CHALMERS
UNIVERSITY OF TECHNOLOGY

A theory of higher-order subtyping with type intervals

Downloaded from: https://research.chalmers.se, 2024-07-18 14:32 UTC

Citation for the original published paper (version of record):
Stucki, S., Giarrusso, P. (2021). A theory of higher-order subtyping with type intervals. Proceedings of the ACM on Programming Languages, 5(ICFP). http://dx.doi.org/10.1145/3473574

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology.
It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004.
research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

A Theory of Higher-Order Subtyping with Type Intervals

SANDRO STUCKI, Chalmers University of Technology, Sweden
PAOLO G. GIARRUSSO, Bedrock Systems Inc., Germany

The calculus of Dependent Object Types (DOT) has enabled a more principled and robust implementation of Scala, but its support for type-level computation has proven insufficient. As a remedy, we propose $F^{\omega}_{\cdot\cdot}$, a rigorous theoretical foundation for Scala's higher-kinded types. $F^{\omega}_{\cdot\cdot}$ extends $F^{\omega}_{<:}$ with *interval kinds*, which afford a unified treatment of important type- and kind-level abstraction mechanisms found in Scala, such as bounded quantification, bounded operator abstractions, translucent type definitions and first-class subtyping constraints. The result is a flexible and general theory of higher-order subtyping. We prove type and kind safety of $F^{\omega}_{\cdot\cdot}$, as well as weak normalization of types and undecidability of subtyping. All our proofs are mechanized in Agda using a fully syntactic approach based on hereditary substitution.

CCS Concepts: • **Theory of computation** → **Type theory**; **Type structures**; *Operational semantics*; *Program verification.*

Additional Key Words and Phrases: Scala, higher-kinded types, subtyping, type intervals, bounded polymorphism, bounded type operators, singleton kinds, dependent kinds, hereditary substitution

ACM Reference Format:
Sandro Stucki and Paolo G. Giarrusso. 2021. A Theory of Higher-Order Subtyping with Type Intervals. *Proc. ACM Program. Lang.* 5, ICFP, Article 69 (August 2021), 30 pages. https://doi.org/10.1145/3473574

1 INTRODUCTION

Modern statically typed programming languages provide powerful type-level abstraction facilities such as parametric polymorphism, subtyping, generic datatypes, and varying degrees of type-level computation. When several of these features are present in the same language, new and more expressive combinations arise, such as (F1) bounded quantification, (F2) bounded type operators, and (F3) translucent type definitions. Such mechanisms further increase the expressivity of the language, but also the complexity of its type system and, ultimately, its implementation.

A case in point is the Scala programming language. Scala is a multi-paradigm language that integrates functional and object-oriented concepts. It features all of the aforementioned type-level constructs and many more. While programmers enjoy the expressivity of Scala's type system, the language developers have struggled for years to manage its complexity – tracing down elusive compiler bugs and soundness issues in a seemingly ad-hoc fashion. The development of the calculus of Dependent Object Types (DOT) [Amin et al. 2016] marked a turning point. By providing a solid theoretical foundation for a large part of the Scala language, DOT inspired a complete re-design of the Scala compiler as well as a substantial redesign of the language itself – giving rise to Scala 3 [Dotty Team 2020]. But despite initial hopes to the contrary, DOT has proven insufficient to express Scala's *higher-kinded* (HK) types, which are used pervasively throughout the Scala standard library [Moors et al. 2008b]. The lack of a proper theory of HK types severely complicates

Authors' addresses: Sandro Stucki, Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, sandros@chalmers.se; Paolo G. Giarrusso, Bedrock Systems Inc., Berlin, Germany, p.giarrusso@gmail.com.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/8-ART69
https://doi.org/10.1145/3473574

their implementation in Scala 3 [Odersky et al. 2016]. To address this problem, we introduce $F^{\omega}_{\cdot\cdot}$, a rigorous theoretical foundation for Scala's HK types with a machine-checked safety proof.

We start by illustrating the use of HK types through an example.

```
type Ordering[A] = (A, A) => Boolean
class SortedView[A, B >: A](xs: List[A], ord: Ordering[B]) {
  def foldLeft[C](z: C, op: (C, A) => C): C = //...
  def concat[C >: A <: B](ys: List[C]): SortedView[C, B] = //...
  // definitions of further operations such as 'map', 'flatMap', etc.
}
```

The example is inspired by the definition of the class `SeqView.Sorted` from the standard library.[1] It is heavily simplified for conciseness and to avoid the use of Scala idioms irrelevant to this paper.

The `SortedView` class allows one to iterate over the elements of an underlying list `xs` in increasing order (via `foldLeft`) without modifying the original list. The second constructor argument `ord` provides the comparison operation used for sorting. Note that `ord` compares data of type `B`, which is declared a supertype of the element type `A` via the annotation `B >: A`. Thus we may use a comparison function for a less precise type to sort the elements in the view.[2]

In Scala, parametrized classes like `SortedView` are first-class type operators. Hence, `SortedView` is an instance of a lower-bounded type operator (F2). The operator `Ordering[A]` is just a convenient type alias for the function type `(A, A) => Boolean` of binary operators. It is a *transparent* type definition (F3), in that instances of `Ordering[A]` can be replaced by its definition anywhere in the program. The method `concat` illustrates the use of lower- and upper-bounded polymorphism (F1) to combine views on lists of different but related element types. Views are covariant: if `A` is a subtype of `C` then a view `v` on an `A`-list is also a view on a `C`-list and can be extended as such using `v.concat(ys)`. The upper bound on `C` ensures that the ordering remains applicable. This pattern of using lower-bounds to implement operations on covariant data structures is common in the Scala standard library.

Although bounded quantification (F1) and bounded operators (F2) may seem conceptually different from translucent type definitions (F3), all three are closely related. A type alias declaration of the form `X = A` (such as `Ordering` above) declares that the type `X` has type `A` as upper and lower bound. Because subtyping in Scala is antisymmetric, this effectively identifies `X` with `A`. In general, a type declaration of the form `X >: A <: B`, introduces an abstract type `X` that is bounded by `A` from below and by `B` from above. In other words, the declaration `X >: A <: B` specifies a *type interval* in which `X` must be contained. Type aliases take the form of *singleton intervals* `X >: A <: A`, where the lower and upper bounds coincide.

In our example, the types `Ordering` and `SortedView` can be seen as abstract types. We can write their declarations as[3]

```
type Ordering[A] >: (A, A) => Boolean <: (A, A) => Boolean
type SortedView[A, B >: A] <: {
  def foldLeft[C](z: C, op: (C, A) => C): C
  def concat[C >: A <: B](ys: List[C]): SortedView[C, B]   /* ... */   }
```

[1] See https://www.scala-lang.org/api/2.13.6/scala/collection/SeqView$$Sorted.html

[2] Expert readers may notice that `Ordering` is contravariant in its argument, hence `SortedView` could simply take a parameter of type `Ordering[A]` and still accept instances of `Ordering[B]`. But this simplification does not extend to the Scala standard library, where `Ordering` is invariant due to additional methods; hence the lower bound is necessary. For readability, we use the simplified version of `Ordering` but keep the bound in the definition of `SeqView`.

[3] This code is accepted by the Scala 2.13.6 compiler. Ironically, it is rejected by Scala 3, which is built on DOT.

This version is closer to how type definitions are represented in DOT. The class `SortedView` is now represented as an abstract type declaration with only an upper bound. This means that its interface remains exposed – any instance of `SortedView[A, B]` can be up-cast to a record with fields `foldLeft`, `concat`, etc. But not every record (or class) containing those fields is automatically an instance of the type `SortedView[A, B]`. This ensures that `SortedView` continues to behave like a nominal type, as Scala classes are supposed to.[4] Unlike `SortedView`, the abstract type `Ordering` is both upper- and lower-bounded.

Intervals can also encode abstract types missing a bound. Scala features a pair of extremal types `Any` and `Nothing`: the maximal type `Any` is a supertype of every other type; the minimal type `Nothing` is a subtype of every other type. Abstract types `X` with only an upper or lower bound `A` thus inhabit the degenerate intervals `X >: Nothing <: A` and `X >: A <: Any`, respectively.

This suggests a uniform treatment of F1–F3 through type intervals, and indeed, this is essentially how bounded quantification (F1) and type definitions (F3) are modeled in DOT. Unfortunately, DOT lacks intrinsics for higher-order type computation, such as type operator abstractions and applications, preventing it from encoding simple HK type definitions such as the identity operator **`type`** `Id[X] = X` [Odersky et al. 2016]. Traditionally, F1–F3 have been studied through orthogonal extensions of Girard's higher-order polymorphic λ-calculus F_ω [1972]. Bounded higher-order subtyping (F1 and F2) has been formalized in variants of $F^{\omega}_{<:}$ [Compagnoni and Goguen 2003; Pierce and Steffen 1997], translucent type definitions (F3) through singleton kinds [Stone and Harper 2000]. The treatment of F1 and F3 in DOT suggest a different, unified approach to studying F1–F3: via a formal theory of higher-order subtyping with type intervals, which we pursue with $F^{\omega}_{..}$.

Our goal in doing so is twofold. First, we want to establish a theoretical foundation for Scala's HK types, with full support for type-level computations (including F1–F3). A principled theoretical understanding of HK types is crucial because potential safety issues are hard to identify and fix by "trial and error" alone, especially when they arise from feature interactions. Second, we want to study the concept of type intervals in its own right. Despite their apparent simplicity, adding type intervals to $F^{\omega}_{<:}$ leads to a surprisingly rich theory of higher-order subtyping that goes beyond previous treatments of F1–F3. That is because type intervals encode first-class subtyping constraints, or *type inequations*, similar to extensional identity types in Martin-Löf type theory (MLTT).

In DOT, type intervals are baked into abstract type members and therefore tied to the use of *path-dependent types*; in $F^{\omega}_{..}$, we break this bond. A DOT type member declaration $\{\, X \colon A \mathinner{..} B \,\}$ roughly corresponds to a Scala-style type member declaration **`class`** `C {` **`type`** `X >: A <: B }`. It combines two separate type-system features: the declaration of an abstract type member X in a record or class, and the declaration of subtyping constraints on X via the bounds A and B. These two features are independent. For example, the Agda programming language features unbounded abstract type members via record types while the $F_{<:}$ calculus features subtyping constraints via bounded quantification but no type members. The notion of path-dependent types in DOT and Scala is intimately linked to abstract type members. Given an instance $z \colon \{\, X \colon A \mathinner{..} B \,\}$, the type expression $z.X$ denotes the type value assigned to X in z. The type $z.X$ is *path-dependent* because it depends on the term-level expression z (the "path" to X). In DOT (but not Scala), type members are also used to model bounded quantification.

In $F^{\omega}_{..}$, we deliberately separate the notion of type intervals from that of abstract type members (and path-dependent types) and drop the latter. This simplifies the theory and allows us to study the power of type intervals in the context of higher-order subtyping: for bounded quantification, bounded operator abstraction and translucent type definitions – all of which are independent of path-dependent types, yet commonly used when working with Scala's HK types.

[4] For details on how Scala classes may be encoded in DOT, we refer the reader to Amin [2016, Chap. 2].

We leave the development of a combined theory of HK and path-dependent types for future work, and focus here on the theoretical and practical insights afforded by the novel combination of *higher-order subtyping with type intervals*. Concretely, we make the following contributions.

(1) We propose *type intervals* as a unifying concept for expressing bounded quantification, bounded operator abstractions, and translucent type definitions. Going beyond the status quo, we show that type intervals are expressive enough to also cover less familiar constructs, such as lower-bounded operator abstractions and first-class inequations (§2).

(2) We introduce $F^{\omega}_{\cdot\cdot}$ – an extension of F_{ω} with *interval kinds* – as a formal calculus of higher-order subtyping with type intervals (§3). $F^{\omega}_{\cdot\cdot}$ is the first formalization of Scala's higher-kinded types with a rigorous, machine-checked type safety proof. As such, it provides a theoretical foundation for several important features of Scala-like type systems.

(3) We establish important metatheoretic properties of our theory: kind safety (§3), weak normalization of types (§4), type safety (§5), and undecidability of subtyping (§6). The metatheoretic proofs are complicated substantially by the interaction of advanced type system features such as dependent kinds, subtyping and subkinding, and (in)equality reflection. As others have recognized [Aspinall and Compagnoni 2001; Yang and Oliveira 2017; Zwanenburg 1999], the combination of dependent types (or kinds) and subtyping poses a particular challenge in metatheoretic developments. The usefulness of our proof techniques thus extends beyond the scope of $F^{\omega}_{\cdot\cdot}$ to other systems combining dependent types and subtyping.

(8) The metatheoretic development is entirely syntactic (it involves no model constructions) and has been fully mechanized using the Agda proof assistant [Norell 2007]. The main technical device is a purely syntactic, bottom-up normalization procedure based on a novel variant of *hereditary substitution* that computes the $\beta\eta$-normal forms of types and kinds (§4).

We outline our proof strategy in §3.6, review related work in §7 and give concluding remarks in §8.

Because of space constraints, we omit most proofs and many details of the metatheory from the paper and focus instead on the big picture: the design and expressiveness of $F^{\omega}_{\cdot\cdot}$ as well as the many challenges involved in proving its type safety and our strategies for addressing them. However, the complete metatheory, including full proofs of all lemmas and theorems stated in the paper, has been mechanized in Agda, and the source code is freely available as an artifact [Stucki and Giarrusso 2021]. An overview of the Agda formalization, establishing the connection to the theory presented in the paper, is available as supplementary material in the ACM digital library, along with detailed human-readable descriptions of metatheoretic results that have been omitted from the paper. Yet more details can be found in the first author's PhD dissertation [Stucki 2017].

2 SUBTYPING WITH TYPE INTERVALS

Before we define our formal theory of type intervals, let us illustrate the core ideas in a bit more detail. Consider the following Scala type definitions.

```
abstract class Bounded[B, F[_ <: B]] { def apply[X <: B]: F[X] }
type All[F[_]] = Bounded[Any, F]
```

The class `Bounded` and the type alias `All` are Scala encodings of the bounded and unbounded universal quantifiers found in $F_{\leq}$ [Curien and Ghelli 1992]. We chose this example for its brevity and because it exemplifies the type-level mechanisms found in more realistic definitions, such as those given in §1. In particular, it features bounded quantification (of `X <: B` in `apply`), a bounded operator (the parameter `F[_ <: B]` of `Bounded`) and a transparent type alias (`All`).

We want to translate these two definitions into a typed λ-calculus. The challenge is to find a type system that is expressive enough to do so. The example involves type-level computations, so our first candidate is Girard's higher-order polymorphic λ-calculus F_ω [1972], but it lacks even basic support for subtyping. Our next candidate is $F^\omega_{<:}$, which extends F_ω with higher-order subtyping and bounded quantification. But most variants of $F^\omega_{<:}$ lack support for bounded operators [cf. Pierce and Steffen 1997; Pierce 2002]. Thankfully, Compagnoni and Goguen have developed $\mathcal{F}^\omega_{\leq}$, a variant of $F^\omega_{<:}$ with bounded operators [2003]. $\mathcal{F}^\omega_{\leq}$ has four type variable binders:

$\lambda X \leq A{:}K.\, t$	term-level type abstraction	$\forall X \leq A{:}K.\, B$	type-level bounded quantifier
$\lambda X \leq A{:}K.\, B$	type-level type abstraction	$(X \leq A{:}K) \to J$	kind-level dependent arrow

The type A in a binding $X \leq A : K$ is called the *upper bound* of X, and must be of kind K. To represent unconstrained bindings, $\mathcal{F}^\omega_{\leq}$ features a *top* type $\top$, which is a supertype of every other type (like `Any` in Scala). The bound $\top$ in $X \leq \top : *$ is thus trivially satisfied and can be omitted.

Since operator abstractions carry bounds in $\mathcal{F}^\omega_{\leq}$, so must arrow kinds. This makes arrow kinds type-dependent, which substantially complicates the meta theory of $\mathcal{F}^\omega_{\leq}$ when compared to other variants of $F^\omega_{<:}$. As usual, we abbreviate $(X{:}J) \to K$ to $J \to K$ when X does not occur freely in K.

A possible translation of `Bounded` and `All` to $\mathcal{F}^\omega_{\leq}$ is

$$\mathit{Bounded} \;\coloneqq\; \lambda B{:}*.\, \lambda F{:}(X \leq B{:}*) \to *.\, \forall X \leq B{:}*.\, F\, X \qquad\qquad \mathit{All} \;\coloneqq\; \mathit{Bounded}\, \top$$

The named, parametrized class `Bounded[B, F[_ <: B]]` has been replaced by a pair of nested anonymous operator abstractions taking arguments $B : *$ and $F : (X \leq B{:}*) \to *$; the signature `apply[X <: B]: F[X]` of the method `apply` by the bounded universal $\forall X \leq B{:}*.\, F\, X$.

The declared kinds of the variables B, F and X in the definition of *Bounded* indicate what sort of type they represent: B and X are proper types, while F is a unary bounded operator. This makes *Bounded* itself a higher-order type operator of kind $* \to ((X \leq B{:}*) \to *) \to *$. For example, we obtain the type of the polymorphic identity function by applying *Bounded* as follows:

$$\begin{aligned}\mathit{Bounded}\, \top\, (\lambda X{:}*.\, X \to X) \;&=\; (\lambda B{:}*.\, \lambda F{:}(X \leq B{:}*) \to *.\, \forall X \leq B{:}*.\, F\, X)\, \top\, (\lambda X{:}*.\, X \to X)\\ &=\; \forall X{:}*.\, X \to X.\end{aligned}$$

The translation of the type alias `All[F[_]]` is then just the partial application $\mathit{All} = \mathit{Bounded}\, \top$.

The above definitions of *Bounded* and *All* are meta-definitions, i.e. they are just convenient shorthands for the type expressions $\lambda B{:}*.\, \lambda F{:}(X \leq B{:}*) \to *.\, \forall X \leq B{:}*.\, F\, X$ and $\mathit{Bounded}\, \top$. But we can also give object-level definitions of *Bounded* and *All* in $\mathcal{F}^\omega_{\leq}$, using standard syntactic sugar for let-binding type and term variables:

$$\textbf{let}\ X : K = A\ \textbf{in}\ t \;\coloneqq\; (\lambda X{:}K.\, t)\, A, \qquad\qquad \textbf{let}\ x : B = s\ \textbf{in}\ t \;\coloneqq\; (\lambda x{:}B.\, t)\, s.$$

We can use *Bounded* as an abstract type operator in a term t by let-binding it to a type variable:

$$\textbf{let}\ \texttt{Bounded} : (B{:}*) \to ((X \leq B{:}*) \to *) \to * \;=\; \lambda B{:}*.\, \lambda F{:}(X \leq B{:}*) \to *.\, \forall X \leq A{:}*.\, F\, X\ \textbf{in}\ t.$$

This definition is *opaque*, i.e. the term t sees the signature of `Bounded`, but not its definition. Consider

$$\begin{array}{llll}\textbf{let}\ \texttt{Bounded} : (B{:}*) \to ((X \leq B{:}*) \to *) \to * & = \;\ldots & \textbf{in} & \\ \textbf{let}\ x : \forall X{:}*.\, X \to X & = \;\lambda X{:}*.\, \lambda z{:}X.\, z & \textbf{in} & \text{— } \mathit{OK}\\ \textbf{let}\ y : \texttt{Bounded}\, \top\, (\lambda X{:}*.\, X \to X) & = \;\lambda X{:}*.\, \lambda z{:}X.\, z & \textbf{in}\ \ldots & \text{— } \mathit{type\ error}\end{array}$$

The third definition does not type check because $\texttt{Bounded} \neq \lambda B{:}*.\, \lambda F{:}(X \leq B{:}*) \to *.\, \forall X{:}*.\, F\, X$ as types, despite the binding. Indeed, $\mathcal{F}^\omega_{\leq}$ cannot express *transparent* type definitions.

Furthermore, $\mathcal{F}^\omega_{\leq}$ also lacks support for *lower-bounded* definitions. As discussed in §1, these have important applications e.g. in the Scala standard library. Both transparent and lower-bounded definitions, and all the features of $\mathcal{F}^\omega_{\leq}$, can be uniformly expressed using *interval kinds*.

2.1 Intervals and Singletons

As the name implies, an interval kind $A\,..\,B$ is inhabited by a range of proper types C, namely those that are supertypes $C \geq A$ of its lower bound A and subtypes $C \leq B$ of its upper bound B. Hence, kinding statements of the form $C : A\,..\,B$ are equivalent to pairs of subtyping statements $A \leq C$ and $C \leq B$. We make this equivalence formal in the next section.

Since every proper type is a subtype of $\top$, intervals of the form $A\,..\,\top$ are effectively unconstrained from above, and can thus be used to encode *lower-bounded* definitions. Similarly, upper-bounded definitions can be expressed using intervals of the form $\bot\,..\,A$ where the lower bound is the minimum or *bottom* type $\bot$, our equivalent of Scala's `Nothing` type. For example, we recover $F_{\leq}$-style bounded quantifiers $\forall X \leq B.\, A$ as $\forall X{:}\bot\,..\,B.\, A$. Interval kinds of the form $A\,..\,A$, where the lower and upper bounds coincide, are called *singleton kinds* or simply *singletons*. Given $B : A\,..\,A$, we have both $A \leq B$ and $B \leq A$, which, assuming an antisymmetric subtyping relation, implies $A = B$. Singleton kinds can thus encode transparent definitions and have been studied for that purpose by Stone and Harper [2000]. We adopt their notation $S(A) = A\,..\,A$ for the singleton containing just A.

Using interval kinds, we refine our definitions of `Bounded` and `All` to make them transparent.

$$\begin{aligned}
&\mathbf{let}\ \texttt{Bounded} : (B{:}*) \to (F{:}\,\bot\,..\,B \to *) \to S(\forall X{:}\bot\,..\,B.\, F\,X) \\
&\qquad\qquad = \lambda B{:}*.\, \lambda F{:}\,\bot\,..\,B \to *.\, \forall X{:}\bot\,..\,B.\, F\,X \qquad\qquad \mathbf{in} \\
&\mathbf{let}\quad \texttt{All} : (F{:}* \to *) \to S(\texttt{Bounded}\,\top\,F) \;=\; \texttt{Bounded}\,\top \quad \mathbf{in}\ \ldots
\end{aligned}$$

The signature of `Bounded` tells us that, when we apply it to suitable type arguments B and F, the result is both a subtype and a supertype of $\forall X{:}\bot\,..\,B.\, F\,X$. In other words, we have $\texttt{Bounded}\,B\,F = \forall X{:}\bot\,..\,B.\, F\,X$ in the body of the let-binding. Similarly, we have $\texttt{All}\,F = \texttt{Bounded}\,\top\,F$, as desired.

Interval kinds $A\,..\,B$ are only well-formed if A and B are proper types, i.e. of kind $*$. To express all of the binders found in $\mathcal{F}^{\omega}_{\leq}$, we need a way to encode bindings of the form $X \leq A : K$, for arbitrary kinds K. As we will see in §3, this is indeed possible because $F^{\omega}_{\cdot\cdot}$ can encode *higher-order interval kinds* $A\,{..}_K\,B$ for arbitrary K using its other kind- and type-level constructs.

2.2 First-Class Inequations

Instances $C : A\,..\,B$ of an interval kind $A\,..\,B$ represent types bounded by A and B respectively. But they also represent proofs that $A \leq C$ and $C \leq B$, and – by transitivity of subtyping – that $A \leq B$. In other words, the inhabitants of interval kinds $A\,..\,B$ represent first-class *type inequations* $A \leq B$. Similarly, higher-order intervals represent type operator inequations. Interval kinds thus provide us with a mechanism for *(in)equality reflection*, i.e. a way to extend the subtyping relation via assumptions made at the term- or type-level (via type abstractions).

Among other things, this allows us to postulate type operators with associated subtyping rules through type variable bindings. We will see an example of this in §6; other examples are intersection types or equi-recursive types and their associated subtyping theories (see Appendix E for a detailed example). This is possible because we do not impose any *consistency constraints* on the bounds of intervals. That is, an interval kind $A\,..\,B$ is well-formed, irrespective of whether $A \leq B$ is actually provable or not. If we can prove that $A \leq B$, we say that the bounds of $A\,..\,B$ are *consistent*.

Having both (in)equality reflection and inconsistent bounds makes $F^{\omega}_{\cdot\cdot}$ very expressive, but breaks subject reduction of open terms and decidability of (sub)typing. This is common in type theories with equality reflection (e.g. extensional MLTT [Nordström et al. 1990]) because they allow the reflection of absurd assumptions. For example, in a context where $Z : \top\,..\,\bot$, we have $\top \leq Z \leq \bot$,

$$\begin{array}{rcll}
x, y, z, \dots & & & \textbf{Term variable} \qquad X, Y, Z, \dots \qquad \textbf{Type variable} \\
s, t & ::= & x \mid \lambda x{:}A.\, t \mid s\, t \mid \lambda X{:}K.\, t \mid t\, A & \textbf{Term} \\
u, v & ::= & \lambda x{:}A.\, t \mid \lambda X{:}K.\, t & \textbf{Value} \\
A, B, C & ::= & X \mid \top \mid \bot \mid A \to B \mid \forall X{:}K.\, A \mid \lambda X{:}K.\, A \mid A\, B & \textbf{Type} \\
\Gamma, \Delta & ::= & \varnothing \mid \Gamma, x{:}A \mid \Gamma, X{:}K & \textbf{Typing context} \\
J, K, L & ::= & A \mathrel{..} B \mid (X{:}J) \to K \quad \textbf{Kind} \qquad j, k, l \;::=\; * \mid j \to k & \textbf{Shape (simple kind)}
\end{array}$$

Fig. 1. Syntax of $F^{\omega}_{::}$.

i.e. the subtyping relation becomes trivial, and we can type non-terminating and stuck terms.

$$(\lambda x{:}\top.\, x\, x)\; (\lambda x{:}\top.\, x\, x) : \top \qquad \textit{— non-terminating, but } \top \le Z \le \bot \le \top \to \top$$

$$(\lambda X{:}*.\, \lambda x{:}X.\, x)\; (\lambda x{:}\top.\, x) : \top \qquad \textit{— stuck, but } \forall X{:}*.\, X \to X \le \top \le Z \le \bot \le \top \to \top$$

It is therefore unsafe to reduce terms under absurd assumptions in general. Note that these examples do not break type safety of $F^{\omega}_{::}$ overall though. The absurd assumption $Z : \top \mathrel{..} \bot$ can never be instantiated, and hence reduction of *closed* terms remains perfectly safe. We discuss this point in more detail at the end of §3. The use of inconsistent bounds to prove undecidability of subtyping is more subtle; we return to it in §6.

Seeing the trouble inconsistent bounds can cause, one may wonder why we do not just enforce consistency of interval bounds statically. There are several reasons.

- *Statically enforcing consistent bounds in Scala is hard.* While we could statically enforce consistent bounds in $F^{\omega}_{::}$, Amin et al. [2014] have shown that this would not extend to systems closer to Scala; a detailed explanation is given by Amin [2016, Sec. 3.4.4, 4.2.3].
- *Inconsistent bounds are useful.* Intervals with unconstrained bounds are useful, e.g. to encode generalized algebraic datatypes (GADTs) via first-class inequality constraints [Cretin and Rémy 2014; Parreaux et al. 2019]. Consistency of such constraints cannot be established when a GADT is defined, only when it is instantiated.
- *Decidability of subtyping could be recovered.* Even if we enforced consistent bounds in $F^{\omega}_{::}$, subtyping would likely remain undecidable because $F^{\omega}_{::}$, like full $F_{\le}$ and all variants of the DOT calculus, use a strong subtyping rule for universals that is a known sources of undecidability [Pierce 1992]. Recent work by Hu and Lhoták [2019] suggests a novel approach for algorithmic subtyping that handles both inconsistent bounds and strong subtyping for universals. Whether or not their approach can be generalized to higher-order subtyping is a question we leave for future work.

3 THE DECLARATIVE SYSTEM

In this section, we introduce $F^{\omega}_{::}$ – our formal theory of higher-order subtyping with type intervals. We present its syntax and its type system, and establish some basic metatheoretic properties – just enough to show that subject reduction holds for well-kinded open types. Finally, we discuss the challenges involved in proving type safety, and outline our strategy for doing so.

3.1 Syntax

The syntax of $F^{\omega}_{::}$ is given in Fig. 1. The syntax of terms and types is identical to that of F_{ω} except for the extremal type constants $\top$ and $\bot$. The *top type* $\top$ is the maximal proper type: any other proper type is a subtype of $\top$. Dually, the *bottom type* $\bot$ is the minimal proper type. Following Pierce [2002], λs carry domain annotations. This will become important in §3.2, §4.3 and §4.4.

Kind constants

$$\begin{aligned} * &:= \bot \mathrel{..} \top \\ \emptyset &:= \top \mathrel{..} \bot \end{aligned}$$

Higher-order type intervals $\boxed{A \mathrel{..}_K B}$ $\boxed{*_K}$

$$\begin{aligned} A \mathrel{..}_{A' \mathrel{..} B'} B &:= A \mathrel{..} B \\ A \mathrel{..}_{(X:J)\to K} B &:= (X:J) \to A\,X \mathrel{..}_K B\,X \\ &\quad\ \text{for } X \notin \mathrm{fv}(A) \cup \mathrm{fv}(B) \end{aligned}$$

$$\begin{aligned} *_{A \mathrel{..} B} &:= \bot \mathrel{..} \top \\ *_{(X:J)\to K} &:= (X:J) \to *_K \end{aligned}$$

Higher-order extrema $\boxed{\top_K}$ $\boxed{\bot_K}$

$$\begin{aligned} \top_{A \mathrel{..} B} &:= \top & \qquad \bot_{A \mathrel{..} B} &:= \bot \\ \top_{(X:J)\to K} &:= \lambda X{:}J.\, \top_K & \qquad \bot_{(X:J)\to K} &:= \lambda X{:}J.\, \bot_K \end{aligned}$$

Type erasure $\boxed{|K|}$

$$\begin{aligned} |A \mathrel{..} B| &:= * \\ |(X:J) \to K| &:= |J| \to |K| \end{aligned}$$

Bounded quantification and type operators

$$\begin{aligned} \forall X \le A{:}K.\, B &:= \forall X{:}(\bot_K) \mathrel{..}_K A.\, B & \qquad (X \le A{:}J) \to K &:= (X{:}(\bot_J) \mathrel{..}_J A) \to K \\ \lambda X \le A{:}K.\, t &:= \lambda X{:}(\bot_K) \mathrel{..}_K A.\, t & \qquad \lambda X \le A{:}K.\, B &:= \lambda X{:}(\bot_K) \mathrel{..}_K A.\, B \end{aligned}$$

Fig. 2. Syntactic shorthands and encodings

The main differences between $F^{\omega}_{..}$ and other variants of F_ω are reflected in its kind language. First, the usual kind of proper types $*$ is replaced by the *interval kind* former $A \mathrel{..} B$. The interval $A \mathrel{..} B$ is inhabited by exactly those proper types that are supertypes of A and subtypes of B. The degenerate interval $\bot \mathrel{..} \top$ spans *all* proper types. Hence we use $*$ as a shorthand for $\bot \mathrel{..} \top$. Second, most variants of F_ω have a *simple* kind language (as described by the non-terminal k in Fig. 1). In contrast, $F^{\omega}_{..}$ has a *dependent* kind language. The arrow kind $(X:J) \to K$ acts as a binder for the type variable X which may appear freely in the codomain K. Dependent kinds play an important role when modeling *bounded type operators*. For example, consider a binary type operator of kind $(X{:}*) \to (Y{:}\bot \mathrel{..} X) \to *$. The upper-bounded kind $\bot \mathrel{..} X$ of Y ensures that the operator can only be applied to types A, B if B is a subtype of A. This idea goes back to Compagnoni and Goguen's $\mathcal{F}^{\omega}_{\le}$, which features both upper-bounded type operators and dependent arrow kinds [2003].

We abbreviate $\bot \mathrel{..} \top$ by $*$ and $(X:J) \to K$ by $J \to K$ when X is not free in K. This allows us to treat *simple kinds* or *shapes* k as a subset of (dependent) kinds K. In the opposite direction, we define an *erasure* map $|K|$ which forgets any dependencies in K (see Fig. 2). Given a kind K, we say *K has shape* $|K|$. Unlike kinds, shapes are stable under substitution, i.e. $|K[A/X]| \equiv |K|$.

Following Barendregt [1992], we identify expressions e (terms, types and kinds), up to *α-equivalence* and assume that the names of bound and free variables are distinct. We write $e \equiv e'$ to stress that e and e' are α-equivalent. The set of free variables of e is denoted by $\mathrm{fv}(e)$, and we write $e[t/x]$ and $e[A/X]$ for capture-avoiding term and type substitutions in e, respectively. We require that the variables bound in a typing context Γ be distinct so that we may think of Γ as a finite map and use function notation, such as $\mathrm{dom}(\Gamma)$, $\Gamma(x)$, $\Gamma(X)$. We write (Γ, Δ) for the concatenation of two contexts Γ and Δ with disjoint domains, and we often omit the empty context $\emptyset$, writing e.g. $\Gamma = x{:}A, Y{:}K$ instead of $\Gamma = \emptyset, x{:}A, Y{:}K$.

3.1.1 Encodings. Together with the extremal types $\top$ and $\bot$, interval kinds allow us to express *bounded quantification* and *bounded operators* over proper types. For example, the $F_{\le}$-style universal type $\forall X \le A.\, B$ can be expressed as $\forall X{:}\bot \mathrel{..} A.\, B$ in $F^{\omega}_{..}$. To extend this principle to *higher-order* bounded quantification and type operators, we define encodings for higher-order interval kinds and extremal types via type abstraction and dependent kinds in Fig. 2. The encoding of *higher-order maxima* $\top_K$ is standard [cf. Compagnoni and Goguen 2003; Pierce 2002]; that of *higher-order* *minima* $\bot_K$ follows the same principle. The encoding of *higher-order interval kinds* $A\,.._K\,B$ resembles that of higher-order *singleton kinds* given by Stone and Harper [2000]. Indeed, singleton kinds are just interval kinds where the upper and lower bounds coincide. Encodings of higher-order $\mathcal{F}^{\omega}_{\leq}$-style bounded operators and universal quantifiers are also given in Fig. 2.

3.1.2 Structural Operational Semantics. For computations in terms, we adopt the standard call-by-value (CBV) semantics given by Pierce [2002, Fig. 30-1], writing $t \longrightarrow^{*}_{\mathsf{v}} t'$ when the term t CBV-reduces in one or more steps to t'. For types and kinds, we define the *one-step β-reduction* relation $\longrightarrow_{\beta}$ as the compatible closure of β-contraction of type operators w.r.t. all the type and kind formers. We write $\longrightarrow^{*}_{\beta}$ for its reflexive, transitive closure, *β-reduction*.

3.2 Declarative Typing and Kinding

The static semantics of $F^{\omega}_{..}$ are summarized in Figs. 3 and 4. We refer to this set of judgments as the *declarative* system, as opposed to the *canonical* system introduced in §5. We sometimes write $\Gamma \vdash \mathcal{J}$ to denote an arbitrary judgment of the declarative system. Throughout the paper, we silently assume that judgments are *well-scoped*, i.e. if $\Gamma \vdash \mathcal{J}$ then $\mathsf{fv}(\mathcal{J}) \subseteq \mathsf{dom}(\Gamma)$. We now discuss each of the judgments, emphasizing novel rules.

3.2.1 Context and Kind Formation. The rules for context formation Γ ctx are standard. They ensure that the type and kind annotations of all bindings are well-formed. The rules of the remaining judgments are set up so that they can only be derived in well-formed contexts.

Our kind formation judgment $\Gamma \vdash K$ kd ensures that (a) all types appearing in K are well-kinded, and (b) that the bounds of intervals are proper types (not λs), forbidding for instance $\bot\,..\,\lambda X{:}K.\,A$. The formation rule Wf-DArr for dependent arrows is standard. An interval $A\,..\,B$ is well-formed if A and B are proper types. As discussed in §2.2, we choose not to enforce $A \leq B$; e.g. the empty kind $\varnothing = \top\,..\,\bot$ is well-formed. We say that the bounds of an interval $A\,..\,B$ are *consistent* in Γ if $\Gamma \vdash A \leq B : *$ and *inconsistent* otherwise. If A and B are closed types and $\nvdash A \leq B : *$ in the empty context, we say that the bounds of $A\,..\,B$ are *absurd*. For example, the bounds of $*$ are always consistent while those of $\top\,..\,\bot$ are absurd.[5] The bounds of $X\,..\,Y$ are inconsistent in $\Gamma = X{:}*, Y{:}*$ but not absurd because X and Y are open types.

3.2.2 Kinding and Typing. The kinding rules K-Var, K-Top, K-Bot, K-Arr and K-All are all standard. The rule K-All resembles that found in F_{ω}: no bound annotations are needed because the bounds of X are internalized in the kind K. Similarly, K-Abs and K-App, resemble those in F_{ω} more than those in $F^{\omega}_{<:}$. The rules K-Sing and K-Sub are used to adjust the kind of a type: K-Sub is the kind-level analog of T-Sub (subsumption); K-Sing resembles Stone and Harper's singleton introduction rule [2000]. Note that K-Sing only *narrows* the kind of a type whereas K-Sub only *widens* it. The premise of K-Sing may look a bit surprising: why use $\Gamma \vdash A : B\,..\,C$ instead of $\Gamma \vdash A : *$? This extra flexibility is necessary to prove that types inhabiting intervals are proper types, i.e. that $\Gamma \vdash A : B\,..\,C$ implies $\Gamma \vdash A : *$. Thus the relaxed premise justifies itself.

The typing rules are again entirely standard, with the possible exception of some additional context and kind formation premises that would be redundant in variants of F_{ω} with simple kinds.

3.2.3 Subkinding and Subtyping. $F^{\omega}_{..}$ features both subtyping and *subkinding*. The use of subkinding in $F^{\omega}_{..}$ is both natural and essential. If a type is contained in an interval $A\,..\,B$, then one naturally expects it to also be contained in a *wider* interval $A'\,..\,B'$ where $A' \leq A$ and $B \leq B'$. This is captured in the rule SK-Intv. Subkinding is also essential. It is thanks to SK-Intv that we can express bounded polymorphism and bounded type operators in $F^{\omega}_{..}$. Consider e.g. a polymorphic

[5] See Lemma 5.4 in §5.2.

Context formation $\boxed{\Gamma\ \mathsf{ctx}}$

$$\frac{}{\varnothing\ \mathsf{ctx}}\ (\text{C-Empty}) \qquad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash K\ \mathsf{kd}}{\Gamma, X{:}K\ \mathsf{ctx}}\ (\text{C-TmBind}) \qquad \frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A : *}{\Gamma, x{:}A\ \mathsf{ctx}}\ (\text{C-TpBind})$$

Kind formation $\boxed{\Gamma \vdash K\ \mathsf{kd}}$

$$\frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \mathrel{..} B\ \mathsf{kd}}\ (\text{Wf-Intv}) \qquad \frac{\Gamma \vdash J\ \mathsf{kd} \qquad \Gamma, X{:}J \vdash K\ \mathsf{kd}}{\Gamma \vdash (X{:}J) \to K\ \mathsf{kd}}\ (\text{Wf-DArr})$$

Kinding $\boxed{\Gamma \vdash A : K}$

$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma(X) = K}{\Gamma \vdash X : K}\ (\text{K-Var}) \qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \top : *}\ (\text{K-Top}) \qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \bot : *}\ (\text{K-Bot})$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \to B : *}\ (\text{K-Arr}) \qquad \frac{\Gamma \vdash K\ \mathsf{kd} \qquad \Gamma, X{:}K \vdash A : *}{\Gamma \vdash \forall X{:}K.\, A : *}\ (\text{K-All})$$

$$\frac{\begin{array}{c}\Gamma \vdash J\ \mathsf{kd} \qquad \Gamma, X{:}J \vdash A : K \\ \colorbox{lightgray}{$\Gamma, X{:}J \vdash K\ \mathsf{kd}$}\end{array}}{\Gamma \vdash \lambda X{:}J.\, A : (X{:}J) \to K}\ (\text{K-Abs}) \qquad \frac{\begin{array}{c}\Gamma \vdash A : (X{:}J) \to K \qquad \Gamma \vdash B : J \\ \colorbox{lightgray}{$\Gamma, X{:}J \vdash K\ \mathsf{kd} \qquad \Gamma \vdash K[B/X]\ \mathsf{kd}$}\end{array}}{\Gamma \vdash A\, B : K[B/X]}\ (\text{K-App})$$

$$\frac{\Gamma \vdash A : B \mathrel{..} C}{\Gamma \vdash A : A \mathrel{..} A}\ (\text{K-Sing}) \qquad \frac{\Gamma \vdash A : J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash A : K}\ (\text{K-Sub})$$

Typing $\boxed{\Gamma \vdash t : A}$

$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash K\ \mathsf{kd} \qquad \Gamma, X{:}K \vdash t : A}{\Gamma \vdash \lambda X{:}K.\, t : \forall X{:}K.\, A}\ (\text{T-TAbs})$$

$$\frac{\begin{array}{c}\Gamma \vdash A : * \qquad \Gamma \vdash B : * \\ \Gamma, x{:}A \vdash t : B\end{array}}{\Gamma \vdash \lambda x{:}A.\, t : A \to B}\ (\text{T-Abs}) \qquad \frac{\Gamma \vdash t : \forall X{:}K.\, A \qquad \Gamma \vdash B : K}{\Gamma \vdash t\, B : A[B/X]}\ (\text{T-TApp})$$

$$\frac{\Gamma \vdash s : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\, t : B}\ (\text{T-App}) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash t : B}\ (\text{T-Sub})$$

Fig. 3. Declarative presentation of $F^{\omega}_{..}$ – part 1. Premises in gray are validity conditions (see §3.3).

term $t : \forall X{:}\bot \mathrel{..} A.\, B$ and a type argument C such that $C \leq A : *$. We can apply t to C because C has kind $C \mathrel{..} C$ (by K-Sing) which in turn is a subkind of $\bot \mathrel{..} A$ (by SK-Intv). The rule SK-DArr lifts the interval containment order through dependent arrow kinds. It resembles Aspinall and Compagnoni's subtyping rule (s-π) for dependent product types Aspinall and Compagnoni [2001].

Subtyping judgments $\Gamma \vdash A \leq B : K$ are indexed by the common kind K in which A and B are related. Note that two types may be related in some kinds but not others. For example, the extremal types $\top$ and $\bot$ are related as proper types, i.e. $\vdash \bot \leq \top : *$, but not as inhabitants of

Subkinding $\boxed{\Gamma \vdash J \leq K}$

$$\frac{\Gamma \vdash A_2 \leq A_1 : * \qquad \Gamma \vdash B_1 \leq B_2 : *}{\Gamma \vdash A_1 \mathbin{..} B_1 \leq A_2 \mathbin{..} B_2}\ (\text{SK-Intv}) \qquad \frac{\begin{array}{c}\Gamma \vdash (X{:}J_1) \to K_1 \text{ kd} \\ \Gamma \vdash J_2 \leq J_1 \qquad \Gamma, X{:}J_2 \vdash K_1 \leq K_2\end{array}}{\Gamma \vdash (X{:}J_1) \to K_1 \leq (X{:}J_2) \to K_2}\ (\text{SK-DArr})$$

Subtyping $\boxed{\Gamma \vdash A \leq B : K}$

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash A \leq A : K}\ (\text{ST-Refl}) \qquad \frac{\Gamma \vdash A \leq B : K \qquad \Gamma \vdash B \leq C : K}{\Gamma \vdash A \leq C : K}\ (\text{ST-Trans})$$

$$\frac{\Gamma \vdash A : B \mathbin{..} C}{\Gamma \vdash A \leq \top : *}\ (\text{ST-Top}) \qquad \frac{\Gamma \vdash A : B \mathbin{..} C}{\Gamma \vdash \bot \leq A : *}\ (\text{ST-Bot})$$

$$\frac{\begin{array}{c}\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J \\ \Gamma \vdash A[B/X] : K[B/X] \\ \Gamma, X{:}J \vdash K \text{ kd} \qquad \Gamma \vdash K[B/X] \text{ kd}\end{array}}{\Gamma \vdash (\lambda X{:}J.\, A)\, B \leq A[B/X] : K[B/X]}\ (\text{ST-}\beta_1) \qquad \frac{\begin{array}{c}\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J \\ \Gamma \vdash A[B/X] : K[B/X] \\ \Gamma, X{:}J \vdash K \text{ kd} \qquad \Gamma \vdash K[B/X] \text{ kd}\end{array}}{\Gamma \vdash A[B/X] \leq (\lambda X{:}J.\, A)\, B : K[B/X]}\ (\text{ST-}\beta_2)$$

$$\frac{\Gamma \vdash A : (X{:}J) \to K \qquad X \notin \mathsf{fv}(A)}{\Gamma \vdash \lambda X{:}J.\, A\, X \leq A : (X{:}J) \to K}\ (\text{ST-}\eta_1) \qquad \frac{\Gamma \vdash A : (X{:}J) \to K \qquad X \notin \mathsf{fv}(A)}{\Gamma \vdash A \leq \lambda X{:}J.\, A\, X : (X{:}J) \to K}\ (\text{ST-}\eta_2)$$

$$\frac{\Gamma \vdash A_2 \leq A_1 : * \qquad \Gamma \vdash B_1 \leq B_2 : *}{\Gamma \vdash A_1 \to B_1 \leq A_2 \to B_2 : *}\ (\text{ST-Arr}) \qquad \frac{\begin{array}{c}\Gamma \vdash \forall X{:}K_1.\, A_1 : * \\ \Gamma \vdash K_2 \leq K_1 \quad \Gamma, X{:}K_2 \vdash A_1 \leq A_2 : *\end{array}}{\Gamma \vdash \forall X{:}K_1.\, A_1 \leq \forall X{:}K_2.\, A_2 : *}\ (\text{ST-All})$$

$$\frac{\begin{array}{c}\Gamma \vdash \lambda X{:}J_1.\, A_1 : (X{:}J) \to K \\ \Gamma \vdash \lambda X{:}J_2.\, A_2 : (X{:}J) \to K \\ \Gamma, X{:}J \vdash A_1 \leq A_2 : K\end{array}}{\Gamma \vdash \lambda X{:}J_1.\, A_1 \leq \lambda X{:}J_2.\, A_2 : (X{:}J) \to K}\ (\text{ST-Abs}) \qquad \frac{\begin{array}{c}\Gamma \vdash A_1 \leq A_2 : (X{:}J) \to K \qquad \Gamma \vdash B_1 = B_2 : J \\ \Gamma \vdash B_1 : J \quad \Gamma, X{:}J \vdash K \text{ kd} \quad \Gamma \vdash K[B_1/X] \text{ kd}\end{array}}{\Gamma \vdash A_1\, B_1 \leq A_2\, B_2 : K[B_1/X]}\ (\text{ST-App})$$

$$\frac{\Gamma \vdash A : B_1 \mathbin{..} B_2}{\Gamma \vdash B_1 \leq A : *}\ (\text{ST-Bnd}_1) \qquad \frac{\Gamma \vdash A : B_1 \mathbin{..} B_2}{\Gamma \vdash A \leq B_2 : *}\ (\text{ST-Bnd}_2)$$

$$\frac{\Gamma \vdash A_1 \leq A_2 : B \mathbin{..} C}{\Gamma \vdash A_1 \leq A_2 : A_1 \mathbin{..} A_2}\ (\text{ST-Intv}) \qquad \frac{\Gamma \vdash A_1 \leq A_2 : J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash A_1 \leq A_2 : K}\ (\text{ST-Sub})$$

Kind equality $\boxed{\Gamma \vdash J = K}$ **Type equality** $\boxed{\Gamma \vdash A = B : K}$

$$\frac{\Gamma \vdash J \leq K \qquad \Gamma \vdash K \leq J}{\Gamma \vdash J = K}\ (\text{SK-AntiSym}) \qquad \frac{\Gamma \vdash A \leq B : K \qquad \Gamma \vdash B \leq A : K}{\Gamma \vdash A = B : K}\ (\text{ST-AntiSym})$$

Fig. 4. Declarative presentation of $F^{\omega}_{\cdot\cdot}$ – part 2. Premises in gray are validity conditions (see §3.3).

their respective singleton kinds $\bot\,..\,\bot$ and $\top\,..\,\top$. For a given context Γ and kind K, the subtyping relation $\Gamma \vdash A \leq B : K$ is a preorder, as witnessed by the rules ST-Refl and ST-Trans. Note that there are no such rules for subkinding, but it is easy to prove them admissible. In §5, we will see that some (but not all) instances of ST-Refl and ST-Trans can be eliminated too.

The rules ST-Top and ST-Bot establish $\top$ and $\bot$ as the maximum and minimum proper types w.r.t. subtyping. The premise $\Gamma \vdash A : B\,..\,C$ ensures that the extremal types are only related to other proper types. As for K-Sing, the kind $B\,..\,C$ in the premise allows us to prove that types inhabiting intervals are proper types:

$$\dfrac{(\text{K-Sing})\ \dfrac{\Gamma \vdash A : B\,..\,C}{\Gamma \vdash A : A\,..\,A} \qquad \dfrac{(\text{ST-Bot})\ \dfrac{\Gamma \vdash A : B\,..\,C}{\Gamma \vdash \bot \leq A : *} \qquad \dfrac{\Gamma \vdash A : B\,..\,C}{\Gamma \vdash A \leq \top : *}\ (\text{ST-Top})}{\Gamma \vdash A\,..\,A \leq \bot\,..\,\top}\ (\text{SK-Intv})}{\Gamma \vdash A : \bot\,..\,\top}\ (\text{K-Sub})$$

This derivation would not be possible if the rules K-Sing, ST-Bot or ST-Top had premise $\Gamma \vdash A : *$.

The rules ST-β_1 and ST-β_2 correspond to β-contraction and expansion, respectively. Two separate rules are needed because subtyping is not symmetric. We could have combined them into a single type equality rule but that would have complicated the definition of type equality. Similarly, the rules ST-η_1 and ST-η_2 relate η-convertible types. The rule for universals resembles SK-DArr.

Most variants of $F^{\omega}_{<:}$ separate subtyping of type operator applications into a subtyping rule that only compares the heads of applications, and a congruence rule for type equality w.r.t. application. Here we fuse these two rules into a single subtyping rule ST-App. Since we do not track the variance of type operators, the arguments must be equal types. Because arrow kinds are dependent, either B_1 or B_2 must be substituted for X in K in the conclusion. Both are equally suitable; we pick B_1.

The rule for subtyping operator abstractions, ST-Abs, is maybe the most unusual when compared to other variants of F_{ω} since it allows abstractions to be subtypes even if their domain annotations J_1 and J_2 are not subkinds. Other systems adopt weaker versions of this rule where $J \equiv J_2$ or even $J \equiv J_1 \equiv J_2$. But such rules are not suitable for a theory featuring both subkinding and η-equality. Let $A : * \to *$ be an operator and $B : *$ a type in Γ. By K-Sub, ST-Sub, ST-Eta$_{1,2}$ and antisymmetry,

$$\Gamma \;\vdash\; \lambda X{:}\bot\,..\,B.\, A\,X \;=\; A \;=\; \lambda X{:}B\,..\,\top.\, A\,X \;:\; (X{:}B\,..\,B) \to *$$

i.e. the two η-expansions of A are equal as types, despite having distinct domain annotations. Because the η-rules allow such equations, we adopt a compatible subtyping rule for abstractions. The first two premises of ST-Abs ensure that both abstractions – irrespective of their domain annotations – inhabit the common arrow kind $(X{:}J) \to K$; the remaining premise ensures that the bodies of the two abstractions are pointwise subtypes assuming the common domain $X{:}J$. Note that systems without domain annotations avoid such complications [cf. Abel 2008].

So far, we have seen how a type interval $A\,..\,B$ can be formed using Wf-Intv, and introduced using K-Sing, but we have yet to see how the bounds A and B of the interval can be put to use. Type intervals are "eliminated" by turning them into subtyping judgments via a pair of *bound projection* rules ST-Bnd$_1$ and ST-Bnd$_2$. Given a type $A : B\,..\,C$, the rules ST-Bnd$_1$ and ST-Bnd$_2$ assert that B and C are indeed lower and upper bounds, respectively, of A. When A is a variable, we may use rule ST-Bnd$_2$ to derive judgments of the form $\Gamma \vdash X \leq C$, similar to those obtained using the variable subtyping rule from $F_{\leq}$. More generally, the bound projection rules allow us to *reflect* any well-formed assumption $\Gamma(X) = A\,..\,B$ – consistent or not – into a corresponding subtyping judgment $\Gamma \vdash A \leq B : *$. We discuss the ramifications this has for type safety in §3.5.

As for kinding judgments, there are two subtyping rules that allow us to adjust the kinds of subtyping judgments, ST-Sub and ST-Intv. The former is the analog of K-Sub for subtyping, whereas the latter is the subtyping counterpart of the interval introduction rule K-Sing: if A and B are subtypes in some interval $C\,..\,D$, then surely they are still subtypes in the interval $A\,..\,B$ bounded by those very same types. Indeed, $A \mathrel{..} B$ is the smallest interval in which the two types are related. The ST-Intv rule plays an important role in the proof of subject reduction for types and kinds (Theorem 3.3) because it allows us to relate β-equal types inhabiting singleton kinds.

3.2.4 Kind and Type Equality. The kind and type equality judgments are each generated by exactly one rule: SK-AntiSym for kind equality and ST-AntiSym for type equality. In most variants of F_ω with subtyping, the subtyping relation is not defined to be antisymmetric. Instead antisymmetry may or may not be an admissible property that has to be proven [cf. Compagnoni and Goguen 1999]. In $F^\omega_{..}$, antisymmetry is not an admissible property, however. To see this, consider the context $\Gamma = X{:}A \mathrel{..} A$ for some proper type A. Then $\Gamma \vdash X : A \mathrel{..} A$, and we can derive that X and A are mutual subtypes using ST-Bnd$_1$ and ST-Bnd$_2$. But without antisymmetry we have no way to derive $\Gamma \vdash X = A : *$. Faced with this issue, we could have chosen to add a *singleton reflection* rule for deriving $\Gamma \vdash X = A : *$ from $\Gamma \vdash X : A \mathrel{..} A$ directly, such as the one due to Stone and Harper [2000]. Interestingly, antisymmetry for proper types is derivable from Stone and Harper's rule and other rules about type intervals. We conjecture that antisymmetry of subtyping under arbitrary kinds would also have been admissible in such a system, albeit at the cost of a more complicated type equality judgment. We prefer the simpler judgment with an explicit antisymmetry rule.

3.3 Basic Metatheoretic Properties

With the dynamics and statics in place, we can begin our work on the metatheory of $F^\omega_{..}$. Our system enjoys the usual basic metatheoretic properties, such as preservation of all the judgments under weakening, substitution and narrowing of contexts, as well as admissibility of the missing order-theoretic and congruence rules for subkinding, kind equality and type equality. Although these properties constitute the foundation on which we build the remainder of our metatheory, they are also entirely standard, and little insight is gained by spelling them out in detail. We therefore relegate them to Appendix B. There, the reader will also find a collection of admissible rules that justify the encodings of the higher-order extrema and interval kinds given in §3.1.1. These include formation, subtyping and subkinding rules for the encoded kinds, and typing rules for introducing and eliminating the more familiar forms of bounded universals. Unlike the other admissible rules, they are not important for the remainder of the metatheoretic development.

There are two standard properties of the declarative system that are exceptional in that their proofs are not routine inductions, namely *validity* of the various judgments and *functionality* of substitutions. Roughly, a judgment is valid if all its parts are well-formed.

Lemma 3.1 (validity).

(kinding validity) *If* $\Gamma \vdash A : K$, *then* Γ ctx *and* $\Gamma \vdash K$ kd.
(typing validity) *If* $\Gamma \vdash t : A$, *then* Γ ctx *and* $\Gamma \vdash A : *$.
(kind (in)equation validity) *If* $\Gamma \vdash J = K$ *or* $\Gamma \vdash J \leq K$, *then* $\Gamma \vdash J$ kd *and* $\Gamma \vdash K$ kd.
(type (in)equation validity) *If* $\Gamma \vdash A = B : K$ *or* $\Gamma \vdash A \leq B : K$, *then* $\Gamma \vdash A : K$ *and* $\Gamma \vdash B : K$.

The validity lemma provides a "sanity check" for the static semantics, but it also plays a crucial role in the proofs of other important properties, such as subject reduction or soundness of type normalization. Unfortunately, it is harder to prove than one might expect. The proofs of kinding, subkinding and subtyping validity require the following *functionality* lemma for the case of ST-App.

Lemma 3.2 (functionality). *Let* $\Gamma \vdash A_1 = A_2 : K$.

(1) If $\Gamma, X{:}K \vdash J$ kd, *then* $\Gamma \vdash J[A_1/X] = J[A_2/X]$.
(2) If $\Gamma, X{:}K \vdash B : J$, *then* $\Gamma \vdash B[A_1/X] = B[A_2/X] : J[A_1/X]$.

The proof of functionality, in turn, depends on kinding and subtyping validity. Proving the two statements by simultaneous induction is not enough to resolve the circular dependency because the proof of functionality would require us to apply the IH to derivations obtained via validity. Since these are not generally sub-derivations of the relevant premise, the induction does not go through.

Instead, we follow Harper and Pfenning [2005] and establish validity by "temporarily extending" certain rules of the declarative system with additional premises, which we call *validity conditions*, shown in gray in Figs. 3 and 4. We then prove functionality and validity for the extended system, show that the two systems are equivalent and the validity conditions are redundant after all, and obtain Lemma 3.1 for the original system. For details see Appendix B.3.

3.4 Subject Reduction for Well-Kinded Types

For most versions of F_ω, subject reduction for types is easy to prove because types are simply-kinded. In $F^{\omega}_{\cdot\cdot}$, the proof is complicated by the presence of type-dependent kinds and subkinding. However these complications are minor since there are only two *shapes* of kinds – intervals and arrows – with exactly one subkinding rule per shape. Hence subkinding is easy to invert.

To prove subject reduction, we show that β-reduction steps can be lifted to type and kind equality.

Theorem 3.3.

(1) If $\Gamma \vdash J$ kd *and* $J \longrightarrow_\beta K$*, then* $\Gamma \vdash J = K$.
(2) If $\Gamma \vdash A : K$ *and* $A \longrightarrow_\beta B$*, then* $\Gamma \vdash A = B : K$.

Subject reduction for kinds and types then follows immediately from Theorem 3.3 and validity.

Corollary 3.4 (subject reduction for kinding)**.**

(1) If $\Gamma \vdash J$ kd *and* $J \longrightarrow^*_\beta K$*, then* $\Gamma \vdash K$ kd.
(2) If $\Gamma \vdash A : K$ *and* $A \longrightarrow^*_\beta B$*, then* $\Gamma \vdash B : K$.

3.5 The Long Road to Type Safety

After establishing subject reduction for well-kinded types, we prove type safety via *progress* and *preservation* (aka subject reduction) [Wright and Felleisen 1994]. But as we show in this section, we must first weaken the statement of preservation for it to hold in $F^{\omega}_{\cdot\cdot}$.

Preservation typically applies to all open terms:

Proposition 3.1 (preservation)**.** *If* $\Gamma \vdash t : A$ *and* $t \longrightarrow_{\mathsf{v}} t'$*, then* $\Gamma \vdash t' : A$.

However, in $F^{\omega}_{\cdot\cdot}$ this statement fails because reduction of open terms is unsafe. The culprit are type variable bindings with absurd bounds. Consider the following example. Let v be the polymorphic identity function $v = \lambda X{:}*.\, \lambda x{:}X.\, x$ which is of type $A = \forall X{:}*.\, X \to X$. In $F^{\omega}_{\cdot\cdot}$, closed universals are not subtypes of closed arrows;[6] hence v cannot be applied to itself. For the same reason, the term application $t = (\lambda x{:}A.\, x)\, v\, v$ is ill-typed as a closed term. Yet, t is well-typed in the context $\Gamma = X{:}(A \to A) \mathrel{..} (A \to A \to A)$ because we can use ST-Bnd$_1$ and ST-Bnd$_2$ to derive $\Gamma \vdash A \to A \leq X \leq A \to A \to A : *$ and subsumption to derive $\Gamma \vdash \lambda x{:}A.\, x : A \to A \to A : *$. Note that the bounds of X are proper types, so its kind is well-formed, as is the context Γ. But since $\nvdash A \leq A \to A : *$, the bounds of the interval are absurd.

Next, consider what happens when t takes a reduction step.

$$(\lambda x{:}A.\, x)\, v\, v \;\longrightarrow_{\mathsf{v}}\; (x[v/x])\, v \;\equiv\; v\, v.$$

According to preservation, the application $v\, v$ should have type A, but instead it is ill-typed, even in Γ. The assumption $X : (A \to A) \mathrel{..} (A \to A \to A)$ is useless here, since v does not have type

[6] See Lemma 5.4 in §5.2.

$A \longrightarrow A$. Not only is $v\,v$ ill-typed, it is also stuck. Hence $v\,v$ is neither a value nor can it be reduced further – type safety clearly does not hold in Γ.

But all is not lost. Type safety still holds for closed terms, as does a weaker form of preservation.

Proposition 3.2 (preservation – weak version)**.** *If* $\vdash t : A$ *and* $t \longrightarrow_{\mathsf{v}} t'$, *then* $\vdash t' : A$.

Throughout the next two sections, we will work our way towards a proof of this proposition.

3.6 Challenges and Proof Strategy

To conclude the section, let us briefly explore the challenges involved in proving weak preservation and our strategy to address them. The complexity of the subtyping relation throws a spanner in the works when we try to prove weak preservation for cases where β-contractions occur. To prove these cases, one normally starts by showing that the following rules are admissible:

$$\frac{\Gamma \vdash A_1 \longrightarrow B_1 \leq A_2 \longrightarrow B_2 : *}{\Gamma \vdash A_2 \leq A_1 : * \qquad \Gamma \vdash B_1 \leq B_2 : *} \qquad \frac{\Gamma \vdash \forall X{:}K_1.\, A_1 \leq \forall X{:}K_2.\, A_2 : *}{\Gamma \vdash K_2 \leq K_1 \qquad \Gamma, X{:}K_2 \vdash A_1 \leq A_2 : *}$$

These properties are generally known as *inversion of subtyping*, and are closely related to the Π-injectivity property, which is a well-known source of complexity in dependent type theories. There are several features of subtyping that severely complicate the proof of subtyping inversion.

(1) The rules for β and η-conversion, together with transitivity, may change the shapes of related types in the middle of a subtyping derivation, e.g. from a type former to a type application.

$$\Gamma \;\vdash\; A_1 \longrightarrow A_2 \;\leq\; (\lambda X{:}*.\, X \longrightarrow A_2)\, A_1 \;\leq\; \cdots \;\leq\; (\lambda X{:}*.\, X \longrightarrow B_2)\, B_1 \;\leq\; B_1 \longrightarrow B_2 \;:\; *$$

(2) The subsumption rule ST-Sub may change the kinds of related types at any time.

(3) As outlined above, we can derive judgments of the form $\Gamma \vdash A \leq X \leq B : *$ where A and B need not be of the same shape, from absurd assumptions in Γ. For example

$$\Gamma \;\vdash\; A \longrightarrow B \;\leq\; X \;\leq\; \forall X{:}K.\, C \;:\; *$$

We address these points as follows. First, we eliminate uses of the $\beta\eta$ rules (1) by adopting an alternative presentation of subtyping which we dub *canonical subtyping* (§5). Canonical subtyping only relates types in *η-long β-normal form* (§4), so there is no need for $\beta\eta$ rules. This approach works in variants of F_ω with η-conversion [cf. Abel and Rodriguez 2008], whereas rewriting-based proofs of Π-injectivity [e.g. Adams 2006; Barendregt 1992] do not generalize readily to our setting.

The canonical presentation of subtyping also restricts the placement of subsumption (2) to certain strategic positions, just as in algorithmic or bidirectional subtyping (§5).

Finally, we avoid issues caused by absurd bounds (3) by proving subtyping inversion only in the empty context, i.e. only for closed types. That suffices for proving weak preservation and, as the above example illustrates, it is the best we can do in a system with inequality reflection (§5.2).

The core challenges of the proof of subtyping inversion thus consists in showing that well-formed kinds and well-kinded types have normal forms (§4), so that the canonical and declarative versions of subtyping can be proven equivalent (§5). We address these challenges in the next two sections.

4 NORMALIZATION OF TYPES

As discussed in the previous section, we cannot prove inversion of subtyping directly, because $\beta\eta$-convertible types and kinds differ in their syntactic structure. We address this problem in two steps: (1) in this section, we show that types and kinds in $F^{\omega}_{\cdot\cdot}$ can be reduced to $\beta\eta$-normal form via a *bottom-up normalization* procedure based on *hereditary substitution*; (2) in next section, we give a canonical presentation of subtyping that only relates types in normal form.

4.1 Syntax

We begin by introducing an alternative syntax for types that is better suited to our definition of hereditary substitution. The key difference is that the type arguments of applications are grouped together in sequences called *spines*. Hence, we refer to this presentation of types as *spine form.*

$$D, E ::= F\,\mathbf{E} \qquad F, G ::= X \mid \top \mid \bot \mid D \to E \mid \forall X{:}K.\,E \mid \lambda X{:}K.\,E \qquad \mathbf{D}, \mathbf{E} ::= \epsilon \mid D, \mathbf{E}$$

Applications form a separate syntactic category, *eliminations* E, and consist of a *head* F and a spine $\mathbf{E}$. A head is any type former that is not an application. We adopt vector notation for spines, writing $\mathbf{E}$ for the sequence $\mathbf{E} = E_1, E_2, \dots, E_n$ and $(\mathbf{D}, \mathbf{E})$ for the concatenation of $\mathbf{D}$ and $\mathbf{E}$.

The two representations of types are isomorphic, so we mix them freely, knowing that explicit conversions can always be inserted where necessary.

4.2 Hereditary Substitution in Raw Types

In the §5, we will introduce a system of canonical judgments defined directly on normal forms. Since kinds in $F^{\omega}_{\cdot\cdot}$ are dependent, some of the kinding and subtyping rules involve substitutions in kinds, e.g. K-App or ST-$\beta_{1,2}$. Unfortunately, substitutions do not preserve normal forms because substituting an operator abstraction for the head of a neutral type introduces a new redex. For example $(Y\,A)[\lambda X{:}K.\,B/Y] \equiv (\lambda X{:}K.\,B)\,A$ is not a normal form, even if $Y\,A$ and $\lambda X{:}K.\,B$ are. To define a canonical counterpart of e.g. K-App directly on normal kinds and types, we need a variant of substitution that immediately eliminates the β-redexes it introduces. This type of substitution operation is known as *hereditary substitution* [Watkins et al. 2004]. The challenge in defining a hereditary substitution function is, of course, ensuring its totality.

Our definition of hereditary substitution is given in Fig. 5. Hereditary substitution is defined mutually with *reducing application* of eliminations by recursion on the structure of the shape k. The definitions of the three hereditary substitution functions proceed by inner recursion on the structure of the parameters K, D and $\mathbf{D}$, respectively. Note that there are some recursive calls where no parameter decreases, but one can check (and we have done so) that at least one of the relevant parameters decreases strictly along every cycle in the call graph. Hence the five functions remain structurally recursive, ensuring their totality. Note the crucial use of spine forms: $D \cdot^k \mathbf{E}$ simultaneously unwinds $\mathbf{E}$ and k using the fact that $\mathbf{E}$ matches the right-associative structure of k.

Our presentation of hereditary substitution differs from others in the literature. Like Keller and Altenkirch [2010], we define hereditary substitution by structural recursion and mutually with reducing application. But their definition is based on an intrinsically typed representation, which does not readily generalize to a system with dependent types (or kinds). Instead, like Abel and Rodriguez [2008] we define hereditary substitution directly on raw (i.e. unkinded) types, so our definition contains degenerate cases; unlike Abel and Rodriguez's, our definition is structurally recursive hence easier to mechanize. Our approach of defining hereditary substitution by recursion on shapes rather than (dependent) kinds was inspired by Harper and Licata's formalization of Canonical LF [2007]. However, they define hereditary substitution as an inductive relation, thus they avoid degenerate cases but must establish functionality and termination separately.

Because the essential difference between ordinary and hereditary substitution is that the latter reduces newly created β-redexes, the results of the two operations are β-convertible.

Lemma 4.1. *Let E be an elimination, X a type variable and k a shape, then*

(1) $K[E/X] \longrightarrow^*_\beta K[E/X^k]$ for any kind K; *(3) $E\,D \longrightarrow^*_\beta E \cdot^k D$ for any type D;*
(2) $D[E/X] \longrightarrow^*_\beta D[E/X^k]$ for any type D; *(4) $E\,\mathbf{D} \longrightarrow^*_\beta E \cdot^k \mathbf{D}$ for any spine $\mathbf{D}$.*

It is an immediate consequence of Lemma 4.1 and subject reduction that ordinary and hereditary substitutions produce judgmentally equal results.

Hereditary substitution $\boxed{D[E/X^k]}$

$$
\begin{aligned}
(Y\,\mathbf{D})[E/X^k] &= \begin{cases} E \cdot^k (\mathbf{D}[E/X^k]) & \text{if } Y = X,\\ Y\,(\mathbf{D}[E/X^k]) & \text{otherwise,}\end{cases}\\
(\top\,\mathbf{D})[E/X^k] &= \top\,(\mathbf{D}[E/X^k])\\
(\bot\,\mathbf{D})[E/X^k] &= \bot\,(\mathbf{D}[E/X^k])\\
((D_1 \to D_2)\,\mathbf{D})[E/X^k] &= (D_1[E/X^k] \to D_2[E/X^k])\,(\mathbf{D}[E/X^k])\\
((\forall Y{:}K.\,D')\,\mathbf{D})[E/X^k] &= (\forall Y{:}K[E/X^k].\,D'[E/X^k])\,(\mathbf{D}[E/X^k]) && \text{for } Y \neq X, Y \notin \mathsf{fv}(E),\\
((\lambda Y{:}K.\,D')\,\mathbf{D})[E/X^k] &= (\lambda Y{:}K[E/X^k].\,D'[E/X^k])\,(\mathbf{D}[E/X^k]) && \text{for } Y \neq X, Y \notin \mathsf{fv}(E).
\end{aligned}
$$

$\boxed{\mathbf{D}[E/X^k]}$

$$
\begin{aligned}
\epsilon[E/X^k] &= \epsilon\\
(D', \mathbf{D})[E/X^k] &= (D'[E/X^k]), (\mathbf{D}[E/X^k])
\end{aligned}
$$

$\boxed{K[E/X^k]}$

$$
\begin{aligned}
(D_1 \mathrel{..} D_2)[E/X^k] &= D_1[E/X^k] \mathrel{..} D_2[E/X^k]\\
((Y{:}J) \to K)[E/X^k] &= (Y{:}J[E/X^k]) \to K[E/X^k] && \text{for } Y \neq X, Y \notin \mathsf{fv}(E).
\end{aligned}
$$

Reducing application $\boxed{D \cdot^k E}$

$$
\begin{aligned}
D \cdot^* \quad E &= D\,E\\
D \cdot^{k \to l} E &= \begin{cases} D'[E/X^k] & \text{if } D = \lambda X{:}J.\,D',\\ D\,E & \text{otherwise.}\end{cases}
\end{aligned}
$$

$\boxed{D \cdot^k \mathbf{E}}$

$$
\begin{aligned}
D \cdot^k \epsilon &= D\\
D \cdot^k (E, \mathbf{E}) &= \begin{cases} (D \cdot^{k_1 \to k_2} E) \cdot^{k_2} \mathbf{E} & \text{if } k = k_1 \to k_2,\\ D\,(E, \mathbf{E}) & \text{otherwise.}\end{cases}
\end{aligned}
$$

Fig. 5. Hereditary substitution and reducing application

Corollary 4.2 (soundness of hereditary substitution)**.** *Let* $\Gamma \vdash A : K$*, then*

(1) if $\Gamma, X{:}K, \Delta \vdash J$ kd*, then* $\Gamma, \Delta[A/X] \vdash J[A/X] = J[A/X^{|K|}]$*;*

(2) if $\Gamma, X{:}K, \Delta \vdash B : J$*, then* $\Gamma, \Delta[A/X] \vdash B[A/X] = B[A/X^{|K|}] : J[A/X]$.

4.3 Normalization of Raw Types

Based on hereditary substitution, we define a bottom-up *normalization* function nf on kinds and types. It is a straightforward extension of the normalization function by Abel and Rodriguez [2008] to dependent kinds. The function nf is defined directly on raw types and kinds and relies on a separate function for η-expanding variables. The definition of both functions is given in Fig. 6.

The η-expansion $\eta_K(A)$ of a type A of kind K is defined by recursion on the structure of K. It is used in the definition of nf to expand type variables. Note that $\eta_K(A)$ immediately expands newly introduced argument variables to produce η-long forms. Normalization $\mathsf{nf}_\Gamma(A)$ and $\mathsf{nf}_\Gamma(K)$ of raw types and kinds in context Γ are defined by mutual recursion on A and K, respectively. The case of applications uses hereditary substitution to eliminate β-redexes. Note the crucial use of domain-annotations: in order to hereditarily substitute a type argument in the body of an operator abstraction $\lambda X{:}K.\,E$, we need to guess its shape, or equivalently, the shape of X. Since the normalization function is defined directly on raw, unkinded types, the only way to obtain this information is from the kind annotation K in the abstraction.

The context parameter Γ of nf is used to look up the declared kinds of variables, which drive their η-expansion. To ensure that the resulting η-expansions are normal, the context Γ must itself

η-expansion $\boxed{\eta_K(A)}$

$$\eta_{D_1\,..\,D_2}(A) \;=\; A \qquad\qquad \eta_{(X:J)\to K}(A) \;=\; \lambda X{:}J.\,\eta_K(A\,(\eta_J(X))) \qquad \text{for } X \notin \mathsf{fv}(A).$$

Normalization $\boxed{\mathsf{nf}_\Gamma(A)}$

$$\begin{array}{lcll}
\mathsf{nf}_\Gamma(X) & = & \eta_{\Gamma(X)}(X) & \text{if } X \in \mathsf{dom}(\Gamma),\\
 & & X & \text{otherwise,}\\
\mathsf{nf}_\Gamma(A) & = & A & \text{for } A \in \{\bot, \top\},\\
\mathsf{nf}_\Gamma(A \to B) & = & \mathsf{nf}_\Gamma(A) \to \mathsf{nf}_\Gamma(B) & \\
\mathsf{nf}_\Gamma(\forall X{:}K.\,A) & = & \forall X{:}K'.\,\mathsf{nf}_{\Gamma,X:K'}(A) & \text{where } K' = \mathsf{nf}_\Gamma(K),\\
\mathsf{nf}_\Gamma(\lambda X{:}K.\,A) & = & \lambda X{:}K'.\,\mathsf{nf}_{\Gamma,X:K'}(A) & \text{where } K' = \mathsf{nf}_\Gamma(K),\\
\mathsf{nf}_\Gamma(A\,B) & = & E[\mathsf{nf}_\Gamma(B)/X^{|K|}] & \text{if } \mathsf{nf}_\Gamma(A) = \lambda X{:}K.\,E,\\
 & & (\mathsf{nf}_\Gamma(A))\,(\mathsf{nf}_\Gamma(B)) & \text{otherwise.}
\end{array}$$

$\boxed{\mathsf{nf}_\Gamma(K)}$

$$\begin{array}{lcll}
\mathsf{nf}_\Gamma(A\,..\,B) & = & \mathsf{nf}_\Gamma(A)\,..\,\mathsf{nf}_\Gamma(B) & \\
\mathsf{nf}_\Gamma((X{:}J) \to K) & = & (X{:}J') \to \mathsf{nf}_{\Gamma,X:J'}(K) & \text{where } J' = \mathsf{nf}_\Gamma(J).
\end{array}$$

Fig. 6. Normalization of types and kinds

be normal. We therefore extend normalization pointwise to contexts, defining $\mathsf{nf}(\Gamma)$ as

$$\mathsf{nf}(\varnothing) = \varnothing \qquad \mathsf{nf}(\Gamma, x{:}A) = \mathsf{nf}(\Gamma), x{:}\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A) \qquad \mathsf{nf}(\Gamma, X{:}K) = \mathsf{nf}(\Gamma), X{:}\mathsf{nf}_{\mathsf{nf}(\Gamma)}(K)$$

Since nf is a total function defined directly on raw types and kinds, it necessarily contains degenerate cases, i.e. the resulting types need not be β-normal. For example, the case of applications relies on the domain annotations K of operator abstractions $\lambda X{:}K.\,A$ in head position to be truthful. The ill-kinded type $\Omega = (\lambda X{:}*.\,X\,X)(\lambda X{:}*.\,X\,X)$ will result in $\mathsf{nf}(\Omega) \equiv (X\,X)[\lambda X{:}*.\,X\,X/X^*] \equiv (\lambda X{:}*.\,X\,X) \cdot^* (\lambda X{:}*.\,X\,X) \equiv \Omega$. However, for well-kinded types $\Gamma \vdash A : K$, the type $\mathsf{nf}_\Gamma(A)$ is guaranteed to be an η-long β-normal form, as we will see in §5 (cf. Lemma 5.3).

Furthermore, η-expansion and normalization are *sound*, i.e. they do not alter the meaning of types and kinds. In particular, well-kinded types and well-formed kinds are judgmentally equal to their normalized counterparts. The proof relies on soundness of hereditary substitutions.

Lemma 4.3 (soundness of normalization)**.**

(1) If $\Gamma \vdash K$ kd*, then* $\Gamma \vdash K = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(K)$*.* *(2) If* $\Gamma \vdash A : K$*, then* $\Gamma \vdash A = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A) : K$*.*

4.4 Commutativity of Normalization and Hereditary Substitution

We are now almost ready to introduce the canonical presentation of $F^\omega_{.\,.}$. Our final task in this section is to establish a series of *commutativity properties* of hereditary substitution and normalization. They say, roughly, that the order of these operations can be switched without changing the result. We require these properties to prove equivalence of the canonical and declarative systems. For example, to prove that ST-Beta$_1$ is admissible in the canonical system, we must show that

$$\Gamma \;\vdash\; \mathsf{nf}_\Gamma(A)[\mathsf{nf}_\Gamma(B)/X^{|J|}] \;=\; \mathsf{nf}_\Gamma(A[B/X]) \;:\; \mathsf{nf}_\Gamma(K[B/X]).$$

Since normalization involves hereditary substitutions, we must further show that these preserve the canonical judgments. The case for applications involves kind equations of the form

$$\Gamma \ \vdash \ K[A/Y^{k}][B/X^{j}] \ = \ K[B/X^{j}][A[B/X^{j}]/Y^{k}].$$

Unfortunately, the commutativity properties do not hold for arbitrary raw types and kinds. The reasons are twofold. First, our definition of hereditary substitution contains degenerate cases for ill-kinded inputs, which can cause inconsistencies when we commute hereditary substitutions. For example, it's easy to verify that for $B = \lambda Z{:}\ast \to \ast.\, Z$

$$(X\,A)[Y/X^{\ast}][B/Y^{\ast\to\ast}] \ \equiv \ A \ \not\equiv \ B\,A \ \equiv \ (X\,A)[B/Y^{\ast\to\ast}][Y[B/Y^{\ast\to\ast}]/X^{\ast}].$$

Second, normalization involves η-expansion and, as we have seen in §3.2, η-expansions of the same type variable can differ syntactically in their domain annotations.

We address the two problems separately. For the former, we adopt the approach taken by Abel and Rodriguez [2008], namely to prove commutativity of hereditary substitutions only for well-kinded normal forms. To apply their technique, we first need to show that hereditary substitutions preserve kinding (of normal forms). This is easy in their setting, which is simply kinded, but challenging in ours because our kinding rules involve substitutions in dependent kinds. A direct proof that hereditary substitutions preserve kinding would require the very commutativity lemmas we are trying to establish. We circumvent this issue by relaxing our requirements: for $e[V/Y^{k}]$ to be non-degenerate, the normal form V need not actually be well-kinded; it only needs to have *shape* k. Using this insight, we prove commutativity of hereditary substitutions in 4 steps.

(1) We define a *simple kinding judgment* that assign shapes (rather than kinds) to normal forms.
(2) We show that hereditary substitution preserves simple kinding. Because shapes have no type dependencies, the proof does not require any commutativity lemmas.
(3) We show that hereditary substitutions in simply kinded normal forms commute.
(4) We show that simple kinding is sound: every type of kind K has a normal form of shape $|K|$.

We refer the reader to Appendix C for details.

It remains to show that normalization commutes with substitution. To work around the issue of incongruous domain annotations, we introduce an auxiliary equivalence $A \approx B$ on types and kinds, called *weak equality*, that identifies operator abstractions up to the *shape* of their domain annotations, i.e. $\lambda X{:}J.\,A \ \approx \ \lambda X{:}K.\,A$ when $|J| \ \equiv \ |K|$. Substitution then *weakly commutes* with normalization of well-formed kinds and well-kinded types.

Lemma 4.4. *Let* $\Gamma \vdash A : J$ *and* $V = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$*, then*

(1) if $\Gamma, X{:}J, \Delta \vdash K\ \mathsf{kd}$*, then* $\mathsf{nf}_{\mathsf{nf}(\Gamma,\Delta[A/X])}(K[A/X]) \approx (\mathsf{nf}_{\mathsf{nf}(\Gamma,X:J,\Delta)}(K))[V/X^{|J|}]$*;*
(2) if $\Gamma, X{:}J, \Delta \vdash B : K$*, then* $\mathsf{nf}_{\mathsf{nf}(\Gamma,\Delta[A/X])}(B[A/X]) \approx (\mathsf{nf}_{\mathsf{nf}(\Gamma,X:J,\Delta)}(B))[V/X^{|J|}]$*;*

The proof uses commutativity of hereditary substitutions and a few helper lemmas, e.g. that substitutions weakly commute with η-expansion. The full proof is given in Appendix C.2.3.

5 THE CANONICAL SYSTEM

Having characterized the normal forms of kinds and types in the previous section, we now present our *canonical system* of judgments directly defined on normal forms, and summarize its most important metatheoretic properties: the hereditary substitution lemma, equivalence w.r.t. the declarative system, and inversion of subtyping. We conclude the section by revisiting the type safety proof outlined in §3.

The rules for canonical kinding, subtyping and spine equality are given in Figs. 7 and 8. The canonical system also contains judgments for kind and context formation, subkinding, and type and kind equality, but the rules of those judgments are analogous to their declarative counterparts,

Canonical kinding of variables

$$\boxed{\Gamma \vdash_{\mathsf{var}} X : K}$$

$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma(X) = K}{\Gamma \vdash_{\mathsf{var}} X : K}\ (\text{CV-Var}) \qquad \frac{\Gamma \vdash_{\mathsf{var}} X : J \qquad \Gamma \vdash J \leq K \qquad \Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash_{\mathsf{var}} X : K}\ (\text{CV-Sub})$$

Spine kinding

$$\boxed{\Gamma \vdash J \rightrightarrows \mathbf{V} \rightrightarrows K}$$

$$\frac{}{\Gamma \vdash K \rightrightarrows \epsilon \rightrightarrows K}\ (\text{CK-Empty}) \qquad \frac{\Gamma \vdash U \leftleftarrows J \qquad \Gamma \vdash J\ \mathsf{kd} \qquad \Gamma \vdash K[U/X^{|J|}] \rightrightarrows \mathbf{V} \rightrightarrows L}{\Gamma \vdash (X{:}J) \to K \rightrightarrows U, \mathbf{V} \rightrightarrows L}\ (\text{CK-Cons})$$

Kinding of neutral types

$$\boxed{\Gamma \vdash_{\mathsf{ne}} N : K}$$

$$\frac{\Gamma \vdash_{\mathsf{var}} X : J \qquad \Gamma \vdash J \rightrightarrows \mathbf{V} \rightrightarrows K}{\Gamma \vdash_{\mathsf{ne}} X\,\mathbf{V} : K}\ (\text{CK-Ne})$$

Kinding checking

$$\boxed{\Gamma \vdash V \leftleftarrows K}$$

$$\frac{\Gamma \vdash V \rightrightarrows J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash V \leftleftarrows K}\ (\text{CK-Sub})$$

Kind synthesis for normal types

$$\boxed{\Gamma \vdash V \rightrightarrows K}$$

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \top \rightrightarrows \top\,..\,\top}\ (\text{CK-Top}) \qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \bot \rightrightarrows \bot\,..\,\bot}\ (\text{CK-Bot})$$

$$\frac{\Gamma \vdash U \rightrightarrows U\,..\,U \qquad \Gamma \vdash V \rightrightarrows V\,..\,V}{\Gamma \vdash U \to V \rightrightarrows (U \to V)\,..\,(U \to V)}\ (\text{CK-Arr}) \qquad \frac{\Gamma \vdash K\ \mathsf{kd} \qquad \Gamma, X{:}K \vdash V \rightrightarrows V\,..\,V}{\Gamma \vdash \forall X{:}K.\,V \rightrightarrows (\forall X{:}K.\,V)\,..\,(\forall X{:}K.\,V)}\ (\text{CK-All})$$

$$\frac{\Gamma \vdash J\ \mathsf{kd} \qquad \Gamma, X{:}J \vdash V \rightrightarrows K}{\Gamma \vdash \lambda X{:}J.\,V \rightrightarrows (X{:}J) \to K}\ (\text{CK-Abs}) \qquad \frac{\Gamma \vdash_{\mathsf{ne}} N : U\,..\,V}{\Gamma \vdash N \rightrightarrows N\,..\,N}\ (\text{CK-Sing})$$

Fig. 7. Canonical presentation of $F^{\omega}_{..}$ – part 1

so we omit them. We use the following naming conventions to distinguish normal forms: U, V, W denote normal types; M, N denote neutral types. No special notation is used for normal kinds.

The judgments for kinding, subtyping and spine equality are bidirectional. Double arrows are used to indicate whether a kind is an input ($K \rightrightarrows$ or $\leftleftarrows K$) or an output ($\rightrightarrows K$) of the judgment. The contexts and subjects of judgments are always considered inputs. The rules for kind synthesis are similar to those for declarative kinding, except that the synthesized kinds are more precise, that there is no subsumption rule, and that kinding of variables and applications has been combined into a single rule CK-Sing for kinding neutral proper types. A quick inspection of the rules reveals that all synthesized kinds are singletons. In particular, $\Gamma \vdash V \rightrightarrows V\,..\,V$ for proper types V. The kind checking judgment $\Gamma \vdash V \leftleftarrows K$ has only one inference rule: the subsumption rule CK-Sub.

The canonical kinding judgments $\Gamma \vdash_{\mathsf{var}} X : K$ for variables and $\Gamma \vdash_{\mathsf{ne}} N : K$ for neutral types are not directed because of the presence of the subsumption rule CV-Sub. While this rule is not actually necessary for variable kinding, it considerably simplifies the metatheory. Without it, the proof of context narrowing would circularly depend on at least three other lemmas – transitivity of subkinding, functionality and the hereditary substitution lemma – all of which use somewhat idiosyncratic, possibly incompatible induction strategies.

Subtyping of proper types $\boxed{\Gamma \vdash U \leq V}$

$$\frac{\Gamma \vdash V \rightrightarrows V \mathrel{..} V}{\Gamma \vdash V \leq \top}\;(\text{CST-Top}) \qquad \frac{\Gamma \vdash V \rightrightarrows V \mathrel{..} V}{\Gamma \vdash \bot \leq V}\;(\text{CST-Bot})$$

$$\frac{\Gamma \vdash U \leq V \qquad \Gamma \vdash V \leq W}{\Gamma \vdash U \leq W}\;(\text{CST-Trans}) \qquad \frac{\Gamma \vdash U_2 \leq U_1 \qquad \Gamma \vdash V_1 \leq V_2}{\Gamma \vdash U_1 \to V_1 \leq U_2 \to V_2}\;(\text{CST-Arr})$$

$$\frac{\Gamma \vdash_{\mathsf{var}} X : K \qquad \Gamma \vdash K \rightrightarrows \mathbf{V}_1 = \mathbf{V}_2 \rightrightarrows U \mathrel{..} W}{\Gamma \vdash X\,\mathbf{V}_1 \leq X\,\mathbf{V}_2}\;(\text{CST-Ne}) \qquad \frac{\begin{array}{c}\Gamma \vdash \forall X{:}K_1.\,V_1 \rightrightarrows \forall X{:}K_1.\,V_1 \mathrel{..} \forall X{:}K_1.\,V_1\\ \Gamma \vdash K_2 \leq K_1 \qquad \Gamma, X{:}K_2 \vdash V_1 \leq V_2\end{array}}{\Gamma \vdash \forall X{:}K_1.\,V_1 \leq \forall X{:}K_2.\,V_2}\;(\text{CST-All})$$

$$\frac{\Gamma \vdash_{\mathsf{ne}} N : V_1 \mathrel{..} V_2}{\Gamma \vdash V_1 \leq N}\;(\text{CST-Bnd}_1) \qquad \frac{\Gamma \vdash_{\mathsf{ne}} N : V_1 \mathrel{..} V_2}{\Gamma \vdash N \leq V_2}\;(\text{CST-Bnd}_2)$$

Checked subtyping $\boxed{\Gamma \vdash U \leq V \leftleftarrows K}$

$$\frac{\begin{array}{c}\Gamma \vdash V_1 \leftleftarrows U \mathrel{..} W\\ \Gamma \vdash V_2 \leftleftarrows U \mathrel{..} W\\ \Gamma \vdash V_1 \leq V_2\end{array}}{\Gamma \vdash V_1 \leq V_2 \leftleftarrows U \mathrel{..} W}\;(\text{CST-Intv}) \qquad \frac{\begin{array}{c}\Gamma \vdash \lambda X{:}J_1.\,V_1 \leftleftarrows (X{:}J) \to K\\ \Gamma \vdash \lambda X{:}J_2.\,V_2 \leftleftarrows (X{:}J) \to K\\ \Gamma, X{:}J \vdash V_1 \leq V_2 \leftleftarrows K\end{array}}{\Gamma \vdash \lambda X{:}J_1.\,V_1 \leq \lambda X{:}J_2.\,V_2 \leftleftarrows (X{:}J) \to K}\;(\text{CST-Abs})$$

Spine equality $\boxed{\Gamma \vdash J \rightrightarrows \mathbf{U} = \mathbf{V} \rightrightarrows K}$

$$\frac{}{\Gamma \vdash K \rightrightarrows \epsilon = \epsilon \rightrightarrows K}\;(\text{SpEq-Empty}) \qquad \frac{\Gamma \vdash U_1 = U_2 \leftleftarrows J \qquad \Gamma \vdash K[U_1/X^{\,|J|}] \rightrightarrows \mathbf{V}_1 = \mathbf{V}_2 \rightrightarrows L}{\Gamma \vdash (X{:}J) \to K \rightrightarrows U_1, \mathbf{V}_1 = U_2, \mathbf{V}_2 \rightrightarrows L}\;(\text{SpEq-Cons})$$

Fig. 8. Canonical presentation of $F^{\omega}_{..}$ – part 2

Canonical spine kinding $\Gamma \vdash J \rightrightarrows \mathbf{V} \rightrightarrows K$ differs from the other judgments in that it features both an input kind J and an output kind K. When an operator of shape J is applied to the spine $\mathbf{V}$, the resulting type is of shape K – as exemplified by the rule CK-Ne. In CK-Cons, the head U of the spine $U, \mathbf{V}$ is hereditarily substituted for X in the codomain K of the overall input kind $(X{:}J) \to K$ to obtain the input kind $K[U/X^{\,|J|}]$ for kinding the tail $\mathbf{V}$ of the spine. The use of hereditary (rather than ordinary) substitution ensures that the resulting kind remains normal.

The rules for canonical subtyping are divided into two judgments: *subtyping of proper types* $\Gamma \vdash U \leq V$ and *checked subtyping* $\Gamma \vdash U \leq V \leftleftarrows K$. The separate judgment for proper subtyping $\Gamma \vdash U \leq V$ simplifies the metatheory by disentangling subtyping and kinding. It resembles the subtyping judgment of $F_{\leq}$. Notable differences are the two bound projection rules CST-Bnd$_1$ and CST-Bnd$_2$, which replace the variable subtyping rule, and the rule CST-Ne for subtyping neutrals. The most interesting of these is CST-Ne. It says that two neutral types $X\,\mathbf{V}_1$ and $X\,\mathbf{V}_2$ headed by a common type variable X are subtypes if they have canonically equal spines. Importantly, the spines $\mathbf{V}_1$ and $\mathbf{V}_2$ need not be syntactically equal. In $F^{\omega}_{..}$, normal forms may be judgmentally equal yet differ syntactically, e.g. because they have different domain annotations (via CST-Abs), because one of the types is declared as a type alias of the other (via a singleton kind), or because one of the types can be proven to alias the other due to inconsistent bounds: $\Gamma,\ X\colon U\mathinner{..}V,\ Y\colon V\mathinner{..}U,\ \Delta \vdash\ U = V$. The last example illustrates that there is no easy way for the normalization function nf to resolve such equations. In systems without dependent kinds and equality reflection, judgmentally equal types have syntactically equal normal forms, so that CST-Ne can be omitted [see e.g. Abel and Rodriguez 2008]; in systems with singleton kinds (but no type intervals), type aliases can be resolved during normalization [see e.g. Stone and Harper 2000]. Neither of these apply in $F^{\omega}_{\cdot\cdot}$. Unfortunately, the presence of CST-Ne complicates the metatheory of the canonical system (cf. §5.1.1).

The checked subtyping judgment $\Gamma \vdash U \le V \ \leftleftarrows\ K$ is kind-directed. The kind K determines whether U and V are compared as proper types (CST-Intv) or type operators (CST-Abs). The rule CST-Intv checks that the types V_1 and V_2 have the expected kind $U\mathinner{..}W$ and are proper subtypes. Because normal types are η-long, the only normal types of arrow kind are operator abstractions. They are compared using rule CST-Abs, exactly as in the declarative system. The rules of the *spine equality* judgment $\Gamma \vdash J \ \rightrightarrows\ \mathbf{U} = \mathbf{V} \ \rightrightarrows\ K$ resemble those of spine kinding.

5.1 Metatheoretic Properties of the Canonical System

It is easy to show that the canonical system is sound w.r.t. the declarative one, i.e. that normal forms related by the canonical judgments are also related by the declarative counterparts. To avoid confusion, we mark canonical judgments with the subscript "c" and declarative ones with "d" in the following soundness lemma. The full statement of the lemma has 13 parts, one for each canonical judgment; we only show the most important ones.

Lemma 5.1 (soundness of the canonical rules – excerpt)**.**

(1) If $\Gamma \vdash_{\mathrm{c}} V \ \rightrightarrows\ K$ *or* $\Gamma \vdash_{\mathrm{c}} V \ \leftleftarrows\ K$*, then* $\Gamma \vdash_{\mathrm{d}} V : K$.
(2) If $\Gamma \vdash_{\mathrm{c}} U \le V$*, then* $\Gamma \vdash_{\mathrm{d}} U \le V : *$.
(3) If $\Gamma \vdash_{\mathrm{c}} U \le V \ \leftleftarrows\ K$*, then* $\Gamma \vdash_{\mathrm{d}} U \le V : K$.

Many of the basic properties of the declarative system – weakening, context narrowing, admissible order-theoretic rules, many validity properties, and the commutativity lemmas from the previous section – also hold in the canonical system, and their proofs carry over with minor modifications. Full statements and proofs are given in Appendix D.1. Notable exceptions are the substitution and functionality lemmas. These do not hold because ordinary substitutions do not preserve normal forms. Instead, we need to establish analogous lemmas for hereditary substitutions.

5.1.1 The Hereditary Substitution Lemma.

The most important metatheoretic property of the canonical system is the *hereditary substitution lemma*, which states, roughly, that canonical judgments are preserved by hereditary substitutions of canonically well-kinded types. It is key to proving completeness w.r.t. the declarative system, and thus to our overall goal of establishing type safety. Proving and even stating the hereditary substitution lemma is challenging. The full statement of the lemma has 24 separate parts, all of which have to be proven simultaneously. The large number of canonical judgments is one reason for the complexity of the lemma. But the foremost reason is that the proof of the hereditary substitution lemma circularly depends on *functionality of the canonical judgments*. This circular dependency is caused by the subtyping rule CST-Ne.

To illustrate this, assume we are given two normal forms V and W such that $\Gamma \vdash V = W \ \leftleftarrows\ *$, but $V \not\equiv W$ syntactically. We have seen examples of such normal forms V and W earlier. Assume further that there is some X with $\Gamma(X) = * \to *$ and consider what happens when we hereditarily substitute the operator $\lambda Y{:}*.\,U$ for X in the judgment $\Gamma \vdash X\,V \le X\,W$ obtained using CST-Ne. We would like to show that hereditary substitution preserves this inequation, i.e. that

$$\Gamma\ \vdash\ (X\,V)[\lambda Y{:}*.\,U/X^{*\to*}]\ \equiv\ U[V/Y^{*}]\ \le\ U[W/Y^{*}]\ \equiv\ (X\,W)[\lambda Y{:}*.\,U/X^{*\to*}],$$

which requires functionality. The example illustrates a second point, namely that, in order to prove that hereditary substitutions preserve canonical kinding and subtyping, we need to prove that kinding and subtyping of reducing applications is admissible. Our hereditary substitution lemma must cover all of these properties, leading to the aforementioned grand total of 24 parts. We give a small excerpt here, illustrating some of the properties just discussed.

Lemma 5.2 (hereditary substitution – excerpt). *Assume* $\Gamma \vdash U_1 \ = \ U_2 \ \Leftarrow \ K$.

(1) *Hereditary substitution preserves kind checking:*
if $\Gamma, X{:}K, \Delta \vdash V \ \Leftarrow \ J$, *then* $\Gamma, \Delta[U_1/X^{|K|}] \vdash V[U_1/X^{|K|}] \ \Leftarrow \ J[U_1/X^{|K|}]$.

(2) *Functionality/monotonicity of hereditary substitution:*
if $\Gamma, X{:}K, \Delta \vdash V \ \Leftarrow \ J$ *and* $\Gamma, X{:}K, \Delta \vdash J \ \mathsf{kd}$, *then*
$$\Gamma, \Delta[U_1/X^{|K|}] \vdash V[U_1/X^{|K|}] \leq V[U_2/X^{|K|}] \ \Leftarrow \ J[U_1/X^{|K|}].$$

(3) *Hereditary substitution preserves checked subtyping:*
if $\Gamma, X{:}K, \Delta \vdash V_1 \leq V_2 \ \Leftarrow \ J$ *and* $\Gamma, X{:}K, \Delta \vdash J \ \mathsf{kd}$, *then*
$$\Gamma, \Delta[U_1/X^{|K|}] \vdash V_1[U_1/X^{|K|}] \leq V_2[U_2/X^{|K|}] \ \Leftarrow \ J[U_1/X^{|K|}].$$

(4) *Canonical equality of reducing applications is admissible:*
if $\Gamma \vdash V_1 = V_2 \ \Leftarrow \ (X{:}K) \to J$, *then* $\Gamma \vdash V_1 \cdot^{|K| \to |J|} U_1 = V_2 \cdot^{|K| \to |J|} U_2 \ \Leftarrow \ J[U_1/X^{|K|}]$.

The structure of the proof mirrors that of the recursive definition of hereditary substitution itself. All 24 parts are proven simultaneously by induction in the structure of the shape $|K|$. Most parts proceed by an inner induction on the derivations of the judgments into which U_1 and U_2 are being substituted. The cases involving the rules CK-Cons and SpEq-Cons, rely on commutativity of hereditary substitutions in kinds. Details are given in Appendix D.2.

Just as for the declarative system, the proof of functionality enables us to prove some additional validity properties, which, in turn, are necessary to prove completeness of the canonical system.

5.1.2 Completeness of Canonical Kinding. In §4, we saw that every declaratively well-kinded type has a judgmentally equal $\beta\eta$-normal form (Lemmas 4.3). To establish equivalence of the canonical and declarative systems, it remains to show that the normal forms of types related by a declarative judgment are also canonically related. The full statement of the completeness lemma has 11 parts, one for each declarative judgment plus three auxiliary parts for dealing with hereditary substitutions and $\beta\eta$-conversions. The most important ones are the following, where we again use the subscripts “c” and “d” to distinguish canonical judgments from declarative ones.

Lemma 5.3 (completeness of the canonical rules – excerpt).

(1) *If* $\Gamma \vdash_{\mathsf{d}} A : K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) \ \Leftarrow \ \mathsf{nf}(K)$.

(2) *If* $\Gamma \vdash_{\mathsf{d}} A \leq B : K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) \leq \mathsf{nf}(B) \ \Leftarrow \ \mathsf{nf}(K)$.

The completeness proof relies on Lemma 5.2 to show that the normal forms of applications are canonically well-kinded, and on the weak commutativity properties established in §4.4 to show that β- and η-conversions are admissible in the canonical system. In addition, the proof relies on the validity conditions discussed in §3.3. The full statement and proof of the completeness lemma are given in Appendix D.3.

5.2 Inversion of Subtyping and Type Safety

As we saw in §3, reductions in open terms are unsafe because variable bindings with inconsistent bounds can inject arbitrary inequations into the subtyping relation. Under such assumptions, subtyping cannot be inverted in any meaningful way. We therefore prove inversion of subtyping only in the empty context, following the approach by Rompf and Amin [2016]:

(1) We introduce a helper judgment $\vdash_{\mathrm{tf}} U \leq V$, which states that U is a proper subtype of V in the empty context. It is obtained from the canonical subtyping judgment $\Gamma \vdash U \leq V$ by fixing $\Gamma = \varnothing$ and removing CST-Trans and any rules involving free variables (CST-Ne, CST-Bnd$_1$ and CST-Bnd$_2$). Soundness of $\vdash_{\mathrm{tf}} U \leq V$ w.r.t. canonical subtyping is immediate.
(2) We prove that transitivity is admissible in $\vdash_{\mathrm{tf}} U \leq V$. This is straightforward since there are no $\beta\eta$-conversion rules or bound projection rules that get in the way.
(3) Because transitivity is admissible, it is straightforward to establish completeness, and thus equivalence of $\vdash_{\mathrm{tf}} U \leq V$ w.r.t. canonical subtyping.
(4) Inversion of the canonical subtyping relation in the empty context then follows immediately by inspection of the rules for $\vdash_{\mathrm{tf}} U \leq V$ and equivalence of the two judgments.
(5) Inversion of top-level declarative subtyping follows by equivalence of canonical and declarative subtyping and soundness of normalization.

Lemma 5.4 (inversion of top-level subtyping). *Let* $\varnothing \vdash A_1 \leq A_2 : *$.

(1) If $A_1 = B_1 \to C_1$ *and* $A_2 = B_2 \to C_2$*, then* $\varnothing \vdash B_2 \leq B_1 : *$ *and* $\varnothing \vdash C_1 \leq C_2 : *$.
(2) If $A_1 = \forall X{:}K_1.\, B_1$ *and* $A_2 = \forall X{:}K_2.\, B_2$*, then* $\varnothing \vdash K_2 \leq K_1$ *and* $X{:}K_2 \vdash B_1 \leq B_2 : *$.
(3) $\varnothing \nvdash \top \leq \bot$*, and* $\varnothing \nvdash A \to B \leq \forall X{:}K.\, C$*, and* $\varnothing \nvdash \forall X{:}K.\, A \leq B \to C$.

With subtyping inversion in place, we are finally ready to prove type safety of $F^{\omega}_{\cdot\cdot}$.

Theorem 5.5 (type safety). *Well-typed terms do not get stuck.*

(progress) *If* $\vdash t : A$*, then either* t *is a value, or* $t \longrightarrow_{\mathrm{v}} t'$ *for some term* t'.
(weak preservation) *If* $\vdash t : A$ *and* $t \longrightarrow_{\mathrm{v}} t'$*, then* $\vdash t' : A$.

The proofs are standard. Details are given in Appendix D.4.

6 UNDECIDABILITY OF SUBTYPING

In this section, we prove that subtyping for $F^{\omega}_{\cdot\cdot}$ is undecidable. The culprit is equality reflection via the bound projection rules ST-Bnd$_1$ and ST-Bnd$_2$ (cf. §2.2). Following Castellan et al. [2015], we prove undecidability of subtyping by reduction from convertibility of SK combinator terms. The *SK combinator calculus* has only three term formers $t ::= \mathsf{S} \mid \mathsf{K} \mid s\,t$ and two equational axioms: $\mathsf{S}\,s\,t\,u =_{\mathrm{SK}} s\,u\,(t\,u)$ and $\mathsf{K}\,s\,t =_{\mathrm{SK}} s$. Yet SK is Turing-complete, and convertibility of SK terms is undecidable. We embed SK terms and equations into $F^{\omega}_{\cdot\cdot}$ via the following declarations:

$$\begin{aligned}
\Gamma_{\mathrm{SK}} \;&\coloneqq\; S : *, \qquad K : *, \qquad \odot : * \to * \to *,\\
&\quad S_{\mathrm{r}} : (X{:}*) \to (Y{:}*) \to (Z{:}*) \to S \odot X \odot Y \odot Z \;..\; X \odot Z \odot (Y \odot Z),\\
&\quad S_{\mathrm{e}} : (X{:}*) \to (Y{:}*) \to (Z{:}*) \to X \odot Z \odot (Y \odot Z) \;..\; S \odot X \odot Y \odot Z,\\
&\quad K_{\mathrm{r}} : (X{:}*) \to (Y{:}*) \to K \odot X \odot Y \;..\; X, \qquad K_{\mathrm{e}} : (X{:}*) \to (Y{:}*) \to X \;..\; K \odot X \odot Y.\\
[\![\mathsf{S}]\!] \;&\coloneqq\; S \qquad [\![\mathsf{K}]\!] \;\coloneqq\; K \qquad [\![s\,t]\!] \;\coloneqq\; [\![s]\!] \odot [\![t]\!].
\end{aligned}$$

The map $[\![-]\!]$ encodes SK terms as types under Γ_{SK} and induces a reduction from SK convertibility $s =_{\mathrm{SK}} t$ to subtyping $\Gamma_{\mathrm{SK}} \vdash [\![s]\!] \leq [\![t]\!] : *$, which can be used to prove undecidability of subtyping.

Theorem 6.1 (undecidability). *Let* s, t *be SK terms. Then* $\Gamma_{\mathrm{SK}} \vdash [\![s]\!] \leq [\![t]\!] : *$ *iff* $s =_{\mathrm{SK}} t$.

It is easy to verify the "if" direction. For example, the contraction law for K corresponds to the inequation $\Gamma_{\mathrm{SK}} \vdash K \odot [\![s]\!] \odot [\![t]\!] \;\leq\; K_{\mathrm{r}}\,[\![s]\!]\,[\![t]\!] \;\leq\; [\![s]\!] : *$. The "only if" direction is more challenging. The complexity of subtyping derivations precludes a direct decoding into $=_{\mathrm{SK}}$ for many of the same reasons that a direct proof of subtyping inversion is unfeasible. In addition, types such as

$\top$ or $\forall X{:}K.\,A$ that are not encodings of SK terms can appear as intermediate expression along a subtyping derivation. For example, note the fleeting appearance of $\top$ in the following.

$$\Gamma_{\mathsf{SK}} \ \vdash \ [\![\mathsf{S}]\!] \ \equiv \ S \ \leq \ K \odot S \odot (K \odot \top \odot S) \ \leq \ K \odot S \odot \top \ \leq \ S \ \equiv \ [\![\mathsf{S}]\!] \ : \ *.$$

To eliminate such spurious appearances of undecodable types, our proof takes a detour through four auxiliary judgment forms.

(1) We first prove undecidability of canonical subtyping to eliminate instances of $\beta\eta$-conversions. Undecidability of declarative subtyping follows by equivalence of the two judgments.

(2) We define a *reduced* version of the canonical system from which we exclude any rules (and judgment forms) that are not relevant to the embedding shown above. For example, any judgments involving higher-order operators are excluded, as are the rules CST-Arr and CST-All and the variable subsumption rule CV-Sub. The reduced system still contains CST-Bot and CST-Top since we cannot rule out intermediate occurrences of these rules a-priori. We show that canonical subtyping derivations under Γ_{SK} can be translated into reduced ones.

(3) We extend the SK term syntax with $\bot$ and $\top$ and define an order $\leq_{\mathsf{SK}}$ on extended terms. The term order is an asymmetric version of $\leq_{\mathsf{SK}}$ which features the rules $\bot \leq_{\mathsf{SK}} t$ and $t \leq_{\mathsf{SK}} \top$. Thanks to these, reduced canonical subtyping derivations can be directly decoded into $\leq_{\mathsf{SK}}$.

(4) We introduce a pair of *parallel reduction relations* $\Rrightarrow_{\leq}$ and $\Rrightarrow_{\geq}$ on the extended syntax. These contain the rules $\bot \Rrightarrow_{\leq} t$ and $\top \Rrightarrow_{\geq} t$ for eliminating occurrences of $\bot$ and $\top$, along with the usual contraction rules for SK terms. Crucially, the reduction rules can eliminate but never introduce instances of $\bot$ and $\top$. Hence, if s is a pure SK term, $s \Rrightarrow t$ implies $s =_{\mathsf{SK}} t$. The parallel reductions enjoy a confluence property w.r.t. the term order: If $s \leq_{\mathsf{SK}} t$, then there is a u such that $s \Rrightarrow^{*}_{\leq} u \Lleftarrow^{*}_{\leq} t$. Via confluence, $s \leq_{\mathsf{SK}} t$ implies $s =_{\mathsf{SK}} t$ for pure s and t.

Thus we have established a chain of implications from which the result follows.

$$\Gamma_{\mathsf{SK}} \vdash [\![s]\!] \leq [\![t]\!] : * \implies \Gamma_{\mathsf{SK}} \vdash_{\mathsf{red}} [\![s]\!] \leq [\![t]\!] \implies s \leq_{\mathsf{SK}} t \implies s \Rrightarrow^{*}_{\leq} u \Lleftarrow^{*}_{\leq} t \implies s =_{\mathsf{SK}} t.$$

For full details, we refer the intrepid reader to the mechanized proof of Theorem 6.1, which is given in the `FOmegaInt.Undecidable` module of our Agda formalization [Stucki and Giarrusso 2021].

7 RELATED WORK

Bounded quantification has been studied extensively through $F_{\leq}$, a variant of System F with bounded quantification, which comes in two flavors: the *Kernel* variant $F_{<:}$ [Cardelli et al. 1991] based on Cardelli and Wegner's *Kernel Fun* [1985] has decidable subtyping, while *Full* $F_{\leq}$ [Curien and Ghelli 1992] features a more expressive subtyping rule for bounded universal quantifiers that renders subtyping undecidable [Pierce 1992]. Recently, Hu and Lhoták [2019] have shown that the $D_{<:}$ calculus – a simplified variant of DOT that uses an expressive $\forall$-subtyping rule – suffers from the same decidability issue as Full $F_{\leq}$. For compatibility with DOT, and knowing that subtyping in $F^{\omega}_{\cdot\cdot}$ is undecidable either way, we also adopt the more expressive rule. The metatheory developed in §§3–6 is largely unaffected by this choice.

An extension of Girard's F_{ω} [1972] with higher-order subtyping and bounded quantification was first proposed by Cardelli [1990] under the name $F^{\omega}_{<:}$. Basic meta theoretic properties of $F^{\omega}_{<:}$ were established by Pierce and Steffen [1997], Compagnoni [1995], and Compagnoni and Goguen [1999]. An extension with bounded operator abstractions ($\mathcal{F}^{\omega}_{\leq}$) has been developed by Compagnoni and Goguen [2003]. More recently, Abel and Rodriguez [2008] developed a variant of $F^{\omega}_{<:}$ where types are identified up to $\beta\eta$-equality and proved its decidability using hereditary substitution. Their work inspired the syntactic approach taken in this paper.

Many of the ideas in $F^{\omega}_{<:}$ go back to early work by Cardelli [1988] on *power types*. Though very expressive, power types render the type language non-normalizing, and in a later work Cardelli and Longo [1991] replaced them with the better behaved *power kinds*. Power kinds can be directly expressed in $F^{\omega}_{..}$ as interval kinds that are bounded by $\bot$ from below: $P(A) = \bot \mathrel{..} A$. Crary [1997] developed an extension of F_{ω} with power kinds as a general calculus for higher-order subtyping. His representations of higher-order bounded quantifiers and operators closely resemble ours.

The notion of *translucency* was introduced by Harper and Lillibridge [1994] in the setting of ML-style modules with sharing constraints. They proposed *translucent sums* as a uniform way of representing translucent type definitions. Stone and Harper [2000] later proposed *singleton kinds* as an alternative mechanism for representing type definitions with sharing constraints. Interval kinds are closely related, conceptually and formally, to Stone and Harper's singleton kinds.

The safety of (in)consistent subtyping constraints in $F_{\leq}$-like systems has been studied in depth by Cretin and Rémy [2014] and Scherer and Rémy [2015]. They formalize two distinct types of subtyping coercions: *coherent* coercions can be erased (i.e. used implicitly) while *incoherent* coercions are introduced and eliminated explicitly. Reductions is allowed (and safe) only under coherent abstractions. It is unclear if their results extend to $\mathcal{F}^{\omega}_{\leq}$-like systems such as ours.

Hereditary substitution is due to Watkins et al. [2004] and has been used to prove weak normalization of a variety of systems. A particularly illuminating example is provided by Keller and Altenkirch [2010] who use it to implement a normalization function for STLC in Agda. Other examples are the work by Abel and Rodriguez [2008] on $F^{\omega}_{<:}$ and the presentation of Canonical LF by Harper and Licata [2007], both of which inspired the metatheoretic development in this paper. Hereditary substitution has also been used to mechanize the equational theory of singleton kinds [Crary 2009] and the semantics of the SML language [Lee et al. 2007] in Twelf.

$F^{\omega}_{..}$ belongs to a long line of calculi developed to model Scala's type system. One of the first to support complex type operators and a form of interval kinds was *Scalina* [Moors et al. 2008b]. Type and kind safety of Scalina was never established but it inspired an extension of Scala's type system with HK types, including higher-order bounded polymorphism, type operators and type definitions [Moors et al. 2008a]. More recently, Amin et al. [2016] introduced the calculus of *Dependent Object Types (DOT)* as a theoretical foundation for Scala and a core calculus for the *Scala 3* compiler [Dotty Team 2020]. Many variants of DOT have been developed, differing in expressiveness and presentation; most come with mechanized type safety proofs [see e.g. Amin 2016; Giarrusso et al. 2020; Rapoport and Lhoták 2019; Rompf and Amin 2016]. Central to all is the notion of *abstract type members*. Because type members can have lower and upper bounds, they provide a form of type intervals. In $F^{\omega}_{..}$, we separate the concept of type intervals from that of abstract type members via interval kinds. DOT admits encodings of some type operators, but none of the DOT calculi developed so far can express general HK types as supported by Scala. But the development of the Scala 3 compiler has shown the need for a principled theory of HK types [Odersky et al. 2016]. In this paper, we have proposed such a theory.

In the future, we wish to extend our work on HK types with existing work on DOT by recombining abstract type members with interval kinds. A type member definition would then be of the form $\{ X : K \}$ where K may be a higher-order interval kind. We expect this to cause new feature interactions, some of which may be problematic. A sketch of such an extension, including a brief discussion of potential issues, can be found in the first author's dissertation [Stucki 2017, Ch. 6].

Another direction for future work is to adapt the techniques developed by Hu and Lhoták [2019] for algorithmic subtyping in the DOT-like calculus $D_{<:}$ to our system. Hu and Lhoták address the problems caused by inconsistent bounds in $D_{<:}$ by replacing the general subtyping transitivity rule with a specialized rule that combines transitivity and bound projection. This isolates the problematic use of inequality reflection in a single rule. They then show that one can obtain a decidable system by removing this rule and weakening the rule for subtyping universals. Not only is this an elegant solution, it also closely reflects the strategy implemented in the Scala compiler.

We do believe that a variant of Hu and Lhoták's strategy could be applied to our system. However, we expect transitivity elimination to be considerably more challenging in our system than in $D_{<:}$, as one would expect in a dependently kinded setting.

8 CONCLUSIONS

We have described $F^{\omega}_{\cdot\cdot}$, a formal theory of higher-order subtyping with type intervals. In $F^{\omega}_{\cdot\cdot}$, type intervals are represented through interval kinds. We showed how interval kinds can be used to encode bounded universal quantification, bounded type operators and singleton kinds. We illustrated the use of interval kinds to abstract over and reflect type inequations and discussed the problems that arise when the corresponding intervals have inconsistent bounds.

We established basic metatheoretic properties of $F^{\omega}_{\cdot\cdot}$. We proved subject reduction in its full generality on the type level, and in a restricted form on the term level. We showed that types and kinds are weakly normalizing by defining a bottom-up normalization procedure on raw kinds and types and proving its soundness. We gave an alternative, canonical presentation of the kind and type level of $F^{\omega}_{\cdot\cdot}$, defined directly on $\beta\eta$-normal forms. We showed that hereditary substitutions preserve canonical judgments and used this result to establish equivalence of the declarative and canonical presentations. We showed that canonical and, by equivalence, declarative subtyping can be inverted in the empty context. Based on these results, we established type safety of $F^{\omega}_{\cdot\cdot}$. We concluded our metatheoretic development by showing that subtyping is undecidable in $F^{\omega}_{\cdot\cdot}$. The metatheory has been fully mechanized in Agda.

Our goal in developing $F^{\omega}_{\cdot\cdot}$ was twofold: study the theory of type intervals for higher-order subtyping, and develop a foundation for Scala's higher-kinded types. We believe that $F^{\omega}_{\cdot\cdot}$ fulfills this goal and constitutes an important step toward a full formalization of Scala's expressive type system. During the development of $F^{\omega}_{\cdot\cdot}$, we discovered a number of minor flaws in Scala 3 (listed in Appendix F). None of these issues constitute critical bugs – in particular, they do not break type safety. But they do illustrate that subtyping in Scala 3 is slightly weaker than necessary, suggesting that there is room for improvement. Indeed, we hope that $F^{\omega}_{\cdot\cdot}$ will serve as a blueprint for a more principled implementation of higher-order subtyping in future versions of Scala.

ACKNOWLEDGMENTS

We owe special thanks to Guillaume Martres for many discussions about this work, for his patience in answering our questions about the Scala 3 type checker, and for his striking ability to produce counterexamples to type system drafts. For insightful discussions and feedback on earlier versions of this work we thank Andreas Abel, Nada Amin, Jesper Cockx, Martin Odersky and François Pottier. We thank the anonymous reviewers for their helpful comments and suggestions. This paper is based upon work supported by the European Research Council (ERC) under Grant 587327 DOPPLER and by the Swedish Research Council (VR) under Grants 2015-04154 PolUser and 2018-04230 Perspex.

REFERENCES

Andreas Abel. 2008. Polarized Subtyping for Sized Types. *Mathematical Structures in Computer Science* 18 (10 2008), 797–822. Issue Special Issue 05. https://doi.org/10.1017/S0960129508006853

Andreas Abel and Dulma Rodriguez. 2008. Syntactic Metatheory of Higher-Order Subtyping. In *Proceedings of the 22nd International Workshop on Computer Science Logic (CSL 2008), 17th Annual Conference of the EACSL, Bertinoro, Italy (LNCS, Vol. 5213)*, Michael Kaminski and Simone Martini (Eds.). Springer, Berlin, Heidelberg, 446–460. https://doi.org/10.1007/978-3-540-87531-4_32

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 2 (2006), 219–246. https://doi.org/10.1017/S0956796805005770

Nada Amin. 2016. *Dependent Object Types.* Ph.D. Dissertation. School of Computer and Communication Sciences, École polytechnique fédérale de Lausanne, Lausanne, Switzerland. https://doi.org/10.5075/epfl-thesis-7156 EPFL thesis no. 7156.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). LNCS, Vol. 9600. Springer International Publishing, Cham, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-dependent Types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014), Portland, Oregon, USA*. ACM, New York, NY, USA, 233–249. https://doi.org/10.1145/2660193.2660216

David Aspinall and Adriana Compagnoni. 2001. Subtyping dependent types. *Theoretical Computer Science* 266, 1-2 (2001), 273–309. https://doi.org/10.1016/S0304-3975(00)00175-4

Hendrik P. Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science*, Samson Abramsky, Dov M. Gabbay, and Thomas S. E Maibaum (Eds.). Vol. 2. Oxford University Press, Oxford, UK, Chapter 2, 117–309.

Luca Cardelli. 1988. Structural Subtyping and the Notion of Power Type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1988), San Diego, California, USA*. ACM, New York, NY, USA, 70–79. https://doi.org/10.1145/73560.73566

Luca Cardelli. 1990. Notes about $F^{\omega}_{<:}$. (October 1990). Unpublished manuscript.

Luca Cardelli and Giuseppe Longo. 1991. A Semantic Basis for Quest. *Journal of Functional Programming* 1, 4 (1991), 417–458. https://doi.org/10.1017/S0956796800000198

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An extension of system F with subtyping. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS 1991), Sendai, Japan*, Takayasu Ito and Albert R. Meyer (Eds.). Springer, Berlin, Heidelberg, 750–770. https://doi.org/10.1007/3-540-54415-1_73

Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Comput. Surveys* 17, 4 (Dec. 1985), 471–523. https://doi.org/10.1145/6041.6042

Simon Castellan, Pierre Clairambault, and Peter Dybjer. 2015. Undecidability of Equality in the Free Locally Cartesian Closed Category. In *13th International Conference on Typed Lambda Calculi and Applications, (TLCA 2015), Warsaw, Poland (LIPIcs, Vol. 38)*, Thorsten Altenkirch (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 138–152. https://doi.org/10.4230/LIPIcs.TLCA.2015.138

Adriana Compagnoni and Healfdene Goguen. 1999. Anti-Symmetry of Higher-Order Subtyping. In *Proceedings of the 13th International Workshop on Computer Science Logic (CSL 1999), 8th Annual Conference of the EACSL Madrid, Spain (LNCS, Vol. 1683)*, Jörg Flum and Mario Rodriguez-Artalejo (Eds.). Springer, Berlin, Heidelberg, 420–438. https://doi.org/10.1007/3-540-48168-0_30

Adriana Compagnoni and Healfdene Goguen. 2003. Typed operational semantics for higher-order subtyping. *Information and Computation* 184, 2 (2003), 242–297. https://doi.org/10.1016/S0890-5401(03)00062-2

Adriana B. Compagnoni. 1995. Decidability of higher-order subtyping with intersection types. In *Computer Science Logic, 8th International Workshop, (CSL 1994), Kazimierz, Poland, September 25–30, 1994, Selected Papers (LNCS, Vol. 933)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer, Berlin, Heidelberg, 46–60. https://doi.org/10.1007/BFb0022246

Karl Crary. 1997. Foundations for the Implementation of Higher-order Subtyping. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP 1997), Amsterdam, The Netherlands*. ACM, New York, NY, USA, 125–135. https://doi.org/10.1145/258948.258961

Karl Crary. 2009. A Syntactic Account of Singleton Types via Hereditary Substitution. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-Languages, Theory and Practice (LFMTP 2009), Montreal, Quebec, Canada*. ACM, New York, NY, USA, 21–29. https://doi.org/10.1145/1577824.1577829

Julien Cretin and Didier Rémy. 2014. System F with Coercion Constraints. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL 2014) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2014), Vienna, Austria*. ACM, New York, NY, USA, Article 34, 10 pages. https://doi.org/10.1145/2603088.2603128

Pierre-Louis Curien and Giorgio Ghelli. 1992. Coherence of Subsumption, Minimum Typing and Type-checking in F≤. *Mathematical Structures in Computer Science* 2, 1 (March 1992), 55–91. https://doi.org/10.1017/S0960129500001134

The Dotty Team. 2020. Scala 3 – A next-generation compiler for Scala – http://dotty.epfl.ch. Source code available from https://github.com/lampepfl/dotty.

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris. *PACMPL* 4, ICFP, Article 114 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408996

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Université Paris VII.

Robert Harper and Daniel R. Licata. 2007. Mechanizing Metatheory in a Logical Framework. *Journal of Functional Programming* 17, 4-5 (July 2007), 613–673. https://doi.org/10.1017/S0956796807006430

Robert Harper and Mark Lillibridge. 1994. A Type-theoretic Approach to Higher-order Modules with Sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1994), Portland, Oregon, USA*. ACM, New York, NY, USA, 123–137. https://doi.org/10.1145/174675.176927

Robert Harper and Frank Pfenning. 2005. On Equivalence and Canonical Forms in the LF Type Theory. *ACM Transactions on Computational Logic* 6, 1 (Jan. 2005), 61–101. https://doi.org/10.1145/1042038.1042041

Jason Z. S. Hu and Ondřej Lhoták. 2019. Undecidability of $D_{<:}$ And Its Decidable Fragments. *PACMPL* 4, POPL, Article 9 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371077

Chantal Keller and Thorsten Altenkirch. 2010. Hereditary Substitutions for Simple Types, Formalized. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming (MSFP 2010), Baltimore, Maryland, USA*. ACM, New York, NY, USA, 3–10. https://doi.org/10.1145/1863597.1863601

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007), Nice, France*. ACM, New York, NY, USA, 173–184. https://doi.org/10.1145/1190216.1190245

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008a. Generics of a Higher Kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA 2008), Nashville, TN, USA*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/1449764.1449798

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008b. Safe type-level abstraction in Scala. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL 2008), San Francisco, CA, USA*. 1–13. https://www.cs.cmu.edu/~aldrich/FOOL/fool08/moors.pdf

Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's type theory*. Vol. 200. Oxford University Press, Oxford, UK.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden.

Martin Odersky, Guillaume Martres, and Dmitry Petrashko. 2016. Implementing Higher-kinded Types in Dotty. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala (SCALA@SPLASH 2016), Amsterdam, Netherlands*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2998392.2998400

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 (Scala 2019), London, United Kingdom*. ACM, New York, NY, USA, 12–16. https://doi.org/10.1145/3337932.3338813

Benjamin Pierce and Martin Steffen. 1997. Higher-order subtyping. *Theoretical Computer Science* 176, 1–2 (1997), 235–282. https://doi.org/10.1016/S0304-3975(96)00096-5

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

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT Press, Cambridge, MA, USA.

Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-Dependent Types. *PACMPL* 3, OOPSLA, Article 145 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360571

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016), Amsterdam, Netherlands*. ACM, New York, NY, USA, 624–641. https://doi.org/10.1145/2983990.2984008

Gabriel Scherer and Didier Rémy. 2015. Full Reduction in the Face of Absurdity. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems (ESOP 2015), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2015), London, UK (LNCS, Vol. 9032)*, Jan Vitek (Ed.). Springer, Berlin, Heidelberg, 685–709. https://doi.org/10.1007/978-3-662-46669-8_28

Christopher A. Stone and Robert Harper. 2000. Deciding Type Equivalence in a Language with Singleton Kinds. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000), Boston, MA, USA*. ACM, New York, NY, USA, 214–227. https://doi.org/10.1145/325694.325724

Sandro Stucki. 2017. *Higher-Order Subtyping with Type Intervals*. Ph.D. Dissertation. School of Computer and Communication Sciences, École polytechnique fédérale de Lausanne, Lausanne, Switzerland. https://doi.org/10.5075/epfl-thesis-8014

Sandro Stucki and Paolo G. Giarrusso. 2021. *A Theory of Higher-order Subtyping with Type Intervals – Agda Formalization*. Zenodo. https://doi.org/10.5281/zenodo.4775731

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. 2004. A Concurrent Logical Framework: The Propositional Fragment. In *International Workshop on Types for Proofs and Programs (TYPES 2003), Torino, Italy, April 30–May 4, 2003, Revised Selected Papers (LNCS, Vol. 3085)*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer, Berlin, Heidelberg, 355–377. https://doi.org/10.1007/978-3-540-24849-1_23

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying Typing and Subtyping. *PACMPL* 1, OOPSLA, Article 47 (Oct. 2017), 26 pages. https://doi.org/10.1145/3133871

Jan Zwanenburg. 1999. Pure Type Systems with Subtyping. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999), L'Aquila, Italy*, Jean-Yves Girard (Ed.). LNCS, Vol. 1581. Springer, Berlin, Heidelberg, 381–396. https://doi.org/10.1007/3-540-48959-2_27