Science*. 61–70. https://doi.org/10.1109/LICS.1988.5101 ↪ page 38

G.P. Huet. 1975. A unification algorithm for typed λ-calculus. *Theoretical Computer Science* 1, 1 (1975), 27–57. https://doi.org/10.1016/0304-3975(75)90011-0 ↪ page 31

Trevor Jim. 2000. A Polar Type System.. In *ICALP Satellite Workshops*. Citeseer, 323–338. ↪ pages 8 and 32

Oleg Kiselyov. 2013. Efficient generalization with levels (Okmij Blog). http://okmij.org/ftp/ML/generalization.html#levels. Accessed: 2020-06-30. ↪ page 10

Konstantin Läufer and Martin Odersky. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Trans. Program. Lang. Syst.* 16, 5 (sep 1994), 1411–1430. https://doi.org/10.1145/186025.186031 ↪ page 27

Didier Le Botlan and Didier Rémy. 2003. MLF: Raising ML to the power of System F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. 27–38. ↪ pages 27, 28, 33, and 38

Didier Le Botlan and Didier Rémy. 2009. Recasting MLF. *Information and Computation* 207, 6 (2009), 726–785. https://doi.org/10.1016/j.ic.2008.12.006 ↪ pages 12, 13, 17, 24, 29, 31, and 33

Daan Leijen. 2008. HMF: Simple Type Inference for First-Class Polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1411204.1411245 ↪ page 29

Daan Leijen. 2009. Flexible Types: Robust Type Inference for First-Class Polymorphism. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 66–77. https://doi.org/10.1145/1480881.1480891 ↪ pages 12 and 29

Daniel Leivant. 1990. Discrete Polymorphism. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 288–297. https://doi.org/10.1145/91556.91675 ↪ page 5

John C. Mitchell. 1988. Polymorphic type inference and containment. *Information and Computation* 76, 2 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0 ↪ page 31

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 54–67. https://doi.org/10.1145/237721.237729 ↪ page 31

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55. ↪ pages 2 and 33

J. W. O'Toole and D. K. Gifford. 1989. Type Reconstruction with First-Class Polymorphic Values. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI '89)*. Association for Computing Machinery, New York, NY, USA, 207–217. https://doi.org/10.1145/73141.74836 ↪ page 27

Jens Palsberg and Christina Pavlopoulou. 1998. From Polyvariant Flow Information to Intersection and Union Types. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/268946.268963 ↪ page 32

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. https://doi.org/10.1145/3409006 ↪ pages 2, 7, 10, and 32

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2023. *When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism (Artifact)*. https://doi.org/10.5281/zenodo.8424750 ↪ page 33

Lionel Parreaux, Aleksander Boruch-Gruszecki, Andong Fan, and Chun Yin Chau. 2024. When Subtyping Constraints Liberate: A Novel Type Inference Approach for First-Class Polymorphism. *Proc. ACM Program. Lang.* 8, POPL, Article 48 (jan 2024), 33 pages. https://doi.org/10.1145/3632890 ↪ pages 1 and 22

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019. Towards Improved GADT Reasoning in Scala. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala* (London, United Kingdom) *(Scala '19)*. Association for Computing Machinery, New York, NY, USA, 12–16. https://doi.org/10.1145/3337932.3338813 ↪ page 15

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. https://doi.org/10.1145/3563304 ↪ page 7

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 17, 1 (Jan. 2007), 1–82. https://doi.org/10.1017/S0956796806006034 ↪ pages 3, 11, 31, 38, and 39

Frank Pfenning. 1988. Partial Polymorphic Type Inference and Higher-Order Unification. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (Snowbird, Utah, USA) *(LFP '88)*. Association for Computing Machinery, New York, NY, USA, 153–163. https://doi.org/10.1145/62678.62697 ↪ page 31

Benjamin C Pierce. 1991. *Programming with intersection types and bounded polymorphism*. Ph. D. Dissertation. Citeseer. ↪ page 33

Benjamin C. Pierce. 1997. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science* 7, 2 (1997), 129–193. https://doi.org/10.1017/S096012959600223X ↪ page 32

François Pottier. 1998. *Type Inference in the Presence of Subtyping: from Theory to Practice.* Research Report RR-3483. INRIA. https://hal.inria.fr/inria-00073205 ↪ page 8

Francois Pottier and Didier Rémy. 2005. *The Essence of ML Type Inference.* 389–489. ↪ page 10

Jakob Rehof and Manuel Fähndrich. 2001. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (London, United Kingdom) *(POPL '01)*. Association for Computing Machinery, New York, NY, USA, 54–66. https://doi.org/10.1145/360204.360208 ↪ pages 8 and 32

Didier Rémy. 1990. *Algèbres Touffues. Application au Typage Polymorphe des Objets Enregistrements dans les Langages Fonctionnels.* Thèse de doctorat. Université de Paris 7. ↪ page 10

Didier Rémy. 1992. *Extension of ML type system with a sorted equation theory on types.* Research Report RR-1766. INRIA. https://hal.inria.fr/inria-00077006 Projet FORMEL. ↪ page 10

Didier Rémy. 1994. Programming Objects with ML-ART, an Extension to ML with Abstract and Record Types. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS '94)*. Springer-Verlag, Berlin, Heidelberg, 321–346. ↪ page 27

Didier Rémy. 2005. Simple, Partial Type-Inference for System F Based on Type-Containment. *SIGPLAN Not.* 40, 9 (Sept. 2005), 130–143. https://doi.org/10.1145/1090189.1086383 ↪ pages 5 and 31

Didier Rémy and Boris Yakobowski. 2007. A Graphical Presentation of MLF Types with a Linear-Time Unification Algorithm. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/1190315.1190321 ↪ page 29

Claudio Russo and Dimitrios Vytiniotis. 2009. QML: Explicit First-Class Polymorphism for ML. In *ML '09 Proceedings of the 2009 ACM SIGPLAN workshop on ML* (ml '09 proceedings of the 2009 acm sigplan workshop on ml ed.). ACM New York, NY, USA, 3–14. https://www.microsoft.com/en-us/research/publication/qml-explicit-first-class-polymorphism-for-ml/ ↪ page 29

Didier Rémy and Boris Yakobowski. 2012. A church-style intermediate language for MLF. *Theoretical Computer Science* 435 (2012), 77–105. https://doi.org/10.1016/j.tcs.2012.02.026 Functional and Logic Programming. ↪ page 29

Gabriel Scherer and Didier Rémy. 2015. Full Reduction in the Face of Absurdity. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 685–709. ↪ pages 15 and 33

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (aug 2020), 29 pages. https://doi.org/10.1145/3408971 ↪ pages 4 and 31

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 783–796. https://doi.org/10.1145/3192366.3192389 ↪ pages 4 and 30

Scott F. Smith and Tiejun Wang. 2000. Polyvariant Flow Analysis with Constrained Types. In *Programming Languages and Systems*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 382–396. ↪ pages 8 and 32

Jerzy Tiuryn and Pawel Urzyczyn. 1996. The Subtyping Problem for Second-Order Types is Undecidable. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS 1996)* (New Brunswick, NJ, USA). IEEE Computer Society Press, 74–85. ↪ page 2

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2006. Boxy Types: Inference for Higher-Rank Types and Impredicativity. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 251–262. https://doi.org/10.1145/1159803.1159838 ↪ pages 4, 30, and 31

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. 2008. FPH: First-Class Polymorphism for Haskell. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/1411204.1411246 ↪ page 29

Tiejun Wang and Scott F. Smith. 2001. Precise Constraint-Based Type Inference for Java. In *ECOOP 2001 — Object-Oriented Programming*, Jørgen Lindskov Knudsen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–117. ↪ page 32

Joseph B. Wells. 1996. *Type inference for System F with and without the eta rule.* Ph. D. Dissertation. http://ezproxy.ust.hk/login?url=https://www.proquest.com/dissertations-theses/type-inference-system-f-with-without-eta-rule/docview/304323713/se-2 Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-24. ↪ page 31

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*.

Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150 ↪ pages 33 and 38

Ningning Xie. 2021. Higher-rank polymorphism : type inference and extensions. http://hdl.handle.net/10722/307011 ↪ page 31

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 272–299. ↪ page 31

Mingqi Xue and Bruno C.d.S. Oliveira. 2021. A dependently typed calculus with polymorphic subtyping. *Science of Computer Programming* 208 (2021), 102655. https://doi.org/10.1016/j.scico.2021.102655 ↪ pages 31 and 33

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A Mechanical Formalization of Higher-Ranked Polymorphic Type Inference. *Proc. ACM Program. Lang.* 3, ICFP, Article 112 (jul 2019), 29 pages. https://doi.org/10.1145/3341716 ↪ page 10

A ADDITIONAL SUPERF EXAMPLES

In this appendix, we provide some additional examples related to SuperF type inference.

A.1 Concrete Example of Need for Annotations

Consider a function `mapExprs: (∀ a. Expr a → Expr a) → Program → Program`, which presumably transforms the typed subexpressions of a program by applying an argument function at an arbitrary number of different expression types that are not known statically (`Expr` could be, for instance, the usual typed expression GADT [Xi et al. 2003] or an expression data type with type-level de Bruijn indices [Bird and Paterson 1999; Peyton Jones et al. 2007]). The following SuperF definition fails to infer a sufficiently polymorphic type in this case:

```
mapAndPrintSize f = mapExprs (fun e → print_int (size e) ; f e)
```

Indeed, in this case, parameter `f` is assigned a type variable and the local type variable assigned to `e` is extruded through it. So calling `mapAndPrintSize id` yields an error, as the local `a` skolem in `Expr a` (from the definition of `mapExprs`) flows out to the extruded type variable where it undergoes type avoidance, resulting in `Expr ⊥` and `Expr ⊤` before flowing back into `Expr a`, which fails because ⊥ and ⊤ do not match `a`. In cases like this, a simple type annotation resolves the problem:

```
mapAndPrintSize (f: ∀ a. Expr a → Expr a) =
  mapExprs (fun e → print_int (size e); f e)

mapAndPrintSize id  --  ok
```

Stronger forms of this restriction are present in all previous first-class and higher-rank polymorphism type inference approaches we are aware of, including ML^F. The latter requires that *all parameters used polymorphically* must be annotated, whereas we only require annotations for parameters used *parametrically*-polymorphically (i.e., *non-parametric* — or *finitary* — polymorphism can be left unannotated thanks to intersection type inference). Other approaches generally require annotations for parameters that are to be assigned a polymorphic type at all, which is even more restrictive. For instance, consider that in both SuperF and ML^F, the following definition is accepted:

```
mapAndPrintHello f = print_string "Hello!" ; mapExprs f

mapAndPrintHello : (∀ a. Expr a → Expr a) → Program → Program
```

A.2 Expressiveness of SuperF

Consider the following `monster` term along with some helper definitions:

```
I x = x
K x y = x
auto x = x x

monster = (fun y → (let tmp = y I in y K)) auto
```

which is not typeable in System F (though it is typeable in F_ω) [Giannini and Della Rocca 1988], leading Le Botlan and Rémy [2003] to conjecture that it is not typeable in their ML^F type system either.

But there is nothing *fundamentally* hard to type about this example. System F can in fact type `monster` after it is made to take just one step of reduction (leading to `let tmp = self I in self K`). The reason System F cannot type the original term is that System F lacks the expressiveness needed to accurately describe a set of constraints inferred from the program's data flow.

SuperF infers the following types for this program, without the help of any type annotations:

```
I : ∀ a. a → a
K : ∀ a. a → ⊤ → a
```

```
auto : ∀ a b. ((a → b) ∧ a) → b

monster : ⊤ → ∀ a. a → ⊤ → a
```

To understand how this works, note that the `fun y → (let tmp = y I in y K)` subterm of `monster` is given inferred type `∀ r. ((∀ a. a → ⊤ → a) → r) ∧ ((∀ a. a → a) → ⊤) → r`, which clearly shows that parameter `y` is inferred to be a function expected to yield some type `r` when applied to `K`, of type `KT = ∀ a. a → ⊤ → a`, and to yield any type `⊤` when applied to `I` (since the result of that application is discarded), with the overall result type being `r`. Then, applying that subterm to `auto` is simply a matter of checking that `auto` can receive both `K` and `I` as individual arguments while yielding `r` as a result of the former, which is expressed as the subtyping constraint `∀ a. ((a → b) ∧ a) → b ≤ KT → r`, which decomposes into `b ≤ r` and `KT ≤ (a0 → b) ∧ a0`, into `a1 → ⊤ → a1 ≤ a0 → b` (i.e., `a0 ≤ a1` and `⊤ → a1 ≤ b`) and `KT ≤ a0`, which after simplification yields `r = ⊤ → KT`, the `monster` type shown above. We describe the general idea of how such subtyping constraints are solved in Section 2.5 and rigorously formalize the process in Section 4.3.

It is easy to show that these types are principal using the same approach as in Section 2.2 — intuitively, all the constraints we solved above are indeed *necessary* consequences of the term being well-typed, and thus the constrained types we infer are essentially representations of every constraint any valid type must satisfy.

A.3 Unlimited-rank Type Inference

SuperF is not limited in the rank of the polymorphic types it infers. In practice, it always infers parametrically-polymorphic types for odd ranks and intersection types for even ranks.

Consider the following elaboration on our introduction example:

```
foo f = (f 123, f True)

bar f = (f (fun x → x), f (fun x → Some x))

test = bar foo
```

SuperF successfully infers the following types:

```
foo : ∀ a b. ((Int → a) ∧ (Bool → b)) → (a, b)

bar : ∀ a b. (((∀ c. c → c) → a) ∧ (∀ d. d → Some[d]) → b) → (a, b)

test : ((Int, Bool), (Option Int, Option Bool))
```

Notice that the type inferred for `bar` has rank 3, as it contains `∀` types behind two arrow type left-hand sides [Peyton Jones et al. 2007, §3.1].

A.4 Type Error Messages

Assuming `k : (∀ a. a → a) → int`, the expression `k (fun x → x + 1)` results in the following error message in SuperF, where the flow of rigid type variable `a` into parameter `x` is clear:

```
[ERROR] Type mismatch in application:
  l.8:  k (fun x -> x + 1)
—— type `a` is not an instance of type `int`
  l.5:  let k: (forall a. a -> a) -> int
—— Note: constraint arises from reference:
  l.8:  k (fun x -> x + 1)
```

By contrast, here is the error generated by GHC, an industry-strength compiler that has supported bidirectionally-typed higher-rank polymorphism (a strictly easier feature to implement, which is sufficient for this example) for many years:

```
• Couldn't match expected type ‘Int’ with actual type ‘a’
  ‘a’ is a rigid type variable bound by
    a type expected by the context:
      forall a. a -> a
    at <interactive>:4:3-20
• In the first argument of ‘(+)’, namely ‘(x :: Int)’
  In the expression: (x :: Int) + 1
  In the first argument of ‘k’, namely ‘(\ x -> (x :: Int) + 1)’
• Relevant bindings include x :: a (bound at <interactive>:4:5)
```

The more tricky case to handle is when a type variable leaks out of its scope. SuperF keeps track of where leaks happen (which correspond to *type extrusions*) and reports them in the resulting error messages as possible locations where type annotations may be needed. For instance, assuming the same type for `k` as above, the term `(fun f → k (fun x → f x)) id` yields the following error:

```
[ERROR] Type error in application
  l.233:        (fun f -> k (fun x -> f x)) id
—— type variable `a` leaks out of its scope
  l.229:        def k: (forall a. a -> a) -> int
—— back into type variable `a`
  l.229:        def k: (forall a. a -> a) -> int
—— Adding a type annotation to any of the following terms
    may help resolve the problem
—— • this application:
  l.233:        (fun f -> k (fun x -> f x)) id
```

While the suggested annotation location does not refer specifically to parameter `f`, which is the most natural place to add a type annotation, it does refer to the relevant application where the leak takes place. Here is the rather verbose and unhelpful error produced by GHC:

```
• Couldn't match type ‘a0’ with ‘a’
  Expected: a -> a
    Actual: a0 -> a0
    because type variable ‘a’ would escape its scope
  This (rigid, skolem) type variable is bound by
    a type expected by the context:
      forall a. a -> a
    at <interactive>:5:11-22
• In the expression: f x
  In the first argument of ‘k’, namely ‘(\ x -> f x)’
  In the expression: k (\ x -> f x)
• Relevant bindings include
    x :: a (bound at <interactive>:5:14)
    f :: a0 -> a0 (bound at <interactive>:5:4)
```

A.5 Ω and the SRLC

Here we give an example constraint-solving run which involves the suspiciously recursive-looking criterion (SRLC).

Consider the problem of type checking term Ω, i.e., `(fun x → x x) (fun x → x x)`.

At some point in the main constraint-solving run, we get the constraint $\tau \le \beta$, where $\tau = \forall\alpha\{\alpha \le \alpha \to \alpha\}.\, \alpha \to \alpha$ and $\beta \le \beta \to \beta$

This leads to:

- add lower bound τ to β
- constrain $\tau \le \beta \to \beta$ (UB of β)
- instantiate τ to $\gamma_\alpha \to \gamma_\alpha$ where $\gamma_\alpha \le \gamma_\alpha \to \gamma_\alpha$ (we write γ_α to show that γ's shadow is α)
- constrain $\gamma_\alpha \to \gamma_\alpha \le \beta \to \beta$
- constrain $\beta \le \gamma_\alpha$ (LHS of function types)
- constrain $\beta \le \gamma_\alpha \to \gamma_\alpha$ (UB of γ_α)
- constrain $\tau \le \gamma_\alpha \to \gamma_\alpha$ (LB of β)

In turn, this leads to another round of exactly the same constraints until we reach:

- constrain $\tau \le \delta_\alpha \to \delta_\alpha$

But both pairs of types $(\tau,\ \delta_\alpha \to \delta_\alpha)$ and $(\tau,\ \gamma_\alpha \to \gamma_\alpha)$ have the same root $(\tau,\ \alpha \to \alpha)$, so the SRLC kicks in and aborts the constraint-solving run.

B EXTRA FORMAL DEFINITIONS AND EXAMPLES

We report on extra formal definitions as well as examples of the formal definitions here.

B.1 Box Erasure

Definition B.1. Box erasure, written $\tau \setminus x$, is defined inductively on the syntax of types and bounds contexts as follows:

$$
\begin{aligned}
\top \setminus x &\triangleq \top \\
\alpha \setminus x &\triangleq \alpha \\
(\tau_1 \to \tau_2) \setminus x &\triangleq (\tau_1 \setminus x) \to (\tau_2 \setminus x) \\
\forall V\{\Xi\}.\, \tau \setminus x &\triangleq \forall V\{(\Xi \setminus x)\}.\, (\tau \setminus x) \\
\langle \tau \rangle \setminus x &\triangleq \langle \tau \setminus x \rangle \\
\boxed{\tau}_x \setminus x &\triangleq \tau \setminus x \\
\boxed{\tau}_y \setminus x &\triangleq \boxed{\tau \setminus x}_y \qquad (x \neq y) \\
\epsilon \setminus x &\triangleq \epsilon \\
\Xi \cdot (\tau \le \sigma) \setminus x &\triangleq (\Xi \setminus x) \cdot ((\tau \setminus x) \le (\sigma \setminus x))
\end{aligned}
$$

Example B.2. Consider the following program:

```
test = fun f → fun x → f x

test (fun res → res) id
```

Note that to type the body of `test`, we need to wrap the type τ its parameter `x` into a box $\boxed{\tau}_x$, which means that parameter `f` must be a function that accepts such boxed arguments. Therefore, taking (arbitrarily, for the example) $\tau = \forall\gamma.\, \gamma \to \gamma$, the type of `test` could be, for instance, $\sigma = \forall\alpha.\, (\boxed{\tau}_x \to \alpha) \to \alpha$, or it could be $\sigma' = \forall\alpha\beta\{\beta \le \boxed{\tau}_x\}.\, (\beta \to \alpha) \to \alpha$. In both cases, all the x-branded boxes of this type can be removed at the *use site* of `test`, as in $\sigma \setminus x = \forall\alpha.\, (\tau \to \alpha) \to \alpha$ and $\sigma' \setminus x = \forall\alpha\beta\{\beta \le \tau\}.\, (\beta \to \alpha) \to \alpha$, which allows passing in the polymorphic identity function `id` for `x` and using the box-free result polymorphically.

Remark 2. Naturally, in all developments, we use Barendregt's convention and assume all variables in the program are distinct.

B.2 Syntactic and Semantic Acyclicity

A context is *syntactically acyclic* when its bounds graph is acyclic. By contrast, a context is *semantically acyclic* (or just *acyclic*) when all its bounds graph cycles (if it has any) are "harmless", meaning that traversing the bounds graph while following polarities will not lead to a cyclic path. The simplest example of that would be $\forall\alpha\{\alpha \le \top \to \alpha\}.\,\alpha$ — here, the upper bound of α, which is $\top \to \alpha$, is not reachable from traversing the bounds graph starting from α at a positive polarity (which is the body of the positive polymorphic type, always considered positive), so it is irrelevant and the bounds graph of this type is semantically acyclic despite being syntactically cyclic.

In general, we require all inferred types to be acyclic because supporting cyclic contexts would require support for full-fledged recursive types, which System F_{cc} does not yet possess.[27]

Definition B.3 (Acyclicity). Acyclicity is defined through the *reach* judgment, which is the smallest relation satisfying the following rules (where equality is taken to mean mutual set inclusion):

$$\boxed{\mathit{acyclic}(\Xi) \triangleq \mathit{for\ all}\ a,\ \{a^+, a^-\} \#\ \mathit{reach}^+(a, \Xi, 0)\ \mathit{and}\ a^- \notin \mathit{reach}^-(a, \Xi, 0)} \qquad \mathit{reach}^{\pm}(\top, \Xi, n) = \emptyset$$

$$\mathit{reach}^{\pm}(\tau \to \sigma, \Xi, n) = \mathit{reach}^{\mp}(\tau, \Xi, 1) \cup \mathit{reach}^{\pm}(\sigma, \Xi, 1) \qquad \mathit{reach}^{\pm}(\forall V\{\Sigma\}.\,\tau, \Xi, n) = \mathit{reach}^{\pm}(\tau, \Xi \cdot \Sigma, n)$$

$$\frac{}{a^{\pm} \in \mathit{reach}^{\pm}(a, \Xi, 1)} \qquad \frac{(\tau \le a) \in \Xi \quad b \in \mathit{reach}^+(\tau, \Xi, n)}{b \in \mathit{reach}^+(a, \Xi, n)} \qquad \frac{(a \le \tau) \in \Xi \quad b \in \mathit{reach}^-(\tau, \Xi, n)}{b \in \mathit{reach}^-(a, \Xi, n)}$$

Where we use $\mp$ as a shorthand for $+$ when $\pm$ is $-$ and $-$ when $\pm$ is $+$.

The "flag" parameter n can be 0 or 1 and indicates whether we have traversed at least one type constructor (if not, any cycle is a spurious "immediate" cycle).

Example B.4. Context $\Xi = (\alpha \to \beta) \to \gamma \le \alpha$ is not acyclic because we have:

$$\begin{aligned}
\mathit{reach}^+(\alpha, \Xi, 0) &= \mathit{reach}^+((\alpha \to \beta) \to \gamma, \Xi, 0) \\
&= \mathit{reach}^-(\alpha \to \beta, \Xi, 1) \cup \mathit{reach}^+(\gamma, \Xi, 1) \\
&= \mathit{reach}^+(\alpha, \Xi, 1) \cup \{\gamma^+\} \\
&= \{\alpha^+, \gamma^+\} \\
&\ni \alpha^+
\end{aligned}$$

Example B.5. Context $\Xi = \alpha \le \top \to \alpha$ is not acyclic because we have:

$$\begin{aligned}
\mathit{reach}^-(\alpha, \Xi, 0) &= \mathit{reach}^-(\top \to \alpha, \Xi, 0) \\
&= \mathit{reach}^+(\top, \Xi, 1) \cup \mathit{reach}^-(\alpha, \Xi, 1) \\
&= \{\alpha^-\} \\
&\ni \alpha^-
\end{aligned}$$

Definition B.6 (Syntactic acyclicity).

$$\mathit{synacyc}(\Xi) \triangleq \mathit{for\ all}\ a,\ \{a^+, a^-\} \#\ \mathit{reach}^+(a, \Xi, 0) \cup \mathit{reach}^-(a, \Xi, 0)$$

Example B.7. Context $\Xi = \alpha \le \alpha \to \top$ is semantically acyclic because $\mathit{reach}^+(\alpha, \Xi, 0) = \emptyset$ and $\mathit{reach}^-(\alpha, \Xi, 0) = \alpha^+$. It can be rewritten into the equivalent $\Xi' = \alpha \le \alpha' \wedge (\alpha' \to \top)$ (α' fresh), syntax sugar for $\Xi' = \alpha \le \beta', \beta' \le \alpha', \beta' \le \alpha' \to \top$ (α', β' fresh), which is more obviously semantically acyclic — but is still *syntactically* cyclic, as $\mathit{reach}^-(\alpha, \Xi', 0) = \alpha^+$.

[27] We conjecture that it would not be hard to extend F_{cc} with full recursive types, but we have not yet tried doing so.

B.3 Extrusion Safety Check

We first define *collect* that returns all the bounds nesting in the input type:

$$\begin{aligned}
\mathit{collect}(\top) &= \epsilon \\
\mathit{collect}(\alpha) &= \epsilon \\
\mathit{collect}(\forall \alpha.\, \tau^-) &= \mathit{collect}(\tau^-) \\
\mathit{collect}(\tau^\mp \to \tau^\pm) &= \mathit{collect}(\tau^\mp) \cdot \mathit{collect}(\tau^\pm) \\
\mathit{collect}(\forall V\{\Sigma\}.\, \tau^+) &= V \cdot \mathit{collect}(\tau^+) \cdot \overline{\mathit{collect}(\sigma^+)}^{(\sigma^+ \le \alpha) \in \Sigma} \cdot \overline{\mathit{collect}(\sigma^-)}^{(\alpha \le \sigma^-) \in \Sigma} \cdot \Sigma
\end{aligned}$$

We say $\tau^\pm$ is safe to extrude with the skolem set W if the type satisfies the following **X-ok** check:

Definition B.8. $W \vdash \tau^\pm$ **X-ok** $\triangleq$ for all $\alpha \in S$, $W \# \mathit{reach}^+(\alpha, \mathcal{B}(S), 1)$ or $W \# \mathit{reach}^-(\alpha, \mathcal{B}(S), 1)$ where $S = \mathit{collect}(\tau^\pm)$. We ignore polarities on type variables returned by $\mathit{reach}^\pm$.

C TRANSLATION TO F_{cc}

In this appendix we present the translation from our system to System F_{cc}. We will make two small adjustments to our system. First: we define the translation for an "erased" version of the system which doesn't allow either the $\boxed{\tau}_x$ or the $\langle \tau \rangle$ type form, doesn't allow the $(x : \tau)$ term form, doesn't have the following rules: T-Unbox, T-Asc, S-Unbox1, S-Unbox2, S-CongBound, S-CongFun, and has the following version of T-Abs:

$$\text{T-Abs}\ \frac{\Gamma \cdot (x : \tau_1) \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}$$

Clearly a valid derivation in the normal system is also valid in the erased system, after we erase $\boxed{\tau}_x$ and $\langle \tau \rangle$ to τ and $(x : \tau)$ to x.

Second, we will need to add type well-formedness to our system, since System F_{cc} requires type variables to be bound in context.

We extend contexts Γ and Ξ with type bindings: Γ may have the form $\Gamma \cdot \alpha$, likewise for Ξ.

Definition C.1. Quantifier consistency $\Xi \vdash \forall \overline{\alpha_i}^i \{\overline{B_j}^j\}$ ***cons.*** is defined as both $\Xi \vdash \forall \overline{\alpha_i}^i \{\overline{B_j}^j\}.\, \top$ **wf** and $\overline{\Xi \vdash \theta B_j}^j$, where $\theta = \overline{[\alpha_i \mapsto \tau_i]}^i$ for some $\overline{\tau_i}^i$.

Figure 8 shows rules of type well-formedness $\Gamma \vdash \tau$ **wf**. As every bounds context Ξ is also grammatically a typing context Γ, we will also write $\Xi \vdash \tau$ **wf**.

$\boxed{\Gamma \vdash \tau \textbf{ wf}}$

$$\text{W-Top}\ \frac{}{\Gamma \vdash \top \textbf{ wf}} \qquad \text{W-Var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \textbf{ wf}} \qquad \text{W-Fun}\ \frac{\Gamma \vdash \tau_1 \textbf{ wf} \quad \Gamma \vdash \tau_2 \textbf{ wf}}{\Gamma \vdash \tau_1 \to \tau_2 \textbf{ wf}}$$

$$\text{W-Forall}\ \frac{\overline{\Gamma \cdot \overline{\alpha_i}^i \vdash \sigma_1 \textbf{ wf} \quad \Gamma \cdot \overline{\alpha_i}^i \vdash \sigma_2 \textbf{ wf}}^{(\sigma_1 \le \sigma_2) \in \overline{B}} \qquad \Gamma \cdot \overline{\alpha_i}^i \cdot \overline{B_j}^j \vdash \tau \textbf{ wf}}{\Gamma \vdash \forall \overline{\alpha_i}^i \{\overline{B_j}^j\}.\, \tau \textbf{ wf}}$$

Fig. 8. Type well-formedness.

Since we now desire types to be well-formed, we need to make small changes to the typing and subtyping rules: variables need to be bound in contexts as appropriate and well-formedness needs to be required at certain points. Figures 9 and 10 show these changes.

$\boxed{\Gamma \vdash t : \tau}$

T-Abs
$$\frac{\Gamma \cdot (x : \tau_1) \vdash t : \tau_2 \qquad \Gamma \vdash \tau_1 \ \mathbf{wf}}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}$$

T-Forall
$$\frac{\Gamma \cdot V \cdot \Sigma \ \vdash \ t : \tau \qquad V \notin FV(\Gamma) \qquad \mathcal{B}(\Gamma) \vdash \forall V\{\Sigma\} \ \textbf{\textit{cons.}}}{\Gamma \vdash t : \forall V\{\Sigma\}.\, \tau}$$

Fig. 9. Adjustments to typing rules.

$\boxed{\Xi \vdash \tau \leq \tau}$

S-Top
$$\frac{\Xi \vdash \tau \ \mathbf{wf}}{\Xi \vdash \tau \leq \top}$$

S-VarRefl
$$\frac{\alpha \in \Xi}{\Xi \vdash \alpha \leq \alpha}$$

S-Forall-Cov
$$\frac{\Xi \vdash \forall V\{\Sigma\} \ \textbf{\textit{cons.}} \quad \Xi \cdot V \cdot \Sigma \vdash \tau \leq \sigma}{\Xi \vdash \forall V\{\Sigma\}.\, \tau \leq \forall V\{\Sigma\}.\, \sigma}$$

S-Forall-L
$$\frac{\Xi \vdash \forall V\{\Sigma\} \ \textbf{\textit{cons.}} \quad \overline{\Xi \vdash \tau_i \ \mathbf{wf}}^{\,i} \quad \Xi \vdash [\overline{\alpha_i \mapsto \tau_i}^{\,i}]\Sigma}{\Xi \vdash \forall \overline{\alpha_i}^{\,i}\{\Sigma\}.\, \tau \leq [\overline{\alpha_i \mapsto \tau_i}^{\,i}]\tau}$$

Fig. 10. Adjustments to subtyping rules.

The intent behind these changes is to allow the following *regularity* lemmas to hold.

Lemma C.2. *Let* Γ ***wf****. Then* $\Gamma \vdash t : \tau$ *implies that* $\Gamma \vdash \tau$ ***wf****.*

Lemma C.3. *Let* Γ ***wf****. Then* $\Gamma \vdash \tau \leq \sigma$ *implies that* $\Gamma \vdash \tau$ ***wf*** *and* $\Gamma \vdash \sigma$ ***wf****.*

In the following sections, we implicitly assume that all contexts are well-formed.

C.1 Translation

In this subsection, we show the exact translation to F_{cc}. We write variables standing for F_{cc} objects with a tilde: $\tilde{\alpha}, \tilde{\Gamma}$. We try keep the metavariables uniform, e.g. the metavariable for F_{cc} terms is $\tilde{t}$, not $\tilde{a}$; the only exception is when we reference variables used to instantiate F_{cc} rules, where we use the F_{cc} metavariable with a tilde (e.g. "Use CoerWeak with $\tilde{\Sigma} = \ldots$").
Where useful, we write F_{cc} judgments with a clarification on the turnstile : $\tilde{\Gamma} \vdash^{F_{cc}} \tilde{t} : \tilde{\tau}$.
F_{cc} includes a notion of type (kind, proposition) *equality* $\tilde{\tau} =^{F_{cc}} \tilde{\sigma}$, which should not be confused with the two types being the same. For instance, we have $\pi_1 \langle \tilde{\tau}, \tilde{\sigma} \rangle =^{F_{cc}} \tilde{\tau}$, but the two types are distinct objects. (This equality must be explicitly used by appropriate F_{cc} rules.) To avoid confusion, we add a clarification on the equality sign when using this form of equality.
We use Θ as a metavariable for mappings from our type variables to F_{cc} types.
We use $[\![\cdot]\!]$ to denote the translation function. Where useful, we add a metavariable in bold font in a superscript to clarify what sort of object is produced by the translation. We define the translation function as follows:

$$[\![\forall V\{\Sigma\}]\!]_\Theta \triangleq \{\tilde{\alpha} : [\![V]\!]^{\boldsymbol{\kappa}} \mid [\![\Sigma]\!]_{\Theta \circ [\![V ; \tilde{\alpha}]\!]^\Theta}\}$$

$$[\![\alpha \cdot V; \tilde{\tau}]\!]^{\Theta} \triangleq [\![\alpha; \tilde{\tau}]\!]^{\Theta} \circ [\![V; \pi_2\, \tilde{\tau}]\!]^{\Theta}$$
$$[\![\alpha; \tilde{\tau}]\!]^{\Theta} \triangleq [\alpha \mapsto \pi_1\, \tilde{\tau}]$$

$$[\![\alpha \cdot V]\!]^{\kappa} \triangleq \star \times [\![V]\!]^{\kappa}$$
$$[\![\alpha]\!]^{\kappa} \triangleq \star \times 1$$

$$[\![(\tau \le \sigma) \cdot \Sigma]\!]_{\Theta} \triangleq [\![\tau \le \sigma]\!]_{\Theta} \wedge [\![\Sigma]\!]_{\Theta}$$
$$[\![\epsilon]\!]_{\Theta} \triangleq \top$$

$$[\![\tau \le \sigma]\!]_{\Theta} \triangleq [[\![\tau]\!]_{\Theta} \triangleright [\![\sigma]\!]_{\Theta}]$$

$$[\![\tau \to \sigma]\!]_{\Theta} \triangleq [\![\tau]\!]_{\Theta} \to [\![\sigma]\!]_{\Theta}$$
$$[\![\forall V\{\Sigma\}.\, \tau]\!]_{\Theta} \triangleq \forall(\tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_{\Theta})\, [\![\tau]\!]_{\Theta \circ [\![V; \tilde{\alpha}]\!]^{\Theta}}$$
$$[\![\alpha]\!]_{\Theta} \triangleq \Theta(\alpha)$$
$$[\![\top]\!]_{\Theta} \triangleq \top$$

$$(\epsilon; \Theta) \sim (\varnothing; \Theta)$$

$$\frac{(\Gamma; \Theta_1') \sim (\tilde{\Gamma}; \Theta_2)}{((x : \tau) \cdot \Gamma; \Theta_1) \sim (x : [\![\tau]\!]_{\Theta}, \tilde{\Gamma}; \Theta_2)} \qquad \frac{(\Gamma; \Theta_1') \sim (\tilde{\Gamma}; \Theta_2)}{(V \cdot \Sigma \cdot \Gamma; \Theta_1) \sim (\tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_{\Theta_1}, \tilde{\Gamma}; \Theta_2)}$$

$$\Gamma \sim (\tilde{\Gamma}; \Theta) \triangleq (\Gamma; \mathrm{id}) \sim (\tilde{\Gamma}; \Theta)$$

C.2 Core theorem

In this subsection we show the core translation theorem. While we restrict our derivations to only use type variables bound in the context, this is not a significant loss of expressiveness: any derivation using the original rules can easily be transformed into a derivation obeying the well-formedness restriction by simply binding the type variables in the outermost context.

Theorem C.4. *If* $\vdash t : \tau$, *then* $\vdash^{F_{cc}} \tilde{t} : [\![\tau]\!]$ *for some* F_{cc} *term* $\tilde{t}$.

Proof. A special case of Lemma C.5. □

Lemma C.5. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $\Gamma \vdash t : \tau$ *implies that* $\tilde{\Gamma} \vdash^{F_{cc}} \tilde{t} : [\![\tau]\!]_{\Theta}$ *for some* F_{cc} *term* $\tilde{t}$.

Proof. By induction on the typing derivation.

Case T-Unit. Holds if () is translated to an arbitrary well-typed closed F_{cc} term, for instance the identity function, or a pair of identity functions.

Case T-Var, T-Abs, T-App. Trivial.

Case T-Subs. By TermCoer with empty $\tilde{\Sigma}$. Coercion can be shown with Lemma C.11.

Case T-Forall. By TermCoer with $\tilde{\Sigma} = \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]^{\kappa}_{\Theta}$. Typing can be shown by the IH. Coercion can be shown by CoerGen and by Lemma C.12. □

C.3 Coercion lemmas

The propositions below follow from a simple inspection of the appropriate definitions.

Proposition C.6. *If* $\Gamma \vdash \tau$ ***wf****, then* $FV(\tau) \subseteq FV(\Gamma)$.

Proposition C.7. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $\alpha \in FV(\Gamma)$ *implies that* $\tilde{\Gamma} \vdash \Theta(\alpha) : \star$.

Proposition C.8. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $\Gamma \vdash \tau$ ***wf*** *implies that* $\tilde{\Gamma} \vdash [\![\tau]\!]_\Theta : \star$.

We define an auxiliary translation which maps sequences of types to tuples of F_{cc} types.

$$[\![\tau \cdot \overline{\sigma}]\!]_\Theta^{(\tau \times \tau)} \triangleq \langle [\![\tau]\!]_\Theta, [\![\overline{\sigma}]\!]_\Theta^{(\tau \times \tau)} \rangle$$
$$[\![\tau]\!]_\Theta^{(\tau \times \tau)} \triangleq \langle [\![\tau]\!]_\Theta, \langle\rangle \rangle$$

Proposition C.9. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $\Gamma \vdash \forall V\{\overline{B_i}^i\}.\, \tau$ ***wf*** *and* $\overline{\Gamma \vdash \theta B_i}^i$ *(where* $dom(\theta) = \{ V \}$*) implies that* $\tilde{\Gamma} \vdash^{F_{cc}} [\![\theta V]\!]_\Theta^{(\tau \times \tau)} : [\![V]\!]^\kappa$.

Lemma C.10. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $(\tau \leq \sigma) \in \Gamma$ *implies that* $\tilde{\Gamma} \vdash^{F_{cc}} [\![\tau]\!]_\Theta \triangleright [\![\sigma]\!]_\Theta$.

Proof. By structural induction on Γ.

Case $\Gamma = \Gamma' \cdot (x : \tau')$**.** Then $(\tau \leq \sigma) \in \Gamma'$ and we conclude by the IH and F_{cc} weakening.

Case $\Gamma = \Gamma' \cdot V \cdot \Sigma$ **s.t.** $(\tau \leq \sigma) \in \Gamma'$**.** Likewise.

Case $\Gamma = \Gamma' \cdot V \cdot \Sigma$ **s.t.** $(\tau \leq \sigma) \in \Sigma$**.** Then $\tilde{\Gamma} = \tilde{\Gamma}', \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_{\Theta_0}$ and $\Theta = \Theta_0 \circ [\![V; \tilde{\alpha}]\!]^\Theta$. Since $(\tau \leq \sigma) \in \Sigma$, we have

$$[\![\forall V\{\Sigma\}]\!]_{\Theta_0} = \{\tilde{\beta} : [\![V]\!]^\kappa \mid \cdots \wedge [[\![\tau]\!]_{\Theta_0'} \triangleright [\![\sigma]\!]_{\Theta_0'}] \wedge \cdots\}$$

such that $\tilde{\beta}$ is not free in any type in the image of Θ_0 and $\Theta_0' = \Theta_0 \circ [\![V; \tilde{\beta}]\!]^\Theta$. Note that by TypeUnpack and TypeVar, we have $\tilde{\Gamma} \vdash \tilde{\alpha} : [\![V]\!]^\kappa$. By PropRes and PropAndProj, we then have $\tilde{\Gamma} \vdash^{F_{cc}} [\tilde{\beta} \mapsto \tilde{\alpha}]([[\![\tau]\!]_{\Theta'} \triangleright [\![\sigma]\!]_{\Theta'}])$. We observe that $[\tilde{\beta} \mapsto \tilde{\alpha}]([[\![\tau]\!]_{\Theta_0'} \triangleright [\![\sigma]\!]_{\Theta_0'}])$ is the same as $[[\![\tau]\!]_\Theta \triangleright [\![\sigma]\!]_\Theta]$ and we conclude with CoerProp. □

Lemma C.11. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $\Gamma \vdash \tau \leq \sigma$ *implies that* $\tilde{\Gamma} \vdash^{F_{cc}} [\![\tau]\!]_\Theta \triangleright [\![\sigma]\!]_\Theta$.

Proof. By induction on the subtyping derivation.

Case S-Top, S-VarRefl. By CoerTop and CoerRefl respectively.

Case S-Trans, S-Fun. By the IH and respectively either CoerTrans or CoerArr.

Case S-Hyp. By Lemma C.10.

Case S-Forall-R. By CoerWeak with $\tilde{\Sigma} = \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa$ and CoerGen with $\tilde{\sigma} = \tilde{\alpha}$.

Case S-Forall-Cov. Then $\tau = \forall V\{\Sigma\}.\, \tau'$ and $\sigma = \forall V\{\Sigma\}.\, \sigma'$. By CoerWeak with $\tilde{\Sigma} = \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa$ and CoerTrans, it suffices to show that both $\tilde{\Gamma}, \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa \vdash^{F_{cc}} [\![\tau]\!]_\Theta \triangleright [\![\sigma']\!]_\Theta$ and $\tilde{\Gamma} \vdash^{F_{cc}} (\tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa \vdash [\![\sigma']\!]_\Theta) \triangleright [\![\sigma]\!]_\Theta$. The former can be shown by the IH. The latter can be shown by CoerGen as long as we have $\tilde{\Gamma} \vdash [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa$. As a premise of S-Forall-Cov, we have $\Gamma \vdash \forall V\{\Sigma\}$, which means we can conclude by the same argument as in Lemma C.12. (Note that in this argument we use the IH rather than the entire Lemma C.11).

Case S-Forall-L. Similarly to the previous case: by CoerGen, using Lemma C.12 and the consistency premise of S-Forall-L.

Case S-Forall-Distr. Then $\tau = \forall V\{\Sigma\}.\, \tau_1 \to \tau_2$ and $\sigma = \tau_1 \to \forall V\{\Sigma\}.\, \tau_2$.
By CoerWeak with $\tilde{\Sigma} = \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa$ and CoerTrans, it suffices to show both of:

$$\tilde{\Gamma}, \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa \vdash^{F_{cc}} [\![\forall V\{\Sigma\}.\, \tau_1 \to \tau_2]\!]_\Theta \triangleright [\![\tau_1 \to \tau_2]\!]_\Theta$$
$$\tilde{\Gamma} \vdash^{F_{cc}} (\tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]_\Theta^\kappa \vdash [\![\tau_1 \to \tau_2]\!]_\Theta) \triangleright [\![\tau_1 \to \forall V\{\Sigma\}.\, \tau_2]\!]_\Theta$$

The former can be derived via CoerInst with $\tilde{\sigma} = \tilde{\alpha}$. The latter simplifies by CoerArr to:

$$\tilde{\Gamma}, \tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]^{\kappa}_{\Theta} \vdash^{F_{cc}} \tau_1 \triangleright \tau_1$$
$$\tilde{\Gamma} \vdash^{F_{cc}} (\tilde{\alpha} : [\![\forall V\{\Sigma\}]\!]^{\kappa}_{\Theta} \vdash \tau_2) \triangleright [\![\forall V\{\Sigma\}.\, \tau_2]\!]$$

Which hold respectively by reflexivity (CoerRefl) and CoerGen. Similarly to previous cases, the premise of CoerGen can be shown using Lemma C.12 and the consistency premise of S-Forall-Distr. □

Lemma C.12. *Let* $\Gamma \sim (\tilde{\Gamma}; \Theta)$. *Then* $\Gamma \vdash \forall V\{\Sigma\}$ ***cons.*** *implies that* $\tilde{\Gamma} \vdash^{F_{cc}} [\![\forall V\{\Sigma\}]\!]^{\kappa}_{\Theta}$.

Proof. Let $[\![\forall V\{\Sigma\}]\!]^{\kappa}_{\Theta} = \{\tilde{\alpha} : [\![V]\!]^{\kappa} \mid [\![\Sigma]\!]_{\Theta'}\}$. Our goal is, by definition, the same as $\tilde{\Gamma} \vdash^{F_{cc}} \exists [\![\forall V\{\Sigma\}]\!]^{\kappa}_{\Theta}$. By PropExi and TypePack, it suffices to show that we have $\tilde{\tau}$ such that $\tilde{\Gamma} \vdash^{F_{cc}} \tilde{\tau} : [\![V]\!]$ and $\tilde{\Gamma} \vdash^{F_{cc}} [\tilde{\alpha} \mapsto \tilde{\tau}]([\![\Sigma]\!]_{\Theta'})$. Let θ be the substitution from $\Gamma \vdash \forall V\{\Sigma\}$ and let $\overline{\tau}$ be its image. We pick $\tilde{\tau} = [\![\overline{\tau}]\!]^{(\tau \times \tau)}_{\Theta}$. Note that $\tilde{\tau}$ is a tuple of F_{cc} types.

We evidently have $\tilde{\Gamma} \vdash^{F_{cc}} \tilde{\tau} : [\![V]\!]$, since every type in $\overline{\tau}$ is encoded into a type of F_{cc} kind $\star$ and $\tilde{\tau} = [\![\overline{\tau}]\!]^{(\tau \times \tau)}_{\Theta'}$ is a type tuple of same size as V, meaning its F_{cc} kind is $[\![V]\!]$.

The remaining goal is $\tilde{\Gamma} \vdash^{F_{cc}} [\tilde{\alpha} \mapsto \tilde{\tau}]([\![\Sigma]\!]_{\Theta'})$. We have $[\tilde{\alpha} \mapsto \tilde{\tau}]([\![\Sigma]\!]_{\Theta'}) =^{F_{cc}} [\![\theta\Sigma]\!]_{\Theta}$ by F_{cc} type equality and by the definition of our translation: projections of $\tilde{\alpha}$ occur in $[\![\Sigma]\!]_{\Theta'}$ where variables from V occur in Σ. Since $\tilde{\tau} = [\![\overline{\tau}]\!]^{(\tau \times \tau)}_{\Theta} = [\![\theta V]\!]^{(\tau \times \tau)}_{\Theta}$, we have the desired F_{cc} type equality.

Therefore by PropEq and (repeated) PropAndPair, it suffices to show that $\tilde{\Gamma} \vdash^{F_{cc}} [\![\theta\sigma_1 \leq \theta\sigma_2]\!]_{\Theta}$ for every $(\sigma_1 \leq \sigma_2) \in \Sigma$. As we have $\Gamma \vdash \forall V\{\Sigma\}$, for each such $(\sigma_1 \leq \sigma_2)$ we have $\Gamma \vdash \theta\sigma_1 \leq \theta\sigma_2$, and we can conclude by (repeated) Lemma C.11. □

D TYPE INFERENCE CORRECTNESS PROOFS

In this section, we sketch the proof of the core correctness theorem for type inference. First, we need to slightly modify I-Abs to account for the well-formedness requirement that our System F_{cc} translation introduced. Concretely, we add the binding for α in the premise:

$$\text{I-Abs}\quad \frac{\alpha \ \textbf{\textit{fresh}} \qquad \Gamma \cdot \alpha \cdot (x : \alpha) \vdash t : \tau^{+} \Rightarrow \Delta \qquad V \supseteq FV(\Gamma) \qquad V \,\#\, FV(\Delta) \qquad V, \epsilon \vdash \epsilon \gg \Delta \gg \Xi \qquad split_V(uproot(\Xi)) = (\Xi_0, \overline{\gamma}, \Xi_1)}{\Gamma \vdash \lambda x.\, t : \forall \overline{\gamma}\{\Xi_1\}.\, \alpha \to \tau^{+} \Rightarrow \Xi_0}$$

We also need standard weakening and permutation lemmas.

Lemma D.1 (Weakening). *Let* Γ, Γ' *be well-formed contexts such that* $\Gamma = \Gamma_1 \cdot \Gamma_2$ *and* $\Gamma' = \Gamma_1 \cdot \Gamma_0 \cdot \Gamma_2$. *Then:*

- $\Gamma \vdash t : \tau$ *implies that* $\Gamma' \vdash t : \tau$
- $\Gamma \vdash \tau \leq \sigma$ *implies that* $\Gamma' \vdash \tau \leq \sigma$
- $\Gamma \vdash \forall V\{\Sigma\}$ ***cons.*** *implies that* $\Gamma' \vdash \forall V\{\Sigma\}$ ***cons.***
- $\Gamma \vdash \tau$ ***wf*** *implies that* $\Gamma' \vdash \tau$ ***wf***

Proof. By mechanical inspection of the definition of typing, subtyping, quantifier consistency, type well-formedness. No judgment inspects the number of bindings in the context. □

Lemma D.2 (Permutation). *Let* Γ, Γ' *be well-formed contexts such that* Γ' *is a permutation of* Γ. *Then:*

- $\Gamma \vdash t : \tau$ *implies that* $\Gamma' \vdash t : \tau$
- $\Gamma \vdash \tau \leq \sigma$ *implies that* $\Gamma' \vdash \tau \leq \sigma$

- $\Gamma \vdash \forall V\{\Sigma\}$ ***cons.*** *implies that* $\Gamma' \vdash \forall V\{\Sigma\}$ ***cons.***
- $\Gamma \vdash \tau$ ***wf*** *implies that* $\Gamma' \vdash \tau$ ***wf***

Proof. By mechanical inspection of the definition of typing, subtyping, quantifier consistency, type well-formedness. No judgment inspects the order of bindings in the context. □

We can now restate and prove Theorem 4.5:

Theorem (Soundness of Type Inference). *If* $\vdash t : \tau^+ \Rightarrow \Delta$ *and* $\vdash \epsilon \gg \Delta \gg \Xi'$*, then we have* $\vdash t : \forall V\{\Xi\}.\, \tau^+$*, where* $\Xi = \mathit{uproot}(\Xi')$*, and* $V = FV(\Xi)$*.*

Proof. By Lemma D.3 we have $V \cdot \Xi \vdash t : \tau^+$. By the second conclusion of Theorem 4.7 we have $\vdash \forall V\{\Xi\}$ ***cons.***. (The constraining premise we have has empty skolems, so the skolem replacement is the identity.) We can then conclude by T-Forall. □

Lemma D.3. *Let* $\Gamma \vdash t : \tau^+ \Rightarrow \Delta$*. Then* $V, \epsilon \vdash \epsilon \gg \Delta \gg \Xi'$ *implies that* $\Gamma \cdot \overline{\gamma} \cdot \Xi \vdash t : \tau^+$*, where* $\Xi = \mathit{uproot}(\Xi')$ *and* $\overline{\gamma} = FV(\Xi) \setminus V$*.*

Proof. By induction on the type inference derivation. All cases but I-Abs are trivial, as $\Xi \vdash \Delta$ by the first conclusion of Theorem 4.7. (Again, the constraining premise on which we use the theorem has empty skolems.)

In I-Abs, we have:

$$t = \lambda x.\, t_0 \quad \tau^+ = \forall \overline{\gamma}\{\Xi_1\}.\, \alpha \to \tau_0^+ \quad \Delta = \Xi_0$$
$$\Gamma \cdot \alpha \cdot (x : \alpha) \vdash t_0 : \tau_0^+ \Rightarrow \Delta' \quad V', \epsilon \vdash \epsilon \gg \Delta' \gg \Xi'$$
$$\text{where} \quad V' \supseteq FV(\Gamma) \quad V' \# FV(\Delta) \quad \mathit{split}_{V'}(\Xi') = (\Xi_0, \overline{\gamma}, \Xi_1)$$

We desire $\Gamma \cdot \Xi_0 \vdash \lambda x.\, t_0 : \forall \overline{\gamma}\{\Xi_1\}.\, \alpha \to \tau_0^+$, which by T-Forall and T-Abs reduces to

$$\Xi_\Gamma \cdot \Xi_0 \vdash \forall \overline{\gamma}\{\Xi_1\}\ \textbf{\textit{cons.}}$$
$$\Gamma \cdot \Xi_0 \cdot \overline{\gamma} \cdot \Xi_1 \cdot (x : \alpha) \vdash t_0 : \tau_0^+$$

The former holds by weakening the second conclusion of Theorem 4.7. The latter holds by permutating the conclusion of the IH. □

We now turn to prove the correctness of the subtype constraint solving, which is by far the most interesting and tricky part of the type inference algorithm.

D.1 Termination of constraining

We now justify the termination of subtype constraint solving. This property trivially implies the termination of type inference as a whole since the type inference rules are defined by structural induction on program terms. We only provide a sketch of the proof.

We need auxiliary definitions. The following definition states an invariant maintained by normal constraining derivations.

Definition D.4. V, W, Ξ is a coherent constraining context triple if V is disjoint from W and for all $B \in \Xi$, at least one hand of B is a type variable a such that $a \notin V \cdot W$.

We say such a triple is rooted in $\overline{\alpha}$ if $\overline{\alpha} \subseteq \{\mathit{roots}(a) \mid a \in FV(V \cdot W) \cup FV(\Xi_0)\}$.

Definition D.5. Constraints Δ are *simplifiable* (denoted $\gg \Delta \gg$) in roots $\overline{\alpha}$ if for all coherent constraining context triples V, W, Ξ rooted in $\overline{\alpha}$ there is a finite derivation of $V, W \vdash \Xi \gg \Delta \gg \Sigma^?$ for some $\Sigma^?$.

Lemma D.6 (Worklist extension). *Let $V, W \vdash \Xi_0 \gg \Delta_2 \gg \Sigma_2^?$. Then $\gg \Delta_1 \gg$ in roots $\{roots(a) \mid a \in FV(V \cdot W) \cup FV(\Xi_0)\}$ implies that $V, W \vdash \Xi_0 \gg \Delta_1 \cdot \Delta_2 \gg \Sigma_3^?$ for some $\Sigma_3^?$.*

Proof. Follows from the definitions. □

As a terminology note, we say a constraining derivation "solves" a constraint C if it has a subderivation such that its goals Δ end in C.

Theorem (Termination of Constraining). *For all Δ and all coherent constraining context triples V, W, ϵ, there is a finite derivation of $V, W \vdash \epsilon \gg \Delta \gg \Xi^?$ for some $\Xi^?$.*

Proof Sketch. The intuition behind the sketch is that constraining, if viewed as an algorithm, always terminates (i.e., there is a finite constraining derivation), since it either reduces the constraints it still has to traverse or it reduces the number of roots in Δ which still can be visited. Concretely, we sketch how to show that the premises of each rule are smaller according to some measure.

The measure that we use is an ordered pair: its first element is the amount of roots in Δ which still can be visited according to their corresponding ϕ, and its second element is the size of Δ.

The proof is by induction on the size of Δ, calculated using our measure. The proof's cases are organized according to which constraining rule applies first. Note that in all cases, if the derivation of any premise outputs **err**, we use C-Fail instead of the rule from the current case.

Rule C-Empty trivially terminates, while rules C-Top, C-Skip, C-Fun and C-Forall-L all have a premise with a strictly smaller Δ.

Rules C-Rigid-L and C-Rigid-R add the result of extrusion Σ to the constraints. By the IH, we have $\gg \Delta \gg$ in current roots. We sketch how to show that we also have $\gg B \gg$ in current roots for all $B \in \Sigma$.

For all $B \in \Sigma$, one hand of B is a rigid variable, and another is a flexible variable. This means that only C-Skip, C-Flex-L and C-Flex-R can apply. If B is not skipped and C-Rigid-L or C-Rigid-R applies, then B is added to ϕ and all the constraints added to Δ have a rigid variable on at least one hand. The latter means that either a basic rule (C-Top, C-VarRefl or C-Skip) or one of the variable rules (C-Flex-L, C-Flex-R, C-Rigid-L, C-Rigid-R) applies to each such new constraint. The variable rules cannot apply endlessly, since every constraint they add to Δ is also added to ϕ (indirectly in the case of C-Rigid-L and C-Rigid-R). Then, we have $\gg B \gg$ in current roots as desired.

Hence, by Lemma D.6 we have $\gg \Sigma \gg$ and $\gg \Delta \cdot \Sigma \gg$, which lets us conclude the case.

Rules C-Flex-L and C-Flex-R include the SRLC premise. By the IH we have $\gg \Delta \gg$. The SRLC ensures that each of the constraints added in the premise of C-Flex-L and C-Flex-R is smaller according to our measure. Hence, for each such constraint C we also have $\gg C \gg$, letting us conclude the case by Lemma D.6.

Finally, we consider rule C-Forall-R. Constraint $\tau^+ \leq^\phi \sigma^-$ is smaller than $\Delta \cdot (\tau^+ \leq^\phi \forall\alpha.\tau^-)$, so by the IH there is a finite derivation of $V', W \cdot \beta_\alpha \vdash \epsilon \gg \tau^+ \leq^\phi \sigma^- \gg \Xi^?$ for some $\Xi^?$. If $\Xi^? =$ **err** then we use C-Fail, otherwise $\Xi^? = \Xi$.

Now we focus on the second constraining premise. By the IH, there is a finite derivation of $V, W \cdot \beta_\alpha \vdash \Xi_0 \gg \tau^+ \leq^\phi \sigma^- \gg \ldots$ (the result is not relevant except that we may need to propagate **err** with C-Fail). We will refer to this derivation as $\mathfrak{J}_2$ and to first premise's derivation as $\mathfrak{J}_1$. We will sketch how to show there is a finite derivation of $V, W \cdot \beta_\alpha \vdash \Xi_0 \gg outer_{V'}(\Xi) \gg \ldots$, which can be constructed from $\mathfrak{J}_1$ and $\mathfrak{J}_2$. The intuition is that temporarily freezing some type variables, like in the first premise's derivation $\mathfrak{J}_1$, delays solving constraints which can be solved on the spot, like in $\mathfrak{J}_2$. In a sense, $\mathfrak{J}_2$ merely solves more constraints than $\mathfrak{J}_1$. In particular, all flexible variables appearing in $\mathfrak{J}_1$ can be treated as also appearing in $\mathfrak{J}_2$ (we can act as though both derivations always pick the same variable names). Additionally, note that the only constraints solved in $\mathfrak{J}_1$

which are not solved in $\mathfrak{J}_2$ are constraints on rigid variables in $V' \setminus V$ solved by C-Rigid-L or C-Rigid-R.

We sketch how to construct the desired derivation. Every $B \in \mathit{outer}_{V'}(\Xi)$ was added by a subderivation of $\mathfrak{J}_1$ ending in C-Rigid-L or C-Rigid-R. WLOG assume the former for a particular $B = a \le \tau_a$. Then that subderivation of $\mathfrak{J}_1$ solves some $B_1 = a \le \tau_{a,1}$.

If $a \in V$, then $\mathfrak{J}_2$ also solves B_1 via C-Rigid-L. Otherwise, if a is an approximant for b in $\mathfrak{J}_1$, then (and only then) B_1 was added to the constraints in $\mathfrak{J}_1$ by C-Rigid-L or C-Rigid-R and no subderivation in $\mathfrak{J}_2$ solves the exact same goal. We proceed by distinguishing two cases based on how the constraints added by C-Rigid-L and C-Rigid-R must be solved.

If $\tau_{a,1}$ is also an approximant a' in $\mathfrak{J}_1$, then a and a' are the lower and upper approximants for b. The desired subderivation of $a \le a'$ can be constructed because all constraints on a and a' solved in $\mathfrak{J}_1$ are instead constraints on b, a flexible variable, in $\mathfrak{J}_2$. Hence, $\mathfrak{J}_2$ solves the constraints arising from the desired derivation needing to solve $a \le a'$.

Otherwise, if $\tau_{a,1}$ is not an approximant in $\mathfrak{J}_1$, then it is an upper bound of b and $\mathfrak{J}_2$ solves $b \le \tau_{a,1}$.

Finally, if a is not an approximant in $\mathfrak{J}_1$, then $\mathfrak{J}_2$ has a subderivation which solves $a \le \tau_{a,1}$ via C-Flex-L.

In all these cases, in the desired derivation we need to construct a (sub)derivation solving $a \le \tau_a$. Compared to $\tau_{a,1}$ in $\mathfrak{J}_2$, some type variables a_a may have been replaced by approximants when B_1 was extruded in $\mathfrak{J}_1$. If such type variables are flexible in the desired derivation, i.e., absent from V, then they approximate temporarily frozen variables (ones in $V' \setminus V$). Hence, all constraints on these approximants extruded by $\mathfrak{J}_1$ corresponds to constraints on their underlying type variable solved by $\mathfrak{J}_2$. While the constraints may not necessarily be solved in the same order in $\mathfrak{J}_2$ and in the desired derivation, the order of constraints does not truly matter. As Appendix **??** explains, if we reorder some goals then the derivation as a whole still traverses and solves the same constraints.

Hence, there is a finite derivation of $V, W \cdot \beta_\alpha \vdash \Xi_0 \gg \mathit{outer}_{V'}(\Xi) \gg \ldots$, which can also be derived in W because $\beta_\alpha \notin FV(\Xi)$. Therefore, by Lemma D.6 there is also a finite derivation of $V, W \vdash \Xi_0 \gg \Delta \cdot \mathit{outer}_{V'}(\Xi) \gg \Sigma^?$ for some $\Sigma^?$. If $\Sigma^? = \mathbf{err}$ then we use C-Fail, otherwise we can conclude with C-Forall-R. □

D.2 Soundness of constraining

We now justify that subtype constraining, when it succeeds, leads to valid and consistent subtyping, which allows us to use it in the proof of type inference soundness.

D.2.1 Full propagation. To prove type inference soundness, we define a new inductive relation on bounds context, called *full propagation*, which is a restricted form of subtyping without a transitivity rule and which corresponds closely with what a constraint closure and resolution algorithm should compute as its output context. We show that our subtype constraining algorithm implies full propagation in its output context and then we show that full propagation itself implies that the context's subtyping relationships can be derived *and* that they are consistent. We show consistency by exhibiting a solution to the bounds in the form of the *lower-bounds union* (LBU).

Definition D.7 (Full propagation). We say V, Σ is *fully-propagated* if $V, W, \epsilon \vdash \Sigma \dashv \Sigma$, where the judgment is defined in Figure 11. We only consider well-formed full propagation derivations where $V \# W$.

D.2.2 Concrete types.

$\boxed{V, W, \Sigma \vdash \tau^+ \le \tau^- \dashv \Sigma'}$ *We define* $V, W, \Sigma \vdash \overline{B} \dashv \Sigma' \triangleq \overline{V, W, \Sigma \vdash B \dashv \Sigma'}$

$$\textsc{P-Top}\ \frac{}{V, W, \Sigma \vdash \tau^+ \le \top \dashv \Sigma'} \qquad \textsc{P-VarRefl}\ \frac{}{V, W, \Sigma \vdash \alpha \le \alpha \dashv \Sigma'} \qquad \textsc{P-Fun}\ \frac{V, W, \Sigma \vdash \sigma^+ \le \tau^- \dashv \Sigma' \quad V, W, \Sigma \vdash \tau^+ \le \sigma^- \dashv \Sigma'}{V, W, \Sigma \vdash \tau^- \to \tau^+ \le \sigma^+ \to \sigma^- \dashv \Sigma'}$$

$$\textsc{P-Rigid-L}\ \frac{\alpha \in V \quad \rho = [\overline{\beta- \mapsto \sigma^-,\ \beta+ \mapsto \sigma^+}^{\beta \in FV(\tau^-) \setminus V}] \quad (\alpha \le \rho\tau^-) \in \Sigma' \quad W \vdash \tau^- \textbf{ X-ok} \quad FV(\rho\tau^-) \subseteq V \quad \overline{V, W, \Sigma \vdash (\sigma^- \le \beta) \cdot (\beta \le \sigma^+) \dashv \Sigma'}^{\beta \in FV(\tau^-) \setminus V}}{V, W, \Sigma \vdash \alpha \le \tau^- \dashv \Sigma'}$$

$$\textsc{P-Rigid-R}\ \frac{\alpha \in V \quad \rho = [\overline{\beta- \mapsto \sigma^-,\ \beta+ \mapsto \sigma^+}^{\beta \in FV(\tau^+) \setminus V}] \quad (\rho\tau^+ \le \alpha) \in \Sigma' \quad W \vdash \tau^+ \textbf{ X-ok} \quad FV(\rho\tau^+) \subseteq V \quad \overline{V, W, \Sigma \vdash (\sigma^- \le \beta) \cdot (\beta \le \sigma^+) \dashv \Sigma'}^{\beta \in FV(\tau^+) \setminus V}}{V, W, \Sigma \vdash \tau^+ \le \alpha \dashv \Sigma'}$$

$$\textsc{P-Skip}\ \frac{\alpha \notin V \cdot W \quad (\tau^+ \le \sigma^-) \in \{(\alpha \le \sigma^-), (\tau^+ \le \alpha)\} \quad (\tau^+ \le \sigma^-) \notin \Sigma' \quad (\tau^+ \le \sigma^-) \in \Sigma}{V, W, \Sigma \vdash \tau^+ \le \sigma^- \dashv \Sigma'} \qquad \textsc{P-Flex}\ \frac{\alpha \notin V \cdot W \quad (\tau^+ \le \sigma^-) \in \Sigma' \quad (\tau^+ \le \sigma^-) \in \{(\alpha \le \sigma^-), (\tau^+ \le \alpha)\} \quad \overline{V, W, \Sigma \vdash CLB_V(\tau^+, \Sigma') \le \sigma^{-\prime} \dashv \Sigma'}^{\sigma^{-\prime} \in CUB_V(\sigma^-, \Sigma')}}{V, W, \Sigma \vdash \tau^+ \le \sigma^- \dashv \Sigma'}$$

$$\textsc{P-Forall-L}\ \frac{V, W, \Sigma \vdash [\ \overline{\alpha \mapsto \pi}\ ]\Xi \dashv \Sigma' \quad V, W, \Sigma \vdash [\ \overline{\alpha \mapsto \pi}\ ]\tau^+ \le \sigma^- \dashv \Sigma'}{V, W, \Sigma \vdash \forall \overline{\alpha}\{\Xi\}.\, \tau^+ \le \sigma^- \dashv \Sigma'} \qquad \textsc{P-Forall-R}\ \frac{\beta\ \textbf{\textit{fresh}} \quad V' \supseteq FV(\tau^+ \le \sigma^-) \setminus W \quad V' \# W \cdot \beta \quad \mathit{acyclic}(\Xi) \quad V', W \cdot \beta, \epsilon \vdash \tau^+ \le [\alpha \mapsto \beta]\sigma^- \dashv \Xi \quad V', W \cdot \beta, \epsilon \vdash \Xi \dashv \Xi \quad V, W, \Sigma \vdash \mathit{outer}_{V'}(\Xi) \dashv \Sigma'}{V, W, \Sigma \vdash \tau^+ \le \forall \alpha.\, \sigma^- \dashv \Sigma'}$$

Fig. 11. The polarized full-propagation rules. Note that in P-Rigid-L and P-Rigid-R, σ^- and σ^+ are both positive and negative, i.e. System F types. In P-Forall-L, π can be either positive or negative as long as τ^+ remains a valid positive type after substitutions.

Definition D.8 (Left-Concrete Types). A left-concrete type is *not* a flexible type variable (that may be under universal quantifiers).

$$\top\ \textbf{\textit{concrete}}^+_V \qquad \tau_1 \to \tau_2\ \textbf{\textit{concrete}}^+_V \qquad \frac{a \in V}{a\ \textbf{\textit{concrete}}^+_V} \qquad \frac{\tau^+\ \textbf{\textit{concrete}}^+_V}{\forall \overline{\alpha}\{\Sigma\}.\, \tau^+\ \textbf{\textit{concrete}}^+_V}$$

Definition D.9 (Right-Concrete Types). A right-concrete type is *not* a flexible type variable (that may be under spurious universal quantifiers such as $\forall \alpha.\beta$ where β is flexible).

$$\top\ \textbf{\textit{concrete}}^-_V \qquad \tau_1 \to \tau_2\ \textbf{\textit{concrete}}^-_V \qquad \frac{a \in V}{a\ \textbf{\textit{concrete}}^-_V} \qquad \frac{\alpha \in FV(\tau^-)}{\forall \alpha.\, \tau^-\ \textbf{\textit{concrete}}^-_V} \qquad \frac{\tau^-\ \textbf{\textit{concrete}}^-_V}{\forall \alpha.\, \tau^-\ \textbf{\textit{concrete}}^-_V}$$

Definition D.10 (Concrete Lower Bounds (CLB)). Assume universal types are well-formed and no name collision of universal quantifiers. $CLB_V(\tau, \Xi) = go^+_V(\tau, \Xi, \epsilon)$. go^+ returns τ itself if τ is

left-concrete or the union of *left-concrete* lower bounds of τ in Ξ that can be transitively reached.

$$go^+_V(\tau^+, \Xi, W) = \tau^+ \qquad \text{if } \tau^+ \ \textbf{\textit{concrete}}^+_V$$
$$go^+_V(a, \Xi, W) = \bigvee \overline{go^+_V(\tau^+, \Xi, W \cdot a)}^{(\tau^+ \le a) \in \Xi, \tau^+ \notin (W \cdot a)} \qquad \text{otherwise}$$
$$go^+_V(\forall \overline{\alpha}\{\Sigma\}.\, \tau^+, \Xi, W) = \forall \overline{\alpha}\{\Sigma\}.\, go^+_V(\tau^+, \Sigma \cdot \Xi, W) \qquad \text{otherwise}$$

Example D.11. $CLB_\epsilon(a, (\forall \alpha\beta\gamma\{\alpha \to \beta \le \gamma, \beta \to \alpha \le \gamma, \alpha \le \beta\}.\, \gamma \le a)) = \forall \alpha\beta\gamma\{\alpha \to \beta \le \gamma, \beta \to \alpha \le \gamma, \alpha \le \beta\}.\, (\alpha \to \beta) \vee (\beta \to \alpha)$.

Example D.12. Let $\Xi = (\forall \beta\{\Xi'\}.\, \beta \le \alpha) \cdot (\tau \le \alpha) \cdot (\sigma' \le \gamma)$ and $\Xi' = (\alpha \le \beta) \cdot (\sigma \le \beta) \cdot (\gamma \le \beta)$ where τ, σ, σ' are left-concrete. Then $CLB_\epsilon(\alpha, \Xi) = \tau \vee \forall \beta\{\Xi'\}.\, (\sigma \vee \sigma')$.

Definition D.13 (Concrete Upper Bounds (CUB)). $CUB_V(a, \Xi) = go^-_V(a, \Xi, \epsilon)$. go^- calculates a set of all *right-concrete* upper bounds of a in Ξ that can be transitively reached.

$$go^-_V(\tau^-, \Xi, W) = \tau^- \qquad \text{if } \tau^- \ \textbf{\textit{concrete}}^-_V$$
$$go^-_V(a, \Xi, W) = \overline{go^-_V(\tau^-, \Xi, W \cdot a)}^{(a \le \tau^-) \in \Xi, \tau^- \notin (W \cdot a)} \qquad \text{otherwise}$$
$$go^-_V(\forall \alpha.\, \tau^-, \Xi, W) = go^-_V(\tau^-, \Xi, W) \qquad \text{otherwise}$$

The following properties of CLB and CUB are obvious:

Proposition D.14. *$CLB_V(\tau, \Xi)$ is always defined.*

Proposition D.15. *$CUB_V(\tau, \Xi)$ is always defined.*

Proposition D.16. $CLB_V(\tau, \Xi) = CLB_V(\tau, \Xi \cdot (\tau \le \sigma))$

Proposition D.17. $CUB_V(\tau, \Xi) = CUB_V(\tau, \Xi \cdot (\sigma \le \tau))$

D.2.3 Lower Bound Union (LBU).

Definition D.18 (Declarative Lower Bound Union). For all Ξ and V and set of proper substitutions ρ, if every substitution $(\alpha \mapsto \tau) \in \rho$ satisfies $\tau = \rho CLB_V(\alpha, \Xi)$, then $\rho \in isLBU^\Xi_V$.

Definition D.19 (Algorithmic Lower Bound Union). We obtain the algorithmic lower bound union by $mkLBU(\Xi, V, \epsilon)$ which is a function call that returns a list of substitutions:

$$mkLBU(\Xi, V, W) = \overline{\alpha \mapsto [mkLBU(\Xi, V, W \cdot \alpha)]CLB_V(\alpha, \Xi)}^{\alpha \in FV(\Xi) \setminus (V \cdot W)}$$

Proposition D.20. *If acyclic(Ξ), then:*
(1) $mkLBU(\Xi, V, \epsilon) \in isLBU^\Xi_V$
(2) $mkLBU(\Xi, V, \epsilon)$ is always defined.

Proof sketch. For each type variable $a \in FV(\Xi) \setminus V$, its CLB can always be calculated because each type variable is only accessed at most once in the bounds graph. Due to the acyclicity condition of Ξ, there are no cyclic occurrences of α transitively reachable in the concrete lower bounds of α, so no infinite substitution of LBU would happen when recursively substituting occurrences of other type variables in the concrete lower bounds, and we can disregard α in the recursive computation of lower bound union of other type variables. □

Proposition D.21. *If acyclic(Ξ), then there always exists a proper substitution ρ in $isLBU^\Xi_V$.*

Proof. We provide $mkLBU(\Xi, V, \epsilon)$ as the witness. □

Notation: we write LBU^Ξ_V as an arbitrary instance in $isLBU^\Xi_V$.

D.2.4 Transitive reachability of bounds.

Definition D.22 (Transitively reachable bounds). Given a rigid type variable set V, we denote a flexible type variable α transitively reaches a lower bound τ^+ in context Σ through a set of flexible type variables $\overline{\beta}$ as $\tau^+ \leq_V^* \alpha \in \Sigma \sim \overline{\beta}$.

$$\frac{\tau^+ \leq \alpha \in \Sigma \qquad \alpha \notin V}{\tau^+ \leq_V^* \alpha \in \Sigma \sim \alpha} \qquad\qquad \frac{\tau^+ \leq_V^* \beta \in \Sigma \sim \overline{\gamma} \qquad \beta \leq \alpha \in \Sigma \qquad \alpha, \beta \notin V}{\tau^+ \leq_V^* \alpha \in \Sigma \sim \overline{\gamma} \cdot \beta}$$

Similarly, we denote an upper bound that is transitively reachable as $\alpha \leq_V^* \tau^- \in \Sigma \sim \overline{\beta}$.

D.2.5 Propagation context adjustment.

Lemma D.23. *If for all $a \notin V$, $\tau_i^- \in CUB_V(\tau^-, \Xi')$, we have $V, W, \Xi \cdot (a \leq \tau^-) \vdash CLB_V(a, \Xi') \leq \tau_i^- \dashv \Xi'$ and $acyclic(\Xi \cdot \Xi' \cdot (a \leq \tau^-))$, then $V, W, \Xi \vdash CLB_V(a, \Xi') \leq \tau_i^- \dashv \Xi' \cdot (a \leq \tau^-)$.*

Lemma D.24. *If for all $a \notin V$, $\tau_i^- \in CUB_V(a, \Xi')$, we have $V, W, \Xi \cdot (\tau^+ \leq a) \vdash CLB_V(\tau^+, \Xi') \leq \tau_i^- \dashv \Xi'$ and $acyclic(\Xi \cdot \Xi' \cdot (\tau^+ \leq a))$, then $V, W, \Xi \vdash CLB_V(\tau^+, \Xi') \leq \tau_i^- \dashv \Xi' \cdot (\tau^+ \leq a)$.*

Proof sketch of D.23. Essentially, the types on both sides of the propagation relationship to prove are all concrete. The typical case is when two function types, say $l_1 \to l_2$ and $r_1 \to r_2$, are on both sides. Due to the acyclicity condition, a would never appear (or be transitively reached) in the positive/negative positions of l_2/r_2 and negative/positive positions of l_1/r_1. For the positions otherwise, it is possible that a appears due to syntactically cyclic upper bounds, e.g. $l_1 \to l_2 \leq a \leq \tau^- = a \to \top$, we need to propagate $a \leq l_1$ in $\Xi' \cdot (a \leq \tau^-)$, while $a \leq \tau^-$ does not create any new concrete lower bound for a so it is equivalent to propagating it in Ξ', which is already known. Note that it is impossible that $a \leq l_1$ is propagated by P-Skip when $l_1 = \tau^- = a \to \top$ because this violates the acyclicity condition. So generally, we can inductively reconstruct the whole derivation of the assumed propagation relations with the contexts adjusted as that of the goals. For D.24, the proof idea is the same, while there is no syntactic cyclicity in lower bounds. □

Lemma D.25. *If for all τ^+ and $a \notin V$:*

(1) $V, W, \Xi \cdot (\tau^+ \leq a) \vdash \Xi' \dashv \Xi'$
(2) $V, W, \Xi \cdot (\tau^+ \leq a) \vdash \sigma^+ \leq \sigma^- \dashv \Xi'$
(3) $acyclic(\Xi \cdot \Xi' \cdot (\tau^+ \leq a))$
(4) For all $\tau_i^- \in CUB_V(a, \Xi')$, we have $V, W, \Xi \cdot (\tau^+ \leq a) \vdash CLB_V(\tau^+, \Xi') \leq \tau_i^- \dashv \Xi'$

then $V, W, \Xi \vdash \sigma^+ \leq \sigma^- \dashv \Xi' \cdot (\tau^+ \leq a)$.

Lemma D.26. *If for all τ^- and $a \notin V$:*

(1) $V, W, \Xi \cdot (a \leq \tau^-) \vdash \Xi' \dashv \Xi'$
(2) $V, W, \Xi \cdot (a \leq \tau^-) \vdash \sigma^+ \leq \sigma^- \dashv \Xi'$
(3) $acyclic(\Xi \cdot \Xi' \cdot (a \leq \tau^-))$
(4) For all $\tau_i^- \in CUB_V(\tau^-, \Xi')$, we have $V, W, \Xi \cdot (a \leq \tau^-) \vdash CLB_V(a, \Xi') \leq \tau_i^- \dashv \Xi'$

then $V, W, \Xi \vdash \sigma^+ \leq \sigma^- \dashv \Xi' \cdot (a \leq \tau^-)$.

Proof sketch of D.26 (D.25 is symmetric). By induction on the full propagation derivation from (2). Intuitively, this lemma means bounds are still consistent even with a new upper bound to propagate after checking that the new bound is consistent and propagated to the original bounds.

Cases P-Top, P-VarRefl, P-Fun Immediate.

Cases P-Rigid-L/P-Rigid-R WLOG we consider P-Rigid-L. We are to prove $V, W, \Xi \vdash b \leq \sigma^- \dashv \Xi' \cdot (a \leq \tau^-)$. Since $a \notin V$ and $b \in V$, $a \neq b$. We conclude with IH.

Case P-Skip By assumption, we have $(\sigma^+ \le \sigma^-) \in (\Xi\cdot(a \le \tau^-))$ and $(\sigma^+ \le \sigma^-) \notin \Xi'$. Then the only case we need to consider is when $(\sigma^+ \le \sigma^-) = (a \le \tau^-)$, otherwise we conclude with P-Skip since $(\sigma^+ \le \sigma^-) \in \Xi$.
By P-Flex, we need to prove for all $\tau_i^- \in CUB_V(\tau^-, \Xi'\cdot(a \le \tau^-))$, we have $V, W, \Xi \vdash CLB_V(a, \Xi'\cdot(a \le \tau^-)) \le \tau_i^- \dashv \Xi'\cdot(a \le \tau^-)$.
It is immediate to see that $CLB_V(a, \Xi'\cdot(a \le \tau^-)) = CLB_V(a, \Xi')$ since $a \le \tau^-$ does not introduce new lower bounds for a.
Consider τ^-. If τ^- is right-concrete, then $CUB_V(\tau^-, \Xi'\cdot(a \le \tau^-)) = CUB_V(\tau^-, \Xi') = \tau^-$.
Otherwise, we still have $CUB_V(\tau^-, \Xi'\cdot(a \le \tau^-)) = CUB_V(\tau^-, \Xi')$ because τ^-, as a flexible type variable (possibly under spurious universal quantifiers), could only be visited once.
Now, the goal is to prove for all $\tau_i^- \in CUB_V(\tau^-, \Xi')$, $V, W, \Xi \vdash CLB_V(a, \Xi') \le \tau_i^- \dashv \Xi'\cdot(a \le \tau^-)$.
By assumption, we have for all $\tau_i^- \in CUB_V(\tau^-, \Xi')$, $V, W, \Xi\cdot(a \le \tau^-) \vdash CLB_V(a, \Xi') \le \tau_i^- \dashv \Xi'$. We conclude with D.23.

Case P-Flex
For this case we have:

$$\frac{b \notin V \qquad (\sigma^+ \le \sigma^-) \in \{(b \le \sigma^-), (\sigma^+ \le b)\} \qquad (\sigma^+ \le \sigma^-) \in \Xi' \qquad \overline{V, W, \Xi\cdot(a \le \tau^-) \vdash CLB_V(\sigma^+, \Xi') \le \sigma_i^- \dashv \Xi'}^{\sigma_i^- \in CUB_V(\sigma^-, \Xi')}}{V, W, \Xi\cdot(a \le \tau^-) \vdash \sigma^+ \le \sigma^- \dashv \Xi'}$$

We denote CLB types $CLB_V(\sigma^+, \Xi')$ as C_l, $CUB_V(\sigma^-, \Xi')$ as C_r, $CLB_V(\sigma^+, \Xi'\cdot(a \le \tau^-))$ as C_l', and $CUB_V(\sigma^-, \Xi'\cdot(a \le \tau^-))$ as C_r'.
By P-Flex, the goal is to prove for all $\sigma_i^- \in C_r'$, we have $V, W, \Xi \vdash C_l' \le \sigma_i^- \dashv \Xi'\cdot(a \le \tau^-)$.
We discuss the concreteness of τ^- and σ^+/σ^- to discover how the new bound $a \le \tau^-$ influences the relationships between C_r/C_r' and C_l/C_l'. We use the notation $\equiv$ to describe the equivalency between CLB types, i.e. a CLB type can be replaced by an equivalent one in propagation judgments.
Consider the following cases:

- $(\sigma^+ \le \sigma^-) = (b \le \sigma^-)$.
  By IH, for all $\sigma_i^- \in C_r$, we have $V, W, \Xi \vdash C_l \le \sigma_i^- \dashv \Xi'\cdot(a \le \tau^-)$.
  By D.23, for all $\tau_i^- \in CUB_V(\tau^-, \Xi')$, we have $V, W, \Xi \vdash CLB_V(a, \Xi') \le \tau_i^- \dashv \Xi'\cdot(a \le \tau^-)$ $(*)$.
  Consider the concreteness of τ^- and σ^-:
  - τ^- ***concrete***$_V^-$ and σ^- ***concrete***$_V^-$. We conclude with the IH since $C_l = C_l', C_r = C_r'$.
  - τ^- ***concrete***$_V^-$ and $\neg(\sigma^-$ ***concrete***$_V^-)$. We have $C_l = C_l'$. If σ^- cannot transitively reach a as an upper bound in $\Xi'\cdot(a \le \tau^-)$, then we conclude with IH since σ^- does not introduce new upper bounds to propagate, i.e. $C_r = C_r'$. Otherwise, σ^- transitively reaches a, i.e. in the bounds graph (with the new bound $(a \le \tau^-)$ marked red):
    $$b \le \sigma^- \le^* a \le \tau^-$$
    then $C_r' = C_r \cup \{\tau^-\}$ since $CUB_V(\tau^-, \Xi'\cdot(a \le \tau^-)) = \{\tau^-\}$. For all type in C_r, we conclude with IH. We want $V, W, \Xi \vdash C_l \le \tau^- \dashv \Xi'\cdot(a \le \tau^-)$, which is implied by $(*)$ since a transitively has the lower bound b.
  - $\neg(\tau^-$ ***concrete***$_V^-)$ and σ^- ***concrete***$_V^-$. We have $C_r = C_r' = \{\sigma^-\}$. If b cannot transitively reach τ^-, then we conclude with IH since $(a \le \tau^-)$ does not introduce new lower bounds to propagate, i.e. $C_l = C_l'$. Otherwise, b transitively reaches τ^-, i.e.:
    $$a \le \tau^- \le^* b \le \sigma^-$$

then $C_l' \equiv C_l \vee CLB_V(a, \Xi')$. The bounds C_l being less than each type in C_r are already propagated by IH. We want $V, W, \Xi \vdash CLB_V(a, \Xi') \leq \sigma^- \dashv \Xi' \cdot (a \leq \tau^-)$, which is implied by $(*)$ since τ^- transitively has the upper bound σ^-.

- $\neg(\tau^- \ \boldsymbol{concrete}_V^-)$ and $\neg(\sigma^- \ \boldsymbol{concrete}_V^-)$. For the cases when (1) both τ^-/σ^- could not transitively reach b/a, (2) only one of τ^-/σ^- transitively reaches b/a, we apply the same reasoning in the former cases. When both τ^-/σ^- transitively reaches b/a,

$$\begin{array}{ccc} a & \leq & \tau^- \\ {\scriptstyle *}\vee\!| & & |\wedge_{\scriptstyle *} \\ \sigma^- & \geq & b \end{array}$$

$C_r' = C_r \cdot CUB_V(\tau^-, \Xi')$ and $C_l' \equiv C_l \vee CLB_V(a, \Xi')$. By $(*)$, those relationships are propagated since adding $a \leq \tau^-$ would unify a, τ^-, b, and σ^-.

- $(\sigma^+ \leq \sigma^-) = (\sigma^+ \leq b)$. We apply analogous reasoning to the case above. We discuss the concreteness of τ^- and σ^+/σ^- and whether b/σ^- can transitively reach a/τ^- to discover how the new bound $a \leq \tau^-$ influences the relationships between C_r/C_r' and C_l/C_l'.

Case P-Forall-L By IH.

Case P-Forall-R By the IH on the *third* constraining premise; other premises are unchanged. □

Lemma D.27. *If for all τ^- and $FV(\tau^- \leq \tau^+) \subseteq V$ and $V, W, \Xi \vdash \sigma^+ \leq \sigma^- \dashv \Xi'$ then $V, W, \Xi \vdash \sigma^+ \leq \sigma^- \dashv \Xi' \cdot (\tau^+ \leq \tau^-)$.*

D.2.6 Correctness of CLB and CUB.

Lemma D.28. *For all τ^- and $a \notin V$ and Σ and Σ', if*

(1) for all $\sigma^+ \leq b \in \Sigma$ and $b \leq \sigma^- \in \Sigma'$, we have $V, W, \Sigma \vdash \sigma^+ \leq \sigma^- \dashv \Sigma'$; and
(2) for all $\tau^+ \leq a \in \Sigma'$, we have $V, W, \Sigma \vdash \tau^+ \leq \tau^- \dashv \Sigma'$

then for all $\tau_i^- \in CUB_V(\tau^-, \Sigma')$, we have $V, W, \Sigma \vdash CLB_V(a, \Sigma') \leq \tau_i^- \dashv \Sigma'$.

Lemma D.29. *For all τ^+ and $a \notin V$ and Σ and Σ', if*

(1) for all $b \leq \sigma^- \in \Sigma$ and $\sigma^+ \leq b \in \Sigma'$, we have $V, W, \Sigma \vdash \sigma^+ \leq \sigma^- \dashv \Sigma'$; and
(2) for all $a \leq \tau^- \in \Sigma'$, we have $V, W, \Sigma \vdash \tau^+ \leq \tau^- \dashv \Sigma'$

then for all $\tau_i^- \in CUB_V(a, \Sigma')$, we have $V, W, \Sigma \vdash CLB_V(\tau^+, \Sigma') \leq \tau_i^- \dashv \Sigma'$.

Proof sketch of D.28 (D.29 is similar). By the definition of CLB,

$$\begin{aligned} CLB_V(a, \Sigma') &= go_V^+(a, \Sigma', \epsilon) \\ &= \bigvee \overline{go_V^+(\tau^+, \Sigma', \{a\})}^{(\tau^+ \leq a) \in \Sigma', \tau^+ \notin \{a\}} \end{aligned}$$

By P-Forall-L, for each $(\tau^+ \leq a) \in \Sigma'$ and $\tau^+ \notin \{a\}$, we need to prove for all $\tau_i^- \in CUB_V(\tau^-, \Sigma')$, we have $V, W, \Sigma \vdash go_V^+(\tau^+, \Sigma', \{a\}) \leq \tau_i^- \dashv \Sigma'$. We conclude by invoking Lemma D.30 with the assumption that CLB and CUB always terminate and the following properties: for all $b \leq a \in \Xi'$ where $a, b \notin V$, if $\tau^+ \leq_V^* b \in \Xi' \sim \overline{\beta}$ then $V, W, \Xi \vdash \tau^+ \leq a \dashv \Xi'$. □

Lemma D.30. *For all τ^-, τ_0^+, $\overline{\alpha}$, $a \notin V$, Σ, and Σ', if*

(1) $\tau_0^+ \leq_V^* a \in \Sigma' \sim \overline{\alpha}$; and
(2) for all $\sigma^+ \leq b \in \Sigma$ and $b \leq \sigma^- \in \Sigma'$, we have $V, W, \Sigma \vdash \sigma^+ \leq \sigma^- \dashv \Sigma'$; and
(3) for all τ^+ such that $\tau^+ \leq_V^* a \in \Sigma' \sim \overline{\beta}$, we have $V, W, \Sigma \vdash \tau^+ \leq \tau^- \dashv \Sigma'$

then for all $\tau_i^- \in CUB_V(\tau^-, \Sigma')$, if there exists k and l such that the recursion depth of the CUB call is k and the recursion depth of the following go^+ call is l, then we have $V, W, \Sigma \vdash go_V^+(\tau_0^+, \Sigma', \overline{\alpha}) \le \tau_i^- \dashv \Sigma'$.

Proof sketch of D.30. We prove by induction on l and case analysis on the concreteness of each τ^+ (IH1):

- $l = 0$ and τ^+ $\boldsymbol{concrete}_V^+$.
  Then $go_V^+(\tau^+, \Sigma', \{a\}) = \tau^+$. By assumption, we have $V, W, \Sigma \vdash \tau^+ \le \tau^- \dashv \Sigma'$ $(*)$. We now perform induction on k (IH2).
  - $k = 0$, i.e. $\tau^- = \tau_i^-$ and τ^- $\boldsymbol{concrete}_V^-$, we conclude immediately.
  - $k > 0$, i.e. $\neg(\tau^-$ $\boldsymbol{concrete}_V^-)$ and τ^- transitively has the concrete upper bound τ_i^-. We then perform induction on the propagation assumption $(*)$ (IH3).
    Case P-Top Impossible as $\top$ is concrete.

    Cases P-VarRefl, P-Fun, P-Rigid-R Impossible since τ^- is not concrete.

    Case P-Rigid-L $\tau^+ = b$, $b \in V$, $(b \le \tau') \in \Sigma'$, $FV(\tau') \in V$, and $V, W, \Sigma \vdash \tau' \le \tau^- \dashv \Sigma'$. By $FV(\tau') \in V$, τ' $\boldsymbol{concrete}_V^+$. By IH3, for all $\tau_i^- \in CUB_V(\tau^-, \Sigma')$, we have $V, W, \Sigma \vdash \tau' \le \tau_i^- \dashv \Sigma'$. We conclude with P-Rigid-L.

    Case P-Skip Since τ^+ $\boldsymbol{concrete}_V^+$, we can only have $\tau^- = b$, $b \notin V$, $(\tau^+ \le b) \notin \Sigma'$, and $(\tau^+ \le b) \in \Sigma$. There is a τ' that is non-right-concrete and transitively has the concrete upper bound τ_i^- and $(b \le \tau') \in \Sigma'$. By assumption (1), we have $V, W, \Sigma \vdash \tau^+ \le \tau' \dashv \Sigma'$. We conclude with IH2 since τ' goes through $k - 1$ non-right-concrete types to reach τ_i^-.

    Case P-Flex Immediate.

    Case P-Forall-L By IH3.

    Case P-Forall-R $\tau^- = \forall b.\sigma^-$ and $b \notin FV(\sigma^-)$ and $\forall b.\, \sigma^-$ is not concrete. Notice that $\forall b.\, \sigma^-$ is essentially a flexible type variable (b') under spurious universal quantifiers, we have $(\tau^+ \le b') \in \Xi$ and also $(\tau^+ \le b') \in outer_{V'}(\Xi)$ since $V' \supseteq FV(\tau^+ \le b')$. We conclude with IH3.
- $l > 0$ and $\tau^+ = \forall\overline{\alpha}\{\Xi\}.\, \tau'$ and $\neg(\tau'$ $\boldsymbol{concrete}_V^+)$.
  Then $go_V^+(\forall\overline{\alpha}\{\Xi\}.\, \tau', \Sigma', \{a\}) = \forall\overline{\alpha}\{\Xi\}.\, go_V^+(\tau', \Sigma' \cdot \Xi, \{a\})$.
  By assumption, $V, W, \Sigma \vdash \forall\overline{\alpha}\{\Xi\}.\, \tau' \le \tau^- \dashv \Sigma'$.
  We apply similar reasoning as the former case ($l = 0$) by induction on the number of non-right-concrete types that CUB traverses and the propagation derivation. In case P-Forall-L, we have Ξ propagated after substitution. We use this and P-Forall-L to conclude.
- $l > 0$ and $\tau^+ = b$ and $\neg(b$ $\boldsymbol{concrete}_V^+)$.
  Then $go_V^+(b, \Sigma', \{a\}) = \bigvee \overline{go_V^+(\tau', \Sigma', \{a, b\})}^{(\tau' \le b) \in \Sigma', \tau' \notin \{a,b\}}$. We conclude with P-Forall-L and IH1 since each τ' traverses $l - 1$ non-left-concrete types to reach a left-concrete type. □

Definition D.31. $\phi \subseteq \Xi \triangleq \overline{B \in \Xi}^{B \in \phi}$. Note that $\in$ does not look past $\triangleright$ in ϕ.

D.2.7 *Constraining implies full propagation.*

Lemma D.32. *If $V, W \vdash \Xi_0 \gg \Delta \gg \Xi_0'$ and $\overline{\phi \subseteq \Xi_0}^{(\tau^+ \le^\phi \tau^-) \in \Delta}$ and $acyclic(\Xi \cdot \Xi')$ and $V \# W$, then:*

(1) for all $\tau^+ \le^\phi \tau^- \in \Delta$, we have $V, W, \Xi \vdash uproot(\tau^+ \le \tau^-) \dashv \Xi'$.

(2) (a) for all $\tau^+ \le a \in \Xi$ *and* $a \le \tau^- \in \Xi'$ *where* $a \notin V$*, we have* $V, W, \Xi \vdash \tau^+ \le \tau^- \dashv \Xi'$*; and*
(b) for all $a \le \tau^- \in \Xi$ *and* $\tau^+ \le a \in \Xi'$ *where* $a \notin V$*, we have* $V, W, \Xi \vdash \tau^+ \le \tau^- \dashv \Xi'$*.*
(3) $V, W, \Xi \vdash \Xi' \dashv \Xi'$

where $uproot(\Xi_0) = \Xi$ *and* $uproot(\Xi_0') = \Xi'$*.*

Proof. By induction on the constraining derivation. The cases where the proof is sketched are clearly marked. We implicitly *uproot* types with roots when they are used in propagation judgments.

Cases C-Empty, C-Top, C-VarRefl, C-Fun, C-Fail Immediate.

Case C-Skip (2) and (3) holds by the IH. For (1), WLOG we consider $\Delta = \Delta' \cdot (a \le \tau^-)$. If $(a \le \tau^-) \in \Xi'$ then we conclude with the IH's (3). Otherwise, we conclude with P-Skip since $(a \le \tau^-) \in \phi$ and $\phi \subseteq \Xi$.

Case C-Flex-L/C-Flex-R WLOG we consider C-Flex-L.

$$\frac{B = (a \le \tau^-) \quad root(B) \notin roots(\phi) \quad acyclic(\Xi \cdot B) \quad a \notin V \cdot W \quad V, W \vdash \Xi \cdot B \gg \Delta \cdot \overline{(\tau^+ \le^{B \cdot \phi} \tau^-)}^{(\tau^+ \le a) \in \Xi} \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (a \le^{\phi} \tau^-) \gg \Xi' \cdot B}$$

(1) The goal is $V, W, \Xi \vdash \Delta \cdot (a \le \tau^-) \dashv \Xi' \cdot (a \le \tau^-)$.
To propagate $(a \le \tau^-)$, since $(a \le \tau^-) \in (\Xi' \cdot (a \le \tau^-))$ and $a \notin V$, by P-Flex, we want:
for all $\tau_i^- \in CUB_V(\tau^-, \Xi' \cdot (a \le \tau^-))$, we have

$$V, W, \Xi \vdash CLB_V(a, \Xi' \cdot (a \le \tau^-)) \le \tau_i^- \dashv \Xi' \cdot (a \le \tau^-).$$

By the IH's (2) (b), we have $\overline{V, W, \Xi \cdot (a \le \tau^-) \vdash \tau^+ \le \tau^- \dashv \Xi'}^{(\tau^+ \le a) \in \Xi'}$.
By D.28, using the above as the second premise and the IH's (2) (a) as the first premise, we have for all $\tau_i^- \in CUB_V(\tau^-, \Xi')$, we have

$$V, W, \Xi \cdot (a \le \tau^-) \vdash CLB_V(a, \Xi') \le \tau_i^- \dashv \Xi' (**).$$

By D.26 and the IH's (3), we adjust the context Ξ' such that for all $\tau_i^- \in CUB_V(\tau^-, \Xi')$, we have $V, W, \Xi \vdash CLB_V(a, \Xi') \le \tau_i^- \dashv \Xi' \cdot (a \le \tau^-)$ $(*)$.
Since τ^- would only be accessed once by CUB no matter its concreteness, $CUB_V(\tau^-, \Xi' \cdot (a \le \tau^-)) = CUB_V(\tau^-, \Xi')$. Since a would only be accessed once by CLB, $CLB_V(a, \Xi' \cdot (a \le \tau^-)) = CLB_V(a, \Xi')$. We conclude that $(a \le \tau^-)$ is propagated with $(*)$ by the two equations above.
To propagate Δ, we conclude with the IH and $(**)$ and D.26 that adjusts the context.

(2) By the IH and D.26, we have the property for Ξ, $(a \le \tau^-)$ and Ξ', and we want the same for Ξ and Ξ', $(a \le \tau^-)$. This requires $\overline{V, W, \Xi \vdash \tau^+ \le \tau^- \dashv \Xi' \cdot (a \le \tau^-)}^{(\tau^+ \le a) \in \Xi}$.
By the IH's (1), we have $\overline{V, W, \Xi \cdot (a \le \tau^-) \vdash \tau^+ \le \tau^- \dashv \Xi'}^{(\tau^+ \le a) \in \Xi}$.
Then by D.26 (using $(**)$ and the IH for the premises),
we have $\overline{V, W, \Xi \vdash \tau^+ \le \tau^- \dashv \Xi' \cdot (a \le \tau^-)}^{(\tau^+ \le a) \in \Xi}$, which concludes.

(3) The goal is $V, W, \Xi \vdash \Xi' \cdot (a \le \tau^-) \dashv \Xi' \cdot (a \le \tau^-)$, which requires propagating Ξ' and $(a \le \tau^-)$ under adjusted contexts. We start from the IH's (3) and reason like in (1).

Case C-Rigid-L/C-Rigid-R WLOG we consider C-Rigid-L.

$$\text{C-Rigid-L}\ \frac{a \in V \quad V, W \vdash \tau^- \rightsquigarrow (\Sigma, \sigma^-) \quad V, W \vdash \Xi \gg \Delta \cdot \Sigma \gg \Xi'}{V, W \vdash \Xi \gg \Delta \cdot (a \le^{\phi} \tau^-) \gg \Xi' \cdot (a \le \sigma^-)}$$

(1) The goal is to show $V, W, \Xi \vdash \Delta \cdot (a \le \tau^-) \dashv \Xi' \cdot (a \le \sigma^-)$.
The first subgoal is to show that Δ is propagated.
By IH's (1), we have $V, W, \Xi \vdash \Delta \cdot \Sigma \dashv \Xi'$ where Σ is the output bounds of extrusion.
Since σ^- is the output of extrusion, $FV(\sigma^-) \subseteq V$. Thus we can adjust the output context and obtain $V, W, \Xi \vdash \Delta \cdot \Sigma \dashv \Xi' \cdot (a \le \sigma^-)$ since $(a \le \sigma^-)$ creates no new type to propagate by D.27, which concludes this subgoal.
The second subgoal is to show that $V, W, \Xi \vdash a \le \tau^- \dashv \Xi' \cdot (a \le \sigma^-)$.
We invert the extrusion derivation. For case X-1, the goal is trivial since $\sigma^- = \tau^-$ and $FV(\tau^-) \subseteq V$ which allows us to conclude with P-Rigid-L by picking ρ as empty. For case X-2, $\sigma^- = \rho^- \tau^-$ where ρ is the polarized substitution that extrude all free types variables in $FV(\tau^-) \setminus V$. We conclude with P-Rigid-L by using ρ as the substitution and $V, W, \Xi \vdash \Sigma \dashv \Xi' \cdot (a \le \sigma^-)$. We can reuse the **X-ok** premise of X-2. Note that since $V \# W$, we have $W \cap FV(\tau^-) = W \cap (FV(\tau^-) \setminus V)$. For all $b \in W \cap FV(\tau^-)$ we trivially have $V, W, \Xi \vdash (\bot \le b) \cdot (b \le \top) \dashv \Xi' \cdot (a \le \sigma^-)$ for the positive and negative approximants of the skolem b.
(2) We conclude with the IH's (2) since $a \in V$.
(3) The goal is to show $V, W, \Xi \vdash \Xi' \cdot (a \le \sigma^-) \dashv \Xi' \cdot (a \le \sigma^-)$.
By IH's (3), we have $V, W, \Xi \vdash \Xi' \dashv \Xi'$. As in (1), $FV(\sigma^-) \subseteq V$ and we can adjust the output context to obtain $V, W, \Xi \vdash \Xi' \dashv \Xi' \cdot (a \le \sigma^-)$.
By P-Rigid-L, we have $V, W, \Xi \vdash a \le \sigma^- \dashv \Xi' \cdot (a \le \sigma^-)$ by picking empty ρ.

Case C-Forall-L/C-Forall-R All properties hold by the IH. Note that in the first premise of C-Forall-R, the cache is locked so the free part of the cache is empty, which is a subset of the empty input context, i.e. $\triangleright\phi \subseteq \epsilon$. This allows us to invoke the IH on that premise. □

Corollary D.33. *If $V, W \vdash \epsilon \gg \Delta \gg \Xi'_0$, then $V, W, \epsilon \vdash \Xi' \dashv \Xi'$ where $uproot(\Xi'_0) = \Xi'$.*

Proof. Corollary of D.32. □

D.2.8 *Full propagation implies subtyping and consistent bounds.*

Notation: we write $V, W \vdash \sigma \le \tau \dashv \Sigma$ as a shorthand for $V, W, \epsilon \vdash \sigma \le \tau \dashv \Sigma$.

Lemma D.34. *If $V, \overline{\alpha} \vdash \tau \le \sigma \dashv \Sigma$ and $V, \overline{\alpha} \vdash \Sigma \dashv \Sigma$ and acyclic(Σ), then we have*

$$outer_V(\Sigma) \vdash LBU^{inner_V(\Sigma)}_{V \cdot \overline{\alpha}}(\tau \le \sigma).$$

Proof sketch. By induction on the full propagation derivation. Let $\rho = LBU^{inner_V(\Sigma)}_{V \cdot \overline{\alpha}}$.

Case P-Top. By S-Top.

Case P-VarRefl. By reflexivity of subtyping.

Case P-Fun. By IH and S-Fun.

Case P-Rigid-L.

$$\frac{a \in V \qquad \theta = [\overline{b^- \mapsto \sigma^-,\ b^+ \mapsto \sigma^+}^{b \in FV(\tau^-) \setminus V}] \qquad (a \le \theta\tau^-) \in \Sigma \qquad W \vdash \tau^- \textbf{ X-ok} \qquad FV(\theta\tau^-) \subseteq V \qquad \overline{V, \overline{\alpha} \vdash (\sigma^- \le b) \cdot (b \le \sigma^+) \dashv \Sigma}^{b \in FV(\tau^-) \setminus V}}{V, \overline{\alpha} \vdash a \le \tau^- \dashv \Sigma}$$

Since we have both $FV(\theta\tau^-) \subseteq V$ and $a \in V$, it is the case that $(a \le \theta\tau^-) \in outer_V(\Sigma)$ and also that $\rho(a \le \theta\tau^-) = a \le \theta\tau^-$.
By S-Hyp, we have $outer_V(\Sigma) \vdash \rho(a \le \theta\tau^-)$. Then by the IH, we have the following.

$$\overline{outer_V(\Sigma) \vdash \rho(\sigma^- \le b) \cdot \rho(b \le \sigma^+)}^{b \in FV(\tau^-) \setminus V}$$

With the polarized substitutions θ and the subtyping derivations above,
we can have $outer_V(\Sigma) \vdash \rho(\theta\tau^- \le \tau^-)$ by D.38, which allows us to conclude with S-Trans.

Case P-Rigid-R. Symmetric to the former case.
Case P-Skip Impossible.
Case P-Flex.

$$\frac{a \notin V \qquad \overline{\frac{(\tau \le \sigma) \in \{(a \le \sigma), (\tau \le a)\}}{V, \overline{\alpha} \vdash CLB_V(\tau, \Sigma) \le \sigma' \dashv \Sigma}^{\sigma' \in CUB_V(\sigma, \Sigma)}} \qquad (\tau \le \sigma) \in \Sigma}{V, \overline{\alpha} \vdash \tau \le \sigma \dashv \Sigma}$$

- $(\tau \le \sigma) = (a \le \sigma)$. We want $outer_V(\Sigma) \vdash \rho(a \le \sigma)$.
  After expanding ρa, we want:

  $$outer_V(\Sigma) \vdash \rho CLB_V(a, inner_V(\Sigma)) \le \rho\sigma$$

  By the IH, we have $(*)$:

  $$\overline{outer_V(\Sigma) \vdash \rho CLB_V(a, \Sigma) \le \rho\sigma'}^{\sigma' \in CUB_V(\sigma, \Sigma')}$$

  At this point we can conclude with the following auxilliary lemma.

  Lemma D.35. *Let* $\overline{outer_V(\Sigma) \vdash \rho CLB_V(a, \Sigma) \le \rho\sigma'}^{\sigma' \in CUB_V(\sigma, \Sigma')}$, *where* $\rho = LBU^{inner_V(\Sigma)}_{V \cdot \overline{\alpha}}$.
  Then $outer_V(\Xi) \vdash \rho(a \le \sigma)$.

  Proof. By induction on σ. We inspect its concreteness at the same time.
  - σ ***concrete***$^-$. Then $CUB_V(\sigma, \Sigma') = \sigma$. The lemma's premise is sufficient to conclude, since $FV(outer_V(\Sigma)) \subseteq V$ and CLB never traverses rigid type variables, i.e., $CLB_V(a, \Sigma) = CLB_V(a, inner_V(\Sigma))$.
  - $\neg(\sigma$ ***concrete***$^-)$ and $\sigma = b$ and $a \notin V$. Then $\rho\sigma = \rho CLB_V(b, inner_V(\Sigma))$. It is intuitive to see that $outer_V(\Sigma) \vdash \rho CLB_V(a, inner_V(\Sigma)) \le \rho CLB_V(b, inner_V(\Sigma))$ when $(a \le b) \in \Sigma'$ because a is b's immediate lower bound, any concrete lower bound of the former is included in the union of concrete lower bounds of the latter.
  - $\neg(\sigma$ ***concrete***$^-)$ and $\sigma = \forall\alpha.\, \sigma_0$ and $\alpha \notin FV(\sigma_0)$. We conclude by the IH on σ and $\alpha \notin FV(\sigma')$, as $CUB_V(\sigma, \Sigma') = CUB_V(\sigma_0, \Sigma')$. □

- $(\tau \le \sigma) = (\tau \le a)$. The intuition is similar to the second bullet point above: as a direct subtype of a, the concrete lower bounds of τ must be included by that of a.

Case P-Forall-L.

$$\frac{V, \overline{\alpha} \vdash [\,\overline{\beta \mapsto \pi'}\,]\Xi \dashv \Sigma \qquad V, \overline{\alpha} \vdash [\,\overline{\beta \mapsto \pi'}\,]\tau \le \sigma \dashv \Sigma}{V, \overline{\alpha} \vdash \forall\overline{\beta}\{\Xi\}.\, \tau \le \sigma \dashv \Sigma}$$

We want $outer_V(\Sigma) \vdash \rho(\forall\overline{\beta}\{\Xi\}.\, \tau \le \sigma)$, or equivalently $outer_V(\Sigma) \vdash (\forall\overline{\beta}\{\rho\Xi\}.\, \rho\tau \le \rho\sigma)$.
By the IH, we have $outer_V(\Sigma) \vdash \rho[\,\overline{\beta \mapsto \pi'}\,]\Xi$, or equivalently $outer_V(\Sigma) \vdash \rho[\,\overline{\beta \mapsto \rho\pi'}\,]\Xi$.
Then by S-Forall-L, we have $outer_V(\Sigma) \vdash \forall\overline{\beta}\{\rho\Xi\}.\, \rho\tau \le [\,\overline{\beta \mapsto \rho\pi'}\,]\rho\tau$ $(**)$.
Further, by the IH we have $outer_V(\Sigma) \vdash \rho([\,\overline{\beta \mapsto \pi'}\,]\tau \le \sigma)$, or equivalently $outer_V(\Sigma) \vdash \rho([\,\overline{\beta \mapsto \rho\pi'}\,]\tau \le \sigma)$. Since $\overline{\beta}$ is locally bound, we have $\rho[\,\overline{\beta \mapsto \rho\pi'}\,]\tau = [\,\overline{\beta \mapsto \rho\pi'}\,]\rho\tau$.
Then we can conclude by S-Trans.

Case P-Forall-R.

$$\frac{\begin{array}{ll} b \ \textit{fresh} & V' \supseteq FV(\tau \le \sigma) \qquad V' \# \overline{\alpha} \cdot b \\ \textit{acyclic}(\Xi) & V', \overline{\alpha} \cdot b \vdash \tau \le [a \mapsto b]\sigma \dashv \Xi \\ V', \overline{\alpha} \cdot b \vdash \Xi \dashv \Xi & V, \overline{\alpha} \vdash \textit{outer}_{V'}(\Xi) \dashv \Sigma \end{array}}{V, \overline{\alpha} \vdash \tau \le \forall a.\, \sigma \dashv \Sigma}$$

The goal is to show $\textit{outer}_V(\Sigma) \vdash \rho\tau \le \forall a.\, \rho\sigma$.
By the IH, we have

$$\textit{outer}_{V'}(\Xi) \vdash \rho'\tau \le \rho'[a \mapsto b]\sigma,$$

where $\rho' = LBU^{\textit{inner}_{V'}(\Xi)}_{V' \cdot \overline{\alpha} \cdot b}$.
Since the domain of ρ' is $FV(\textit{inner}_{V'}(\Xi)) \setminus (V' \cdot \overline{\alpha} \cdot b)$, which is disjoint with $FV(\tau \le [a \mapsto \pi']\sigma)$, we then equivalently have that:

$$\textit{outer}_{V'}(\Xi) \vdash \tau \le [a \mapsto b]\sigma \quad \text{(I)}.$$

By the IH, we have

$$\textit{outer}_V(\Sigma) \vdash \rho(\textit{outer}_{V'}(\Xi)) \quad \text{(II)}.$$

Given (I) and (II), by subtyping entailment under substitutions (D.39) we have:

$$\textit{outer}_V(\Sigma) \vdash \rho\tau \le \rho[a \mapsto b]\sigma \quad \text{(III)}.$$

We now show a chain of subtyping relationships and connect them by S-Trans to conclude:

$$\begin{aligned} \textit{outer}_V(\Sigma) \vdash \rho\tau &\le \forall b.\, \rho\tau && \text{(by S-Forall-R and } b \text{ is fresh)} \\ &\le \forall b.\, \rho[a \mapsto b]\sigma && \text{(by S-Forall-Cov and III)} \\ &= \forall a.\, \rho\sigma && \text{(by } \alpha\text{-equivalency)} \end{aligned}$$ □

Lemma D.36. *If* $V, \overline{\alpha} \vdash \tau \le \sigma \dashv \Sigma$ *and* $\textit{acyclic}(\Sigma)$*, then we have*

$$\textit{outer}_V(\Sigma) \cdot \textit{inner}_V(\Sigma) \vdash \tau \le \sigma.$$

Proof. By induction on the full propagation derivation.

Case P-VarRefl By S-VarRefl.
Case P-Fun By the IH and S-Fun.
Case P-Rigid-L, P-Rigid-R By S-Hyp & S-Trans & D.38.
Case P-Flex By S-Hyp. Note that $(\tau \le \sigma) \notin \textit{outer}_V(\Sigma)$.
Case P-Forall-L By the IH, S-Forall-L & S-Trans.
Case P-Forall-R

$$\frac{\begin{array}{ll} b \ \textit{fresh} & V' \supseteq FV(\tau \le \sigma) \qquad V' \# \overline{\alpha} \cdot b \\ \textit{acyclic}(\Xi) & V', \overline{\alpha} \cdot b \vdash \tau \le [a \mapsto b]\sigma \dashv \Xi \\ V', \overline{\alpha} \cdot b \vdash \Xi \dashv \Xi \ \text{(I)} & V, \overline{\alpha} \vdash \textit{outer}_{V'}(\Xi) \dashv \Sigma \end{array}}{V, \overline{\alpha} \vdash \tau \le \forall a.\, \sigma \dashv \Sigma}$$

The goal is to show $\textit{outer}_V(\Sigma) \cdot \textit{inner}_V(\Sigma) \vdash \tau \le \forall a.\, \sigma$. Note that Ξ may contain new flexible variables "created" while deriving (I); we define those variables as $X \equiv FV(\Xi) \setminus \{ V' \cdot \overline{\alpha} \cdot b \}$.
Note that $FV(\Xi) \setminus \{ V' \cdot \overline{\alpha} \cdot b \} = FV(\textit{inner}_{V'}(\Xi)) \setminus \{ V' \cdot \overline{\alpha} \cdot b \}$.
By the IH, we have:

$$\textit{outer}_{V'}(\Xi) \cdot \textit{inner}_{V'}(\Xi) \vdash \tau \le [a \mapsto b]\sigma \quad \text{(IV)}.$$

By the IH, we have:

$$\textit{outer}_V(\Sigma) \cdot \textit{inner}_V(\Sigma) \vdash \textit{outer}_{V'}(\Xi) \quad \text{(II)}.$$

By D.34 (reusing premise (I)), we have $outer_{V'}(\Xi) \vdash LBU^{inner_{V'}(\Xi)}_{V \cdot \overline{\alpha} \cdot b}(inner_{V'}(\Xi))$, which, if we use $LBU^{inner_{V'}(\Xi)}_{V \cdot \overline{\alpha} \cdot b}$ as the substitution, gives us

$$outer_{V'}(\Xi) \vdash \forall X\{inner_{V'}(\Xi)\} \textbf{ cons.} \quad \text{(III)}.$$

We next show several subtyping relationships. By S-Forall-R and (III), we have $outer_{V'}(\Xi) \vdash \tau \leq \forall X\{inner_{V'}(\Xi)\}.\, \forall b.\, \tau$ where b is a fresh type variable. Note that $FV(\tau) \subseteq V' \# X \cdot b$, as X are fresh flexible variables from (I) and b itself is fresh. Then by S-Forall-Cov, we have

$$outer_{V'}(\Xi) \vdash \forall X\{inner_{V'}(\Xi)\}.\, \forall b.\, \tau \leq \forall X\{inner_{V'}(\Xi)\}.\, \forall b.\, [a \mapsto b]\sigma.$$

By S-Forall-L, we have

$$outer_{V'}(\Xi) \vdash \forall X\{inner_{V'}(\Xi)\}.\, \forall b.\, [a \mapsto b]\sigma \leq \forall b.\, [a \mapsto b]\sigma.$$

Note that $FV(\sigma) \subseteq V' \# X$ and the range of ω avoids X and b.
Finally, we chain the above relationships with S-Trans and rename b to a by α-equivalency to have $outer_{V'}(\Xi) \vdash \tau \leq \forall a.\, \sigma$. We conclude with (II) and subtyping entailment (D.37). □

D.2.9 *Auxiliary subtyping lemmas.*

Lemma D.37 (Subtyping entailment). *If $\Xi \vdash \tau \leq \sigma$ and $\Sigma \vdash \Xi$, then $\Sigma \vdash \tau \leq \sigma$.*

Proof. By $\Sigma \vdash \Xi$, we have $\Sigma \vdash \forall \epsilon\{\Xi\}$ ***cons.*** (I). By weakening, we have $\Sigma \cdot \Xi \vdash \tau \leq \sigma$ (II). We show a chain of subtyping relationships and connect them by S-Trans to conclude:

$$\begin{aligned} \Sigma \vdash \tau &\leq \forall \epsilon\{\Xi\}.\, \tau && \text{(by S-Forall-R and (I))} \\ &\leq \forall \epsilon\{\Xi\}.\, \sigma && \text{(by S-Forall-Cov and (II))} \\ &\leq \sigma && \text{(by S-Forall-L)} \end{aligned}$$ □

Lemma D.38 (Polarized type substitution). *If $\theta = [\overline{\alpha- \mapsto \sigma^-,\ \alpha+ \mapsto \sigma^+}]$ where for each $\beta \in \overline{\beta}$, $\beta-$ is substituted as $\bot$ and $\beta+$ is substituted as $\top$ in θ, and $\Sigma \vdash \rho(\sigma^- \leq \alpha) \cdot \rho(\alpha \leq \sigma^+)$, then:*

*(1) if $\overline{\beta} \vdash \tau^+$ **X-ok** and $\Xi \vdash \rho\tau^+ \leq \rho\tau^+$, then $\Sigma \vdash \rho(\tau^+ \leq \theta^+\tau^+)$.*
*(2) if $\overline{\beta} \vdash \tau^-$ **X-ok** and $\Xi \vdash \rho\tau^- \leq \rho\tau^-$, then $\Sigma \vdash \rho(\theta^-\tau^- \leq \tau^-)$*

Proof sketch. By induction on the polarized type τ^+ for (1) (τ^- for (2)). Polarized substitutions replace a type variable at the negative position by its subtype (i.e. σ^-), and symmetrically for the positive position. $\Xi \vdash \rho\tau^\pm \leq \rho\tau^\pm$ makes sure $\tau^\pm$ itself has consistent bounds under some substitution ρ, and $\tau^\pm$ **X-ok** guarantees that polarized substitutions do not introduce inconsistent polymorphic types. □

Lemma D.39 (Subtyping entailment under substitutions). *If $\Xi \vdash \tau \leq \sigma$ and $\Sigma \vdash \rho\Xi$ and $dom(\rho) \subseteq FV(\tau \leq \sigma)$ and ρ avoids capturing and ρ is idempotent, then $\Sigma \vdash \rho(\tau \leq \sigma)$.*

Proof. By induction on the subtyping judgment. For cases S-Forall-R, S-Forall-Cov, and S-Forall-Distr, we expand the bound consistency premise $\Xi \vdash \forall V\{\Sigma'\}$ ***cons.*** to subtyping judgments $\Xi \vdash \rho'\Sigma'$ where $dom(\rho') = V$ according to the definition of consistency. We invoke the IH on those subtyping judgments and have $\Sigma \vdash \rho\rho'\Sigma'$. To prove the goal, we need $\Sigma \vdash \forall V\{\rho\Sigma'\}$ ***cons.***. We use $\rho\rho'$ as the substitutions of V, and $\Sigma \vdash \rho\rho'\rho\Sigma'$ is implied by $\Sigma \vdash \rho\rho'\Sigma'$ because ρ is capture-avoiding and idempotent. □

D.2.10 Soundness of constraining.

Theorem D.40 (Soundness of Constraining). *Let* $V, \overline{\alpha} \vdash \epsilon \gg \Delta \gg \Sigma'$ *and* $\Sigma = uproot(\Sigma')$. *Then for all* $\overline{\tau}$ *where* $FV(\overline{\tau}) \# \overline{\beta}$ *and* $\overline{\beta} = FV(\Sigma) \setminus \{ V \cdot \overline{\alpha} \}$ *we have*

- $outer_V(\Sigma) \cdot inner_V([\overline{\alpha \mapsto \tau}]\Sigma) \vdash [\overline{\alpha \mapsto \tau}]\Delta$ *and*
- $outer_V(\Sigma) \vdash \forall \overline{\beta}\{inner_V([\overline{\alpha \mapsto \tau}]\Sigma)\}$ ***cons.***.

Proof. By Lemmas D.32 and D.36, we have $outer_V(\Sigma) \cdot inner_V(\Sigma) \vdash \Delta$. By Lemma D.39, we have $outer_V(\Sigma) \cdot inner_V([\overline{\alpha \mapsto \tau}]\Sigma) \vdash [\overline{\alpha \mapsto \tau}]\Delta$. Notice that $[\overline{\alpha \mapsto \tau}]outer_V(\Sigma) = outer_V(\Sigma)$. By Lemmas D.32 and D.34, we have $outer_V(\Sigma) \vdash LBU_{V \cdot \overline{\alpha}}^{inner_V(\Sigma)} inner_V(\Sigma)$. By Lemma D.39, we have $outer_V(\Sigma) \vdash [\overline{\alpha \mapsto \tau}]LBU_{V \cdot \overline{\alpha}}^{inner_V(\Sigma)} inner_V(\Sigma)$. Given the fact that Σ contains no bounds on skolems and skolems in the bounds of rigid type variables are extruded , we equivalently have:

$$outer_V(\Sigma) \vdash [\overline{\alpha \mapsto \tau}]LBU_{V \cdot \overline{\alpha}}^{inner_V(\Sigma)} inner_V([\overline{\alpha \mapsto \tau}]\Sigma).$$

This allows us to conclude that $outer_V(\Sigma) \vdash \forall \overline{\beta}\{inner_V([\overline{\alpha \mapsto \tau}]\Sigma)\}$ ***cons.***. □

Received 2023-07-11; accepted 2023-11-07