View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Higher-Order Subtyping with Type Intervals

THÈSE N° 8014 (2017)

PRÉSENTÉE LE 10 NOVEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE MÉTHODES DE PROGRAMMATION 1
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDERALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Sandro STUCKI

acceptée sur proposition du jury:

Prof. A. Lenstra, président du jury
Prof. M. Odersky, directeur de thèse
Prof. A. Abel, rapporteur
Prof. F. Pottier, rapporteur
Prof. V. Kunčak, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

Acknowledgments

During my time as a PhD student at EPFL, I have had the great fortune to meet, collaborate with, and learn from many wonderful people. Without them – my colleagues, mentors, friends, and family – this work would not have been possible, and I am grateful for their support.

I am grateful to Martin, my advisor, for introducing me to functional programming as an undergraduate many years ago, for taking me on as a PhD student at LAMP, and for giving me the freedom to pursue my own research interests during my PhD studies.

To Andreas Abel, François Pottier, and Viktor Kunčak, for accepting to serve on my thesis jury, and to Arjen Lenstra for chairing it. I greatly appreciate the time and effort they invested in reading the initial draft of this dissertation, giving me insightful and constructive feedback, and traveling to Lausanne to participate in my defense.

To my friends and colleagues Heather and Vojin, who welcomed me during my first visit to LAMP and immediately took me under their wings, always ready to help and to lend an ear, both in times of exhilaration and desperation.

To my friend and lab mate Manohar, for all the fun we've had over the years at and off work, writing papers, attending and organizing conferences, taking road trips, missing a plane and deciding to visit Niagara Falls instead, for countless discussions about life, its meaning and the universe, poetry and jam sessions at Sat that lead to some hilarious performances, and for doing all of it with a smile.

To Denys and Amir, for TA-ing the Foundations of Software class with me, and for their unquestioning support and kind words when I needed to take time off to be with my family.

To Tiark and Nada, for mentoring me during the first part of my PhD, helping me write my first paper, and introducing me to generative programming, domain-specific languages, and dependent object types.

To Sébastien, for his help in translating the abstract of this dissertation, where his eye for detail was much appreciated, but also for his heroic attempts to teach some of us LAMPions to sing, and in general, for his positive attitude to life and his infectious laugh.

To my office mates throughout the years, Lukas, Hubert, Dima, Vladimir, Vera, Guillaume, and Olivier, for many friendly "hello"s and "good night"s to start and end the work day with a smile, and for letting me monopolize our much too small whiteboard for weeks at a time. Special thanks go to Hubert, for many discussions about the PhD, life, and the universe, usually over beers, and to Guillaume for many discussions about type inference rules, Dotty bugs, and the universe, usually over burgers. It's fair to say that this dissertation could not have been completed without Guillaume's uncanny ability to produce counter examples to my theories

and to answer any question about the Dotty type checker one could possibly come up with.
To Aggelos, Mano, and Vlad for working evenings and weekends to organize last year's Scala Symposium with me, and to Heather and Philipp for mentoring us – your advice was much needed and much appreciated.
To all the other LAMPions, past and present, Alex, Danielle, Eugene, Fabien, Felix, Fengyun, Guillaume Massé, Ingo, Jorge, Julien, Natascha, Nico, Olaf, Ondřej, and Tobias for five years of collaborations, lunches, and coffee breaks full of philosophical discussions, beers at Satellite and The Great Escape, ski weekends in the nearby alps, and for simply being a great bunch.
To the organizers of the 2014 Oregon Programming Languages Summer School (OPLSS), for introducing me to dependent types, proof assistants, and a group of inspiring people in the field of programming languages, teachers and students alike.
To Paolo, for sharing his wealth of knowledge on just about any topic in programming languages, and for many discussions about some of the more subtle points of this work.
To my students Arjen, Cédric, and Fengyun who taught me more than I could possibly have taught them.
To Jean, Jérôme, and Vincent – the "Kappa gang" at Paris 7 and ENS – for showing me that programming languages theory is about more than just programming, and for introducing me to the fascinating subjects of systems and synthetic biology. Special thanks go to Vincent, who hosted me during a six-months research internship in Edinburgh, for mentoring me, encouraging me to tackle problems I had no idea how to solve, giving me the tools to solve them anyway, and for teaching me to have fun in the process.
To Vincent's students and collaborators, Ricardo, Sebastian, Philipp, and Tobias, for the great times we've had in Edinburgh, exploring master equations, category theory, the dynamics of branching polymers and most of the cask ales served in Edinburgh's pubs. I met Ricardo in Vincent's office on my first day at Edinburgh University, and we immediately became brothers in crime. Equally bewildered and enthralled by the strange diagrams on Vincent's whiteboard, we decided to figure them out together and each gained a friend in the process. I thank Vincent, Ricardo, and Tobias for the six months that followed and for the many collaborations, meetings, and adventures we shared, and continue to share, long after my initial stay in Edinburgh.
To Anu, Jan Eric, Olle, Ponna, Stefan, and the rest of the team at Mitrionics, where I did my MSc thesis and spent two great years as an engineer, for mentoring me and for recognizing and awaking the budding scientist in me.
To Helen Fielding, for providing an open ear and words of wisdom during a particularly difficult time, and for reminding me of the importance of setting the right priorities in life.
To my brother André and my parents Esthi and Ruedi, to whom I owe everything and whose unconditional love and support have carried me through many ups and downs. I could not have wished for anything more.
And I am especially grateful to my wife Lisa, for her incredible patience and her unwavering support during the past five years, for challenging me when I am being complacent, for encouraging me when I am in despair, but most of all, for believing in me. You are the best.

Lausanne, October 15, 2017 Sandro Stucki

Abstract

Modern, statically typed programming languages provide various abstraction facilities at both the term- and type-level. Common abstraction mechanisms for types include parametric polymorphism – a hallmark of functional languages – and subtyping – which is pervasive in object-oriented languages. Additionally, both kinds of languages may allow parametrized (or generic) datatype definitions in modules or classes. When several of these features are present in the same language, new and more expressive combinations arise, such as (1) bounded quantification, (2) bounded operator abstractions and (3) translucent type definitions. An example of such a language is Scala, which features all three of the aforementioned type-level constructs. This increases the expressivity of the language, but also the complexity of its type system.

From a theoretical point of view, the various abstraction mechanisms have been studied through different extensions of Girard's higher-order polymorphic λ-calculus F_ω. Higher-order subtyping and bounded polymorphism (1 and 2) have been formalized in $F^\omega_{<:}$ and its many variants; type definitions of various degrees of opacity (3) have been formalized through extensions of F_ω with singleton types.

In this dissertation, I propose *type intervals* as a unifying concept for expressing (1–3) and other related constructs. In particular, I develop an extension of F_ω *with interval kinds* as a formal theory of higher-order subtyping with type intervals, and show how the familiar concepts of higher-order bounded quantification, bounded operator abstraction and singleton kinds can all be encoded in a semantics-preserving way using interval kinds. Going beyond the status quo, the theory is expressive enough to also cover less familiar constructs, such as lower-bounded operator abstractions and first-class, higher-order inequality constraints.

I establish basic metatheoretic properties of the theory: I prove that subject reduction holds for well-kinded types w.r.t. full β-reduction, that types and kinds are weakly normalizing, and that the theory is type safe w.r.t. its call-by-value operational reduction semantics. Key to this metatheoretic development is the use of hereditary substitution and the definition of an equivalent, canonical presentation of subtyping, which involves only normal types and kinds. The resulting metatheory is entirely syntactic, i.e. does not involve any model constructions, and has been fully mechanized in Agda.

The extension of F_ω with interval kinds constitutes a stepping stone to the development of a higher-order version of the calculus of Dependent Object Types (DOT) – the theoretical foundation of Scala's type system. In the last part of this dissertation, I briefly sketch a possible extension of the theory toward this goal and discuss some of the challenges involved in

adapting the existing metatheory to that extension.

Keywords: type systems, higher-order subtyping, type intervals, bounded polymorphism, bounded type operators, singleton kinds, dependent kinds, dependent object types, metatheory, type safety, hereditary substitution.

Résumé

Les langages de programmation statiquement typés modernes fournissent des utilitaires d'abstraction tant au niveau des termes que des types. Parmi les mécanismes d'abstraction communs, l'on trouve le polymorphisme paramétrique – un étendard des langages fonctionnels – et le sous-typage – omniprésent dans les langages orientés objet. De plus, ces deux types de langages peuvent permettre de définir des types de données paramétrés (aussi dits génériques) au sein de modules ou de classes. Lorsque plusieurs de ces fonctionnalités sont présentes dans un même langage, de nouvelles combinaisons, plus expressives, surgissent, parmi lesquelles (1) la quantification bornée, (2) les abstractions d'opérateur bornées et (3) les définitions de types translucides. Un exemple d'un tel langage est Scala, lequel comprend les trois constructions au niveau des types susmentionnées. Ces combinaisons augmentent l'expressivité du langage, mais rendent également son système de types plus complexe.

D'un point de vue théorique, les différents mécanismes d'abstraction ont été étudiés au travers de diverses extensions de F_ω, le λ-calcul avec polymorphisme d'ordre supérieur de Girard. Le sous-typage d'ordre supérieur et le polymorphisme borné (1 et 2) ont été formalisés comme $F^\omega_{<:}$ et ses nombreuses variantes ; les définitions de type aux divers degrés de transparence (3) l'ont été au moyen d'extensions de F_ω avec des types singleton.

Dans cette thèse, je propose les *intervalles de type* comme un concept unifié pour exprimer (1–3) ainsi que d'autres constructions qui y sont reliées. En particulier, je développe une extension de F_ω *avec sortes intervalle*, et montre comment les concepts familiers que sont la quantification bornée d'ordre supérieur, l'abstraction d'opérateur bornée et les sortes singleton peuvent tous être encodés avec des sortes intervalle, en préservant la sémantique.

J'établis les propriétés métathéoriques élémentaires de la théorie : je prouve que la réduction de sujet est vérifiée pour les types aux sortes bien formées par rapport à la β-réduction complète, que les types et sortes supportent la normalisation faible, et que la théorie présente un typage sûr par rapport à sa sémantique de réduction opérationnelle par valeur. Deux points clef de ce développement métathéorique sont l'usage de substitution héréditaire et la définition d'une présentation canonique du sous-typage équivalente qui inclut uniquement des types et sortes normalisés. La métathéorie qui en résulte est entièrement syntaxique – c'est-à-dire qu'elle ne comprend aucune construction modèle – et a été complètement mécanisée en Agda.

L'extension de F_ω avec des sortes intervalle constitue une étape majeure pour développer une version du calcul à Types Objet Dépendants (DOT, Dependent Object Types), la fondation théorique du système de types de Scala. Dans la dernière partie de cette thèse, j'entrouvre

une extension possible de la théorie en vue de cet objectif, et traite de quelques défis qui se présentent lorsque l'on tente d'adapter la métathéorie existante à cette extension.

Mots clefs : systèmes de types, sous-typage d'ordre supérieur, intervalles de type, polymorphisme borné, opérateurs de type bornés, sortes singleton, sortes dépendantes, types objet dépendants, métathéorie, typage sûr, substitution héréditaire.

Contents

Acknowledgments **i**

Abstract (English/Français) **iii**

List of figures **ix**

1 Introduction **1**
1.1 Contributions and overview 4

2 Background **7**
2.1 Upper bounds 9
2.2 Intervals and singletons 11
2.3 First-class Inequalities 12
2.4 Related work 14

3 The declarative system **21**
3.1 Syntax 21
3.1.1 Encodings 24
3.1.2 Structural operational semantics 25
3.2 Declarative typing and kinding 26
3.2.1 Context and kind formation 27
3.2.2 Kinding and typing 30
3.2.3 Subkinding and subtyping 31
3.2.4 Kind and type equality 35
3.3 Basic metatheoretic properties 37
3.3.1 Substitution lemmas 37
3.3.2 Admissible order-theoretic rules 39
3.4 Validity 40
3.4.1 The extended system 42
3.4.2 Equivalence 48
3.5 Congruence lemmas for type and kind equality 48
3.6 Admissible rules for higher-order extrema and intervals 49
3.7 Subject reduction for well-kinded types 55
3.8 Type safety 56

4 Normalization of types **61**
4.1 Normalization of raw types and kinds 61
4.1.1 Syntax 62
4.1.2 Weak equality 63
4.1.3 Hereditary substitution in raw types 64
4.1.4 Normalization of raw types 69
4.1.5 Soundness of normalization 71
4.2 Simple kinding of normal types 72
4.2.1 Simply-kinded hereditary substitution 75
4.2.2 Simplification and normalization of kinding 77

5 The canonical system **85**
5.1 Soundness and basic properties 91
5.1.1 Order-theoretic properties 93
5.1.2 Canonical replacements for declarative rules 95
5.1.3 Simplification of canonical kinding 97
5.2 The hereditary substitution lemma 98
5.2.1 Validity 101
5.2.2 Lifting of weak equality to canonical equality 102
5.3 Completeness of canonical kinding 103
5.4 Inversion of canonical subtyping 106
5.5 Type safety revisited 108

6 Extending the theory **111**
6.1 Type members in Scala and DOT 111
6.2 Higher-order type members 113
6.2.1 Avoiding additional reductions in types 114
6.2.2 Permitting additional reductions in types 115
6.2.3 Non-termination 116

7 Conclusion **119**

Bibliography **121**

Curriculum vitae **127**

List of Figures

3.1	Syntax of $F^{\omega}_{\because}$	22
3.2	Syntactic shorthands and encodings	24
3.3	Call-by-value reduction	25
3.4	Full β-reduction in types and kinds	26
3.5	Declarative presentation of $F^{\omega}_{\because}$ – part 1	28
3.6	Declarative presentation of $F^{\omega}_{\because}$ – part 2	29
3.7	Extended declarative kinding and subkinding	43
4.1	Alternative syntax for types	62
4.2	Weak type equality	63
4.3	Hereditary substitution	65
4.4	Recursive structure of hereditary substitution	66
4.5	Normalization of types	69
4.6	Simplified kinding	74
5.1	Canonical presentation of $F^{\omega}_{\because}$ – part 1	86
5.2	Canonical presentation of $F^{\omega}_{\because}$ – part 2	87
5.3	Top-level transitivity-free canonical subtyping	106

1 Introduction

Modern, statically typed programming languages provide various abstraction facilities, both at the term level and at the type level. Common abstraction mechanisms for types include parametric polymorphism – a hallmark of languages following the functional paradigm, such as ML, Haskell or Agda – and subtyping – which is pervasive in object-oriented languages, such as C++, Java or Scala. Additionally, both kinds of languages may allow parametrized (or generic) datatype definitions in modules or classes.

When several of these features are present in the same language, new and more expressive combinations arise. Such combinations include

1. bounded quantification,
2. bounded type operators, and
3. translucent type definitions.

The presence of these abstraction mechanisms increases the expressivity of the language, but also the complexity of its type system.

Scala is a multi-paradigm language that integrates functional and object-oriented concepts and features all of the aforementioned type-level constructs. This is illustrated by the following code snippet.

```
object boundedUniversal {
  trait Bounded[B, F[_ <: B]] { def apply[X <: B]: F[X] }
  type All[F[_]] = Bounded[Any, F]
}
```

The snippet defines a module `boundedUniversal` with two parametrized type members: a trait `Bounded` and a type alias `All`. Scala's traits are similar to Java's interfaces or ML's module signatures. They represent (partly) abstract datatypes or modules that can be refined through subtyping, combined through mixin composition, and instantiated through object creation.

The trait Bounded is parametrized by a proper type B and a type operator F. The abstract operator F is declared to take a single (unnamed) type parameter, which in turn is upper-bounded by B, i.e. constrained to be a subtype of B. The parameter F is thus an example of an abstract bounded operator (2).

The interface of the trait Bounded consists of a single polymorphic method apply, which takes a single type parameter X, again upper-bounded by B, and returns an instance of F[X], i.e. the type resulting from applying the abstract operator F to the abstract type X. This application is permitted because of the bound X <: B, which ensures that X is indeed a subtype of B. The method apply is an example of bounded quantification (1). Indeed the trait Bounded and the type alias All are just Scala encodings of the bounded and unbounded universal quantifiers found in $F_{\leq}$ [25, 14].

The definitions of the trait Bounded and the type alias All are examples of (3). The definition of Bounded is partly abstract (in the object-oriented sense of the word): although some of its interface is exposed, concrete instances of Bounded may refine that interface while keeping the details of such refinements hidden. The type alias All, on the other hand, is completely transparent: the compiler identifies any occurrences of the type All[F] with those of its definition Bounded[Any, F] – hence the name type alias. The following method definition illustrates this.

```
import boundedUniversal._
def boundedListAsAll(x: Bounded[Any, List]): All[List] = x
```

There is a spectrum of more or less abstract type definitions ranging from completely transparent type aliases to completely abstract (or opaque) type members. We will have more to say about this spectrum in the next chapter.

Although the concepts of bounded quantification and bounded type operators (1 and 2) may seem quite different from abstract definitions of various degrees (3), all three are in fact closely related. Returning to our example, the types Bounded and All can themselves be seen as bounded abstract operators. Indeed, the above definitions could be rewritten as follows:[1]

```
object boundedUniversal {
  type Bounded[B, F[_ <: B]] <: { def apply[X <: B]: F[X] }
  type All[F[_]] >: Bounded[Any, F] <: Bounded[Any, F]
}
```

This version is closer to how type definitions are represented in the calculus of Dependent Object Types (DOT), Scala's core calculus [4].

The trait Bounded is now represented as an abstract type declaration with only an upper

[1] As of Scala version 2.12.3 this code is valid and accepted by the compiler. Somewhat ironically, the next generation Scala compiler Dotty [40], which uses DOT as a core calculus, no longer accepts this code.

bound. This means that its interface remains exposed – any instance of `Bounded[B, F]` is also an instance of the structural record `{ def apply[X <: B]: F[X] }` containing the method `apply`. Conversely, not every record (or class) with a method `apply` is automatically an instance of the type `Bounded[B, F]`, only those that explicitly extend or mix in an instance of `Bounded`. This ensures that `Bounded` continues to behave like a nominal type, as Scala traits are supposed to.[2]

Unlike `Bounded`, the abstract type `All` is both upper- and lower-bounded. In general, a type declaration of the form `X >: A <: B`, irrespective of whether `X` is a type member or a type parameter, introduces an abstract type `X` that is bounded by `A` from below and by `B` from above. The bounds constrain the possible concrete definitions of `X` to be supertypes of `A` and subtypes of `B`. In other words, the declaration `X >: A <: B` specifies a *type interval* in which `X` must be contained. Type aliases such as `All` take the form of *singleton intervals* `X >: A <: A`, where the lower and upper bounds coincide. Because subtyping in Scala is antisymmetric, this effectively identifies the abstract type `X` with its singleton bound `A`.

Scala's type system also features a pair of extremal types `Any` and `Nothing`. The type `Any` is maximal, i.e. it is a supertype of every other type, while the type `Nothing` is minimal, i.e. a subtype of every other type. Thanks to the extremal types, abstract types `X` with only an upper or lower bound `A` can be thought of as inhabiting the degenerate intervals `X >: Nothing <: A` and `X >: A <: Any`, respectively.

This suggests a uniform treatment of 1–3 through type intervals, and indeed, this is essentially how bounded quantification (1) and type definitions (3) are modeled in DOT. Unfortunately, DOT lacks intrinsics for higher-order computation, such as type operator abstractions and applications, so that our theoretical understanding of the corresponding Scala features, including bounded operators (2), remains limited.

Traditionally, 1–3 have been studied through typed λ-calculi, and in particular through somewhat orthogonal extensions of Girard's higher-order polymorphic λ-calculus F_ω [27]. Higher-order subtyping and bounded polymorphism (1 and 2) have been formalized in $F^\omega_{<:}$ and its many variants; type definitions of various degrees of opacity (3) have been formalized through extensions of F_ω with singleton types.

The treatment of 1 and 3 in DOT and the above code examples suggest a different, unified approach to studying 1–3: the development of a formal theory of higher-order subtyping with type intervals. By studying type intervals in their own right, we may even hope to find a theory that is strictly more general that previous treatments of 1–3, revealing novel or at least unfamiliar type-level abstraction mechanisms in the process.

[2] For details on how Scala traits and their refinements may be encoded in DOT, we refer the interested reader to Chapter 2 of Amin's dissertation [3].

1.1 Contributions and overview

In summary, we propose the following thesis:

> Higher-order bounded quantification, bounded operator abstraction and singleton kinds are specific instances of the more general concept of type intervals; a formal theory of higher-order subtyping with type intervals thus subsumes and generalizes existing theories of these concepts.

To demonstrate this thesis, we proceed as follows.

Background and related work

In Chapter 2, we review the concepts of bounded quantification, bounded type operators and type definitions through various examples. Along the way, we illustrate the use of type intervals by elaborating the examples into a prospective extension of F_ω with *interval kinds*. We also sketch some less familiar but powerful idioms involving type intervals, such as first-class higher-order type inequalities, and discuss the problems that arise from abstractions over intervals with inconsistent bounds.

We survey previous work on higher-order subtyping, bounded polymorphism and translucent type definitions in the second part of the chapter, and discuss how and where our work fits into the overall program of developing theoretical foundations for Scala's type system.

The declarative system

We introduce $F^{\omega}_{..}$ – our formal theory of higher-order subtyping with type intervals – in Chapter 3. $F^{\omega}_{..}$ is a typed lambda calculus extending F_ω with higher-order subtyping and interval kinds.

Throughout Chapter 3, we present the syntax and structural operational semantics of $F^{\omega}_{..}$, as well as its declarative typing, kinding, subtyping, subkinding and equality rules. We prove some basic metatheoretic properties of the system – just enough to show that the subject reduction property (aka preservation) holds for well-kinded types. In the last part of the chapter, we discuss type safety of $F^{\omega}_{..}$ and the challenges involved in proving it. We conclude Chapter 3 with an outline of our strategy for proving type safety in the next two chapters.

Normalization of types

In Chapter 4, we show that types and kinds in $F^{\omega}_{..}$ are weakly normalizing. To this end, we define a bottom-up normalization procedure based on *hereditary substitution* that computes the $\beta\eta$-normal forms of types and kinds, and prove its soundness.

The chapter is divided into two parts. The first half of the chapter defines the hereditary substitution and normalization functions on raw, i.e. unkinded, types and establishes basic properties. In the second half, we introduce a set of *simplified kinding* judgments. Simplified kinding serves a dual purpose: firstly, it provides a syntactic characterization of normal types, and secondly, it allows us to establish important properties about hereditary substitutions and normal forms that are key to the technical development presented in the next chapter.

The canonical system

In Chapter 5, we introduce a *canonical* system of judgments covering the kind and type level of $F^{\omega}_{::}$. In contrast to those of the declarative system, the judgments of the canonical system are defined directly on $\beta\eta$-normal forms, and restrict the use of subkinding.

We first introduce and discuss the inference rules for the various canonical judgments and compare them against their declarative counterparts. Then, we develop the necessary metatheory to establish equivalence of the canonical and declarative presentations: we prove soundness of the canonical system, preservation of the canonical judgments under hereditary substitutions, inversion of canonical subtyping, and finally completeness of the canonical system. We conclude the chapter by completing the type safety proof laid out in Chapter 3.

The metatheoretic development presented in Chapters 3 to 5 has been fully mechanized using the Agda proof assistant [37].[3]

Extending the theory

The extension of F_{ω} with interval kinds is just one step in a larger effort to formalize Scala's type system. Although it provides a theoretical foundation for several of Scala's type-level abstraction mechanisms, $F^{\omega}_{::}$ lacks other important type system constructs found in Scala, such as *abstract type members* and *path-dependent types*. Conversely, the calculus of Dependent Object Types (DOT) – the core calculus of the new Dotty compiler for Scala [40] – features abstract type members and path-dependent types but no higher-order type operators. Ideally, we would like to combine the two calculi to obtain a full theory of higher-order dependent object types. In Chapter 6, we briefly sketch a possible extension of $F^{\omega}_{::}$ toward this goal and discuss some of the challenges involved in adapting the existing metatheory to that extension.

[3] The Agda source code is freely available at `https://github.com/sstucki/f-omega-int-agda`

2 Background

Before we go on to formally define our theory of type intervals $F^{\omega}_{\cdot\cdot}$, let us illustrate the use of the more familiar abstraction mechanisms – bounded quantification, bounded operator abstraction and singleton kinds – as well as their encodings through interval kinds, in a bit more detail. We do so informally, by deconstructing our introductory example, and reconstructing it in various typed λ-calculi. As we go along, we introduce the above-mentioned constructs, one by one, and show how they can be encoded in $F^{\omega}_{\cdot\cdot}$. In the next chapter, we make these encodings more formal.

We start by revisiting our Scala example from the previous chapter.

```
object boundedUniversal {
  trait Bounded[B, F[_ <: B]] { def apply[X <: B]: F[X] }
  type All[F[_]] = Bounded[Any, F]
}
```

Recall the different abstraction mechanisms used in this example:

1. bounded quantification – in the signature of the polymorphic method `apply`, which has a type parameter `X` bounded by `B`;
2. bounded type operators – specifically the type parameter `F` of the trait `Bounded`, which is itself an abstract type operator with a parameter bounded by `B`, and
3. type definitions – namely that of `Bounded`, which is partly abstract, and that of `All`, which is fully transparent.

If we want to translate this example from Scala into a typed λ-calculus, a good starting point is Girard's higher-order polymorphic λ-calculus F_{ω} [27]. Although F_{ω} has no notion of subtyping, we can at least express the essence of the operator `All`. Consider the following, more direct definition of `All`

```
trait All[F[_]] { def apply[X]: F[X] }
```

and its F_ω-counterpart

$$A = \lambda Y{:}* \to *.\, \forall X{:}*.\, Y\, X$$

The named, parametrized trait `All[F[_]]` has been replaced by an anonymous operator abstraction taking an argument Y of kind $* \to *$; the signature `apply[X]: F[X]` of the polymorphic method `apply` has been replaced by the universally quantified type $\forall X{:}*.\, Y\, X$. Throughout this dissertation, we use the letters A, B, C, etc. as metavariables denoting types, and X, Y, Z for denoting object variables.[1] In accordance with this naming convention, we have changed the name of the type parameter `F[_]` of `All` to Y.

The object variable X, bound by the universal quantifier in the body of A's definition, designates a proper type, as indicated by its declared kind $*$. The variable Y bound in the enclosing abstraction, on the other hand, designates a unary operator, as indicated by its declared arrow kind $* \to *$. Hence, the type A itself is a higher-order type operator. We recover a proper type by applying A to a suitable type argument. For example, by applying A to the operator $\lambda X{:}*.\, X \to X$, we obtain the polymorphic identity

$$A\,(\lambda X{:}*.\, X \to X) \;=\; (\lambda Y{:}* \to *.\, \forall X{:}*.\, Y\, X)\,(\lambda X{:}*.\, X \to X) \;=\; \forall X{:}*.\, X \to X.$$

The above definition of A is a meta-definition, i.e. just a convenient shorthand for the longer type expression $\lambda Y{:}* \to *.\, \forall X{:}*.\, Y\, X$. But we can also give an object-level definition of A in F_ω. Assume we are given some term t in which we would like to use A as an abstract type operator. Then we can bind A to a type variable Z occurring freely in t, using term-level type abstraction and instantiation,

$$(\lambda Z{:}(* \to *) \to *.\, t)\,(\lambda Y{:}* \to *.\, \forall X{:}*.\, Y\, X).$$

Such definitions can be a bit hard to parse, so we adopt the following encoding of let-bindings, to enhance readability:

$$\begin{aligned}
&\textbf{let}\; X_1 : K_1 = A_1;\, \ldots;\, X_m : K_m = A_m;\\
&\quad\; x_1 : B_1 = s_1\, ;\, \ldots;\;\; x_n : B_n \;= s_n\\
&\textbf{in}\;\; t\\
&\;= \;(\lambda X_1{:}K_1.\cdots\lambda X_m{:}K_m.\, \lambda x_1{:}B_1.\cdots\lambda x_n{:}B_n.\, t)\, A_1 \cdots\, A_m\, s_1 \cdots\, s_n.
\end{aligned}$$

Using a let-binding, the definition of Z becomes

$$\begin{aligned}
&\textbf{let}\; Z : (* \to *) \to * \;=\; \lambda Y{:}* \to *.\, \forall X{:}*.\, Y\, X\\
&\textbf{in}\;\; t.
\end{aligned}$$

[1] To be precise, we use X, Y, Z to denote both concrete object variables as well as metavariables ranging over object variables.

This definition is *opaque*, i.e. the term t sees the signature of Z, but not its definition. For example, consider

$$
\begin{array}{llll}
\textbf{let}\; Z : (* \to *) \to * & = \lambda Y{:}* \to *.\,\forall X{:}*.\,Y\,X & \textbf{in} & \\
\textbf{let}\; x : \forall X{:}*.\,X \to X & = \lambda X{:}*.\,\lambda z{:}X.\,z & \textbf{in} & \textit{— OK} \\
\textbf{let}\; y : Z\,(\lambda X{:}*.\,X \to X) & = \lambda X{:}*.\,\lambda z{:}X.\,z & \textbf{in} \ldots & \textit{— type error}
\end{array}
$$

The second definition in this example type checks whereas the third one does not. The problem is that the variable Z is fully abstract, i.e. we have $Z \neq \lambda Y{:}* \to *.\,\forall X{:}*.\,Y\,X$ as types, despite the definition on the first line. We will return to this issue shortly, but first, let us extend the example to include bounded quantification.

2.1 Upper bounds

There are several extensions of F_ω with higher-order subtyping and bounded quantification; one of the simplest and most popular is Pierce and Steffen's version of $F^\omega_{<:}$ [42]. In $F^\omega_{<:}$, universal types take the general form $\forall X \leq A{:}K.\,B$ where A and B are types and K is a kind. The type A is called the *upper bound* of the variable X, and must be of kind K. The upper bound A constrains the possible type arguments C, to which instances of $\forall X \leq A{:}K.\,B$ can be applied, to those that are subtypes $C \leq A$ of A. This is useful for defining polymorphic functions where one has partial information about the types being abstracted over.

For example, the following Scala method `withLength` can only be applied to instances `x` of type `Y`, which must be a subtype of `Seq[Int]`, the trait of integer sequences from the Scala collections library.

```
def withLength[Y <: Seq[Int]](x: Y): (Y, Int) = (x, x.length)
```

The method `withLength` makes crucial use of the information that `Y` is a subtype of `Seq`: since `x` is an instance of `Y`, and hence a sequence of integers, it must have a field called `length`, which is part of the sequence interface.

The type `Seq[Int]` has many concrete implementations, such as `List`, `Array`, etc. If we apply the method `withLength` to an instance of `List[Int]`, we get a result of type `(List[Int], Int)`, i.e. a pair consisting of a list of integers and its length.

```
val x = List(1, 2, 3)
withLength(x)        // returns (List(1, 2, 3), 3) of type (List[Int], Int)
```

Crucially, the type of the result is `(List[Int], Int)` rather than `(Seq[Int], Int)` because `withLength` is a polymorphic method, returning pairs `(Y, Int)` for given `Y`.

Returning to our introductory example, we can define abstract types Z_B and Z_A in $F^\omega_{<:}$, which

closely resemble the definitions of `Bounded` and `All`, respectively.

$$\begin{aligned}
&\textbf{let}\ Z_B : * \to (* \to *) \to * = \lambda Y_1{:}*.\, \lambda Y_2{:}* \to *.\, \forall X \leq Y_1{:}*.\, Y_2\, X;\\
&\quad\ Z_A : (* \to *) \to * \qquad = Z_B\, \top\\
&\textbf{in}\ \ \ldots
\end{aligned}$$

Here $\top$ denotes the maximum or *top* type, $F^{\omega}_{<:}$'s version of `Any`.

There are still two important differences between the above definitions and the corresponding Scala definitions of `Bounded` and `All`. Firstly, the $F^{\omega}_{<:}$ definitions are again opaque: we have $Z_B\, \top \neq \lambda Y_2{:}* \to *.\, \forall X \leq \top{:}*.\, Y_2\, X$ in the body of the let-expression. Secondly, the signature of Z_B is less precise than that of `Bounded`: the second parameter `F[_ <: B]` of `Bounded` is a bounded operator, i.e. its argument has an upper bound `B`. The second argument Y_2 of Z_B, meanwhile, has no such bound. Although this does not make the definition of Z_B ill-typed (or rather, ill-kinded), it still represents a loss of information.

In order to faithfully represent `Bounded`, we have to extend $F^{\omega}_{<:}$ with bounded operator abstractions. Such an extension has been developed by Compagnoni and Goguen under the name $\mathscr{F}^{\omega}_{\leq}$ [18]. Operator abstractions in $\mathscr{F}^{\omega}_{\leq}$ are of the form $\lambda X \leq A{:}K.\, B$, where A is the upper bound of X and must be of kind K, much like the upper bound of a universally quantified type $\forall X \leq A{:}K.\, B$. But unlike universals, which are proper types, operator abstractions inhabit arrow kinds; and since abstractions carry bounds in $\mathscr{F}^{\omega}_{\leq}$, so do arrow kinds. Given types A and B of kind J and K, respectively, the kind of an operator abstraction $\lambda X \leq A{:}J.\, B$ is $(X \leq A{:}J) \to K$. This makes arrow kinds type-dependent, which substantially complicates the meta theory of $\mathscr{F}^{\omega}_{\leq}$ when compared to that of $F^{\omega}_{<:}$.

In $\mathscr{F}^{\omega}_{\leq}$, we can give the following definitions to represent `Bounded` and `All`.

$$\begin{aligned}
&\textbf{let}\ Z_B : (Y_1{:}*) \to ((X \leq Y_1{:}*) \to *) \to * = \lambda Y_1{:}*.\, \lambda Y_2{:}(X \leq Y_1{:}*) \to *.\, \forall X \leq Y_1{:}*.\, Y_2\, X;\\
&\quad\ Z_A : (* \to *) \to * \qquad\qquad\qquad = Z_B\, \top\\
&\textbf{in}\ \ \ldots
\end{aligned}$$

Following common convention, we omit the upper bound A of bindings $X \leq A : K$ when $A = \top$ and write $J \to K$ as a shorthand for simple, non-dependent arrow kinds, i.e. arrow kinds $(X{:}J) \to K$ where X does not occur freely in K.

Compared to $F^{\omega}_{<:}$, type variable binders are more uniform in $\mathscr{F}^{\omega}_{\leq}$. There are four sorts of type variable binders in $\mathscr{F}^{\omega}_{\leq}$, all of which introduce bindings of the form $X \leq A : K$.

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

Note that every variable is accompanied by a kind K and a type bound A of kind K. There is an alternative representation of these four constructs where A and K are combined into a single expression called a *power kind* [13, 20].

Just as the powerset $\mathscr{P}(A)$ of a set A is the set of all subsets of A, so the powerkind $P(A)$ of a proper type A is the kind of all subtypes of A. In other words, the statements $A \leq B$ and $A : P(B)$ are equivalent. In a system with power kinds, bindings of the form $X \leq A : *$ can then be expressed as $X : P(A)$, e.g. the following definition of Z_B is equivalent to one given above.

$$\textbf{let } Z_B : (Y_1{:}*) \to (P(Y_1) \to *) \to * = \lambda Y_1{:}*.\, \lambda Y_2{:}P(Y_1) \to *.\, \forall X{:}P(Y_1).\, Y_2\, X$$
$$\textbf{in } \ldots$$

Power kinds are strictly more expressive than the four binders of $\mathscr{F}^{\omega}_{\leq}$ because their use is not restricted to bindings. They can also be used in every other position where a kind is expected. For example, we can obtain a more transparent definition of Z_B by using a power kind in the codomain of its signature.

$$\begin{aligned} &\textbf{let } Z_B : (Y_1{:}*) \to (Y_2{:}P(Y_1) \to *) \to P(\forall X{:}P(Y_1).\, Y_2\, X) \\ &\qquad = \lambda Y_1{:}*.\, \lambda Y_2{:}P(Y_1) \to *.\, \forall X{:}P(Y_1).\, Y_2\, X \\ &\textbf{in } \ldots \end{aligned}$$

The signature of Z_B tells us that, when we apply Z_B to suitable type arguments A and B, the result is a subtype of $\forall X{:}P(A).\, B\, X$, i.e. we have $Z_B\, A\, B \leq \forall X{:}P(A).\, B\, X$ in the body of the let-definition; or equivalently

$$Z_B \leq \lambda Y_1{:}*.\, \lambda Y_2{:}P(Y_1) \to *.\, \forall X{:}P(Y_1).\, Y_2\, X.$$

Power kinds can thus be used to express semi-transparent, upper-bounded type definitions, similar to the definition of the trait `Bounded` in our introductory example.

Power kinds are very expressive, but they are clearly not the end of the story. We still lack ways to express (a) *lower-bounded* definitions, and (b) fully *transparent* definitions, such as that of `All` in our running example.

Enter *interval kinds*.

2.2 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. In other words, kinding statements of the form $C : A\,..\,B$ are equivalent to pairs of subtyping statements $A \leq C$ and $C \leq B$. We will establish this equivalence more formally in the next chapter.

Power kinds are simply degenerate intervals $P(A) = \bot \mathinner{..} A$ where the lower bound is the minimum or *bottom* type $\bot$, our equivalent of Scala's `Nothing` type. Since every proper type is a supertype of $\bot$, intervals of the form $\bot \mathinner{..} A$ are effectively unconstrained from below. Dually, intervals of the form $A \mathinner{..} \top$ are unconstrained from above, and can thus be used to encode lower-bounded definitions. Interval kinds of the form $A \mathinner{..} A$, where the lower and upper bounds coincide, are called *singleton kinds* or simply *singletons*. Given a singleton instance $B : A \mathinner{..} A$, we have both $A \le B$ and $B \le A$, which, assuming an antisymmetric subtyping relation, implies $A = B$. Singleton kinds can thus be used to encode transparent definitions and have been studied for that purpose by Stone and Harper [49]. We adopt their notation and write $S(A) = A \mathinner{..} A$ for the singleton containing just A.

Using interval kinds, we refine our definitions of Z_B and Z_A one last time.

$$
\begin{aligned}
&\textbf{let } Z_B : (Y_1{:}*) \to (Y_2{:}\bot \mathinner{..} Y_1 \to *) \to \bot \mathinner{..} (\forall X{:}\bot \mathinner{..} Y_1 .\, Y_2\, X) \\
&\qquad\quad = \lambda Y_1{:}*.\, \lambda Y_2{:}\bot \mathinner{..} Y_1 \to *.\, \forall X{:}\bot \mathinner{..} Y_1 .\, Y_2\, X; \\
&\quad\;\; Z_A : (Y{:}* \to *) \to (Z_B \top Y) \mathinner{..} (Z_B \top Y) \;=\; Z_B \top \\
&\textbf{in } \;\ldots
\end{aligned}
$$

In the body of the let-definition, we now have $Z_B\, A\, B \le \forall X{:}\bot \mathinner{..} A.\, B\, X$ and $Z_A\, C = Z_B \top C$ for all suitably kinded types A, B and C, as desired.

Interval kinds $A \mathinner{..} B$ are only well-formed if A and B are proper types, i.e. of kind $*$. Similarly, the extremal types $\top$ and $\bot$ are proper types, so kinding statements of the form $A : \bot \mathinner{..} B$ can only be used to encode upper bounds on proper types A. To express all of the binders found in $\mathscr{F}^{\omega}_{\le}$, we need a way to encode bindings of the form $X \le A : K$, for arbitrary kinds K. This is indeed possible thanks to (a) *higher-order interval kinds* $A \mathinner{..}_K B$ where K is an arbitrary kind and A, B are a pair of types of kind K, and (b) *higher-order extrema* $\top_K$ and $\bot_K$ which are supertypes and subtypes, respectively, of arbitrary types $A : K$. As we will see in the next chapter, both higher-order intervals and higher-order extrema can be encoded using other kind- and type-level constructs.

2.3 First-class Inequalities

Instances $C : A \mathinner{..} B$ of an interval kind $A \mathinner{..} B$ represent types bounded by A and B respectively. But they also represent proofs that $A \le C$ and $C \le B$, and – by transitivity of subtyping – that $A \le B$. In other words, the inhabitants of interval kinds $A \mathinner{..} B$ represent first-class *type inequalities* $A \le B$. Similarly, higher-order intervals represent type operator inequalities.

Among other things, this allows us to postulate type operators with associated subtyping rules through type variable bindings. For example, we may postulate intersection types $A \wedge B$ by assuming an abstract binary type operator $X_\wedge$ and the usual typing rules for intersections as abstract type inequalities $Y_{\le L}$, $Y_{\le R}$ and $Y_{\le \wedge}$. For readability, we abbreviate dependent arrow

kinds $(X_1{:}J) \to (X_2{:}J) \to \cdots \to (X_n{:}J) \to K$ with multiple parameters $X_1, X_2, \ldots, X_n : J$ of kind J as $(X_1, X_2, \ldots, X_n{:}J) \to K$, and we use a bar to mark abstract operators, writing e.g. $\bar{\wedge}$ instead of $X_\wedge$.

$$
\begin{aligned}
&\bar{\wedge} : * \to * \to *,\\
&Y_{\leq L} : (X_1, X_2{:}*) \to (\bar{\wedge}\, X_1\, X_2) \mathinner{..} X_1, \qquad Y_{\leq R} : (X_1, X_2{:}*) \to (\bar{\wedge}\, X_1\, X_2) \mathinner{..} X_2,\\
&Y_{\leq \wedge} : (Z, X_1, X_2{:}*) \to Z \mathinner{..} X_1 \to Z \mathinner{..} X_2 \to Z \mathinner{..} (\bar{\wedge}\, X_1\, X_2).
\end{aligned}
$$

The two assumptions $Y_{\leq L}$ and $Y_{\leq R}$ correspond to the left- and right-hand projection rules for intersections, respectively, i.e. they say that $A \wedge B \leq A$ and $A \wedge B \leq B$ for any pair of proper types A, B. The assumption $Y_{\leq \wedge}$ encodes the fact that intersections are greatest lower bounds, i.e. that $A \leq B \wedge C$ for any triple of types A, B and C such that $A \leq B$ and $A \leq C$.

To see this, let A, B and C be proper types and assume that $A \leq B$ and $A \leq C$. Then we also have $A : A \mathinner{..} B$ and $A : A \mathinner{..} C$, and hence $Y_{\leq \wedge}\, A\, B\, C\, A\, A : A \mathinner{..} (\bar{\wedge}\, B\, C)$, from which we conclude $A \leq \bar{\wedge}\, B\, C$.

We can also postulate recursive inequations. For example, the following bindings encode an equi-recursive type constructor $\mu : (* \to *) \to *$ that, when applied to a unary operator A, represents the fixpoint $\mu\, A$ of A.

$$
\begin{aligned}
&\bar{\mu} : (* \to *) \to *,\\
&Y_{\leq L} : (X{:}* \to *) \to (\bar{\mu}\, X) \mathinner{..} (X\, (\bar{\mu}\, X)), \qquad Y_{\leq R} : (X{:}* \to *) \to (X\, (\bar{\mu}\, X)) \mathinner{..} (\bar{\mu}\, X).
\end{aligned}
$$

Together, the assumptions $Y_{\leq L}$ and $Y_{\leq R}$ say that $\mu\, A = A\, (\mu\, A)$, i.e. that $\mu\, A$ is a fixpoint of A.

The above examples are only possible because we do not impose any *consistency constraints* on the bounds of intervals. That is, an interval kind $A \mathinner{..} B$ is well-formed, irrespective of whether $A \leq B$ is actually provable or not. For example, the signature of the abstract intersection operator $\bar{\wedge}$ above does not tell us anything about how the type application $\bar{\wedge}\, A\, B$ is related to its first argument A, nor does the abstract left projection inequality $Y_{\leq L}$ impose any constraints on its parameters X_1 and X_2. It is therefore impossible to say anything about the relationship of the bounds $\bar{\wedge}\, X_1\, X_2$ and X_1 of the codomain of $Y_{\leq L}$, other than that they are both proper types. We can certainly not prove that $\bar{\wedge}\, X_1\, X_2 \leq X_1$ in general. We say that the bounds of an interval $A \mathinner{..} B$ are *consistent* if we can prove that $A \leq B$, or equivalently, if we can exhibit an instance $C : A \mathinner{..} B$; otherwise we say that the bounds are *inconsistent*.

One drawback of allowing type variable bindings with inconsistent bounds is that reduction of terms under such assumptions is not safe in general, because inconsistent bounds may correspond to absurd inequalities.

For example, under the absurd assumption $Z : \top \mathinner{..} \bot$, we have $\top \leq Z \leq \bot$, i.e. the subtyping relation becomes trivial. As a consequence, we can type both non-terminating and stuck

terms.

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

This means that, in general, it is unsafe to reduce terms under absurd assumptions. Note that these examples do not break type safety of $F^{\omega}_{..}$ overall though. The absurd assumption $Z : \top\,..\,\bot$ can never be instantiated, and hence reduction of *closed* terms remains perfectly safe. We discuss this point in more detail at the end of Chapter 3.

2.4 Related work

Higher-order subtyping and bounded polymorphism. Bounded quantification has been studied in detail through numerous calculi and type systems. The conceptual core of these systems is summarized in $F_{\le}$, an extension of System F with subtyping and bounded universal quantification [25, 14]. There are two variants of $F_{\le}$ that are commonly considered: the *Kernel* variant $F_{<:}$ [14], which is based on Cardelli and Wegner's *Kernel Fun* [15], and *Full* $F_{\le}$, which is due to Curien and Ghelli [25]. The difference between the two calculi lies in their subtyping rule for bounded universal quantifiers. In Full $F_{\le}$, subtyping relates the bounds of universally quantified types contravariantly, while $F_{<:}$ requires that bounds of related universals be identical. This makes $F_{\le}$ more expressive than $F_{<:}$ but also renders subtyping in $F_{\le}$ undecidable, while subtyping remains decidable in $F_{<:}$. We refer interested readers to Chapters 26 and 28 of Pierce [43] for a detailed comparison of $F_{\le}$ and $F_{<:}$ and a discussion of the decidability issue. Decidability of subtyping is not the focus of this dissertation, so we adopt the more permissive, $F_{\le}$-variant of the subtyping rule for bounded universal quantifiers in $F^{\omega}_{..}$. This means that subtyping in $F^{\omega}_{..}$ is most likely undecidable. However, the metatheory developed in Chapters 3–5 is largely unaffected by this choice and could easily be adapted to the more restrictive $F_{<:}$-variant of the subtyping rule.

An extension of Girard's higher-order polymorphic λ-calculus F_{ω} [27] with higher-order subtyping and $F_{\le}$-style bounded quantification was first proposed by Cardelli under the name $F^{\omega}_{<:}$ [12]. Basic meta theoretic properties of $F^{\omega}_{<:}$ were established by Pierce and Steffen [42], Compagnoni [19], and Compagnoni and Goguen [17]. Extensions of $F^{\omega}_{<:}$ with intersection types ($F^{\omega}_{\wedge}$) [19, 16] and bounded operator abstractions ($\mathscr{F}^{\omega}_{\le}$) [18] have been developed by Compagnoni and Goguen. An extension of higher-order subtyping and bounded quantification to dependent types was developed by Zwanenburg using the general framework of Pure Type Systems (PTS) [54]. More recently, Abel and Rodriguez developed a variant of $F^{\omega}_{<:}$ where types are identified up to $\beta\eta$-equality (rather than just β-equality) and proved its decidability using the purely syntactic technique of hereditary substitution [2]. Of these extensions, Compagnoni and Goguen's $\mathscr{F}^{\omega}_{\le}$ and Abel and Rodriguez's variant of $F^{\omega}_{<:}$ resemble $F^{\omega}_{..}$ most closely.

Many of the ideas found in $F^{\omega}_{<:}$ go back to early work by Cardelli on *power types* [11]. In analogy

to a powerset, the power type $P(A)$ of a given type A is inhabited by all subtypes of A, and the subtyping relationship $B \leq A$ is just a shorthand for the typing $B : P(A)$ in Cardelli's system. Note that $P(A)$ is itself a type, despite the fact that it is being used to classify other types (here $B : P(A)$). This is permitted because Cardelli's system has the *Type:Type* property, i.e. the kind of types is itself a type. While this property makes the system very expressive, it has some undesirable side-effects, e.g. type expressions in this system are non-normalizing [27, 10]. Cardelli was well aware of this issue, and in a later work with Longo, replaced power types with the better behaved *power kinds* [13]. Power kinds can be directly expressed in $F^{\omega}_{\cdot\cdot}$ as interval kinds that are bounded by $\bot$ from below: $P(A) = \bot \,..\, A$.

Crary proposes an extension of F_{ω} with power kinds as a general calculus for higher-order subtyping [20]. He uses this calculus to give a coercion-based interpretation of higher-order subtyping via a translation into a subtyping-free extension of F_{ω}. The representations of higher-order bounded quantifiers and operators in this system are very similar to those used in $F^{\omega}_{\cdot\cdot}$. In his PhD dissertation, Crary describes an alternative, predicative system (λ^{K}) featuring both power kinds and singleton kinds [22].

In addition to bounded polymorphism and translucent type definitions, Scala's type system also features *polarized* higher-order subtyping. In a system with polarized subtyping, the kinds of operators are annotated with the *polarity* or *variance* of their arguments, which allows for more flexible subtyping of operator applications. For example, the type parameter `X` of the `List[+X]` datatype from the Scala standard library is annotated covariantly, which means that `List[A] <: List[B]` whenever `A <: B`. By contrast, the type parameter of the `Array[X]` datatype has no annotation, which means that `Array[A] <: Array[B]` only if `A` and `B` are equal types. An extension of $F^{\omega}_{<:}$ with polarized higher-order subtyping was first considered by Cardelli in an unpublished note [12] and subsequently formalized by Steffen [47], and Duggan and Compagnoni [26]. More recently, Abel proposed the use of polarized subtyping in conjunction with sized types as a means for termination checking in languages with recursive types [1]. $F^{\omega}_{\cdot\cdot}$ does not support polarity annotations; hence all type operators in $F^{\omega}_{\cdot\cdot}$ behave like Scala's `Array` datatype, i.e. $A\,B_1 \leq A\,B_2$ only if $B_1 = B_2$. We leave the extension of $F^{\omega}_{\cdot\cdot}$ to higher-order polarized subtyping for future work.

Translucency and singleton kinds. The notion of *translucency* was introduced by Harper and Lillibridge in the setting of of ML-style modules with sharing constraints [29]. They proposed *translucent sums*, a generalization of weak sum types (aka existential types), as a uniform way of representing modules containing type definitions with varying degrees of opacity (fully transparent, fully opaque, or translucent). In contrast, previous approaches had been restricted to either fully opaque or fully transparent definitions [33].

Stone and Harper later proposed *singleton kinds* as an alternative mechanism for representing type definitions with sharing constraints [49]. Like translucent sums, singleton kinds are expressive enough to cover transparent, opaque, and translucent definitions. An in-depth

discussion of singleton kinds, along with a formal treatment of their relation to translucent sums, is given by Stone [48]. A machine-verified semantics of SML, including a formalization of ML-modules based on singleton kinds, is described in Lee et al. [32].

A predicative variant of singleton kinds already appears in Crary's dissertation [22]. Stone and Harper later described their version of singleton kinds in an impredicative theory based on F_ω [49]. Even before that, Aspinall introduced a related notion of *singleton types* [8]. Singleton types allow internalizing term-level definitions in much the same way that singleton kinds internalize type definitions. There are subtle but important differences between the two theories that go beyond the level in the type-kind hierarchy at which they apply. For example, Aspinall's system has no explicit equality judgment (on terms). Instead, equality judgments of the form $\Gamma \vdash s = t : A$ are just shorthands for typing judgments $\Gamma \vdash s : \{t\}_A$, where $\{t\}_A$ denotes the singleton type containing just the term t of type A. Singleton types $\{t\}_A$ are parametrized by an index type A that indicates the underlying type of t. By contrast, the theories of Crary, Harper and Stone use explicit equality judgments (on types) and corresponding rules for relating singletons to judgmental equalities. The singleton kind inhabited by a proper type A is denoted by $S(A)$, without an explicit kind index. Singletons over arbitrary – possibly higher-order – types A of kind K, written $S(A : K)$, are not intrinsic but may be encoded. Interval kinds are most closely related, both conceptually and formally, to Stone and Harper's singleton kinds.

Hereditary substitutions. The metatheoretic development presented in Chapters 3–5 of this dissertation is purely syntactic, i.e. it does not rely on a logical-relations-style interpretation of types or any other form of model construction. This is possible thanks to a technique known as *hereditary substitution* [52]. Roughly, hereditary substitution allows for the substitution of one $\beta\eta$-normal form into another while maintaining normality of the result. Intermediate β-redexes resulting from substituting an abstraction for a variable in head position are eliminated recursively in a bottom-up fashion, i.e. through further hereditary substitutions. Hereditary substitution is defined directly on types by structural recursion on the simplified kind of the type being substituted. This guarantees the totality of the hereditary substitution function and gives rise to a purely syntactic normalization procedure for types and kinds. Hereditary substitution is also crucial in our definition of *canonical kinding*, and *subtyping* judgments in Chapter 5 and in establishing the equivalence of these judgments and their declarative counterparts. It is via this equivalence that we prove subtyping inversion and ultimately type safety of $F^\omega_{\cdot\cdot}$.

Hereditary substitution and similar syntactic techniques have been used to prove weak normalization of a variety of systems. A particularly illuminating example is provided by Keller and Altenkirch who use hereditary substitution to implement a (total) normalization function for simply-typed lambda terms in Agda [31]. They go on to prove soundness (every term has a $\beta\eta$-normal form) and completeness ($\beta\eta$-equal terms have identical normal forms) of their normalizer, using Agda both as a programming language and as a theorem prover.

Abel and Rodriquez use hereditary substitution to show that subtyping is decidable in a variant of $F^{\omega}_{<:}$ extended with η-conversion of types [2]. Their treatment of normal types inspired our definitions of type normalization and simplified kinding in Chapter 4. Another source of inspiration was Harper and Licata's presentation of *Canonical LF* [28]. In Canonical LF, typing and kinding are defined directly on $\beta\eta$-normal forms. Because LF is a dependently typed language, the kinding rules normally involve substitutions. Since substitutions do not preserve normal forms, Canonical LF uses hereditary substitution for such typing rules. Similarly, we use hereditary substitution in our canonical kinding and subkinding rules in Chapter 5.

Because of its syntactic nature, hereditary substitution is well-suited for mechanizing metatheoretic properties in proof assistants with limited proof-theoretic strength such as Twelf [41]. For example, hereditary substitution has been used in the formalization and mechanization of the equational theory of singleton kinds [21] and the semantics of the SML language [32] in Twelf.

Foundations of Scala. One of the goals of $F^{\omega}_{..}$ is to act as an intermediate step in the development of a higher-order version of the calculus of *Dependent Object Types (DOT)*. The DOT calculus [4] is both a theoretical foundation for Scala's type system as well as the core calculus underlying the experimental *Dotty* compiler for Scala [40]. Yet, while the Scala type system and the Dotty compiler support type-level computation and higher-order subtyping through type definitions, DOT lacks higher-order type-level constructs to faithfully model such features.

Over the years, many formal calculi have been developed to model Scala's type system. These can be grouped roughly into two categories: earlier systems model the Scala type system faithfully and in some detail, including features such as inheritance, mixin composition, nominal class hierarchies and inner classes; more recent systems belonging to the DOT family of calculi abstract over many of the details and focus on essential features such as abstract type members and path-dependent types.

Among the early systems to model Scala, the νObj calculus is the most comprehensive [38]. It models objects and classes with abstract and concrete type members, mixin composition, and admits an encoding of full $F_{\leq}$ as well as generic classes. Unlike for most of the other early systems, basic metatheoretic properties of νObj have been established: e.g. both type safety and undecidability of type checking have been proven for νObj. Featherweight Scala [23], was introduced as an algorithmic alternative to νObj in order to study type checking in Scala. Although it is not as rich as the νObj calculus, it still covers many of Scala's type system features, including nested classes, mixins and abstract type members. Type checking was proven decidable for Featherweight Scala, while a proof of type safety was left as future work.

Odersky et al. discussed encodings of some type operators in νObj [38], but these were later found to be insufficient for expressing more complex, higher-order type operators by Moors et al. [35]. To address these shortcoming of νObj, they proposed a new calculus, *Scalina*, as a foundation for higher-kinded types in Scala, which, at that point, did not yet support fully

general higher-order type operators. The type and kind system of Scalina is rather expressive. It includes interval kinds and supports a form of *polarized* higher-order subtyping [47] through the somewhat non-standard notions of *un-types* and *un-members*, which do not have a counterpart in Scala. Type and kind safety was never established for Scalina but it inspired an extension of Scala's type system with higher-order subtyping, including higher-order bounded polymorphism, type operators and type definitions [34]. Recent work on the Dotty compiler prompted a radical redesign of Scala's type checker, and in particular the way in which type parameters in class and type definitions are handled. This sparked renewed interest in the foundations of higher-kinded types in Scala [39].

The core calculus underlying the Dotty compiler is DOT [4]. Several variants of DOT exist, differing both in their expressiveness and in the presentation of the operational semantics (small vs. big-step, environment vs. substitution-based). Compared to earlier systems, the members of the DOT family are much more parsimonious, both in the number of type-system constructs that are intrinsically supported, as well as the number of judgments involved in their static semantics. This was crucial for the development of machine-verified type safety proofs of the various DOT calculi. An initial draft of DOT was given by Amin et al. [5] and a first type safety proof for a restricted subset called μDOT was established by Amin and Rompf [7]. The first proof of full DOT, using a big-step semantics and including a full subtyping lattice and recursive records with self-types was described and mechanized by Rompf and Amin [44]. A simpler safety proof, based on a simplified variant of DOT using a small-step semantics and a more restricted subtyping relation was presented by Amin et al. shortly thereafter [4]. Additional machine-verified proofs for different variations of DOT and other metatheoretic properties have since been developed [45, 3, 6, 51].

Central to all variants of DOT is the notion of *abstract type members*, which corresponds closely to the eponymous construct in Scala. The type, or declaration $\{ L : A \,..\, B \}$ of an abstract type member consists of a type label L and a pair of types A and B that bound L from below and above, respectively. In other words, the declaration associates L with a type interval $A \,..\, B$. In $F^{\omega}_{\cdot\cdot}$, we separate the concept of type intervals from that of abstract type members and study type intervals in their own right through the notion of interval kinds. This allows arbitrary types – not just type members – to inhabit intervals.

While DOT admits encodings of some type operators, none of the DOT calculi developed so far are expressive enough to cover the general (higher-order) type computations supported by Scala's type system. Although we do not develop a higher-order theory of dependent object types in this dissertation, we see $F^{\omega}_{\cdot\cdot}$ as a first step toward the development of such a theory. For example, the notion of interval kinds leads to a natural generalization of type member declarations to the higher-order setting: a higher-order type member declaration $\{ L : K \}$ associates the type label L with an arbitrary kind K, which may or may not be a (higher-order) type interval.

Abstraction over inconsistent assumptions. The problem that computations can become unsafe under absurd assumptions is well known. It is present in theories that permit abstractions over *coercions* that can be applied *implicitly*, i.e. that need not appear in the expressions being coerced. Safety issues arise when abstractions are allowed over possibly absurd coercions, i.e. coercions that can never be instantiated, but expressions making implicit use of such coercions are nevertheless allowed to be reduced. We say that an abstraction is *consistent* if we could instantiate (i.e. prove) its assumption at the point of abstraction, and *inconsistent* otherwise. Note that whether or not an abstraction is consistent may depend on earlier assumptions (and their consistency).

An early example of this problem appears in the *extensional* version of Martin-Löf's Type Theory (MLTT). In his 1990 book *Programming in Martin-Löf's Type Theory* [36], Nordström shows how the *extensional equality type* $\mathsf{Eq}(A, s, t)$, which is inhabited by proofs that s and t are equal terms of the type A, can be used to construct a non-terminating term, assuming a proof of an absurd equation, similar to the example we described earlier in this section.[2] This is despite the fact that MLTT is normalizing, i.e. that every closed, well-typed term has a normal form in MLTT. The key feature of extensional equality that allows for this to happen is that explicit assumptions like $\Gamma \vdash x : \mathsf{Eq}(A, s, t)$, can be reflected into *equality judgments* $\Gamma \vdash s = t : A$ by the elimination rule for Eq. Such judgments can then be used to identify s and t implicitly, i.e. without x appearing explicitly in any term. Inhabitants of Eq can thus act as implicit term coercions. Similarly, explicit assumptions like $\Gamma \vdash X : A \mathrel{..}_K B$, can be turned into *subtyping*, or *inequality judgments* $\Gamma \vdash A \leq B : K$, and thus be used as implicit type coercions.

The effect of inconsistent assumptions on the safety of System F-like calculi with subtyping has been studied in depth by Cretin, Rémy, and Scherer [24, 46]. Cretin and Rémy propose a variant of System F with coercion constraints, called F_{cc} [24], which distinguishes between *coherent* coercions that can be erased (i.e. be used implicitly) and *incoherent* coercions that must be introduced and eliminated explicitly through corresponding syntactic forms. Programs involving only coherent coercions can be evaluated safely using full β-reduction, while reduction under incoherent abstractions is not allowed. The resulting calculus is expressive enough to cover different variants of subtyping, $F_{\leq}$-style bounded quantification, instance-bounded quantification, and GADTs. Scherer and Rémy improve on this scheme, by allowing even finer-grained control over inconsistent abstractions through the notion of *assumption hiding*, which temporarily hides an implicit coercion in a term and thus unblocks reduction [46]. Following the approach used in DOT [4, 45], we do not track coherence of coercions in $F^{\omega}_{..}$, nor do we syntactically distinguish between consistent and inconsistent abstraction. While this simplifies the calculus and its metatheory, it also means that all abstractions must be considered inconsistent and reduction is unsafe in open terms.

[2] For details, see Chapter 14 of Nordström's book [36].

3 The declarative system

In this chapter, we introduce $F^{\omega}_{\cdot\cdot}$ – our formal theory of higher-order subtyping with type intervals. $F^{\omega}_{\cdot\cdot}$ is a typed lambda calculus extending Girard's F_{ω} [27] with subtyping. Like Compagnoni and Goguen's $\mathscr{F}^{\omega}_{\leq}$ [18], it features bounded polymorphism as well as bounded operator abstraction, but unlike $\mathscr{F}^{\omega}_{\leq}$, it expresses abstract type bounds through *interval kinds*. Throughout this chapter, we present the syntax and structural operational semantics of $F^{\omega}_{\cdot\cdot}$, as well as its declarative typing, kinding, subtyping, subkinding and equality rules. We prove some basic metatheoretic properties of the system – just enough to show that the subject reduction property (aka preservation) holds for well-kinded types. To conclude the chapter, we discuss the challenges involved in proving type safety and outline a strategy to do so. The next two chapters are dedicated to putting this strategy into practice.

3.1 Syntax

The syntax of $F^{\omega}_{\cdot\cdot}$ is given in Fig. 3.1. 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* $\top$ denotes the greatest (or *maximum*) proper type w.r.t. subtyping, that is, any other proper type is a subtype of $\top$. The top type is found in most calculi that feature bounded quantification as a convenient means for modeling unbounded quantification. Since $\top$ is an upper bound of *any* type, declaring it as an upper bound on a type variable effectively leaves the variable unbounded. Dually, the *bottom type* $\bot$ is the least (or *minimum*) proper type w.r.t. subtyping. Compared to the top type, the bottom type features in few of the calculi with bounded quantification. However, it plays a crucial role in $F^{\omega}_{\cdot\cdot}$, as we will see shortly. As usual for a calculus in the F_{ω}-family, there are three forms of *abstraction*, one each for abstracting terms in terms ($\lambda x{:}A.\,t$), types in terms ($\lambda X{:}K.\,t$), and types in types ($\lambda X{:}K.\,A$). Each comes with an associated *application* form ($s\,t$, $t\,A$, and AB), and a type or kind former ($A \to B$, $\forall X{:}K.\,A$, and $(X{:}J) \to K$).

Following Pierce and others [43, 42, 16] we use Church-style syntax for abstractions, i.e. λs carry domain annotations (kinds for abstractions over type variables and types for term abstractions). These annotations are a mixed blessing: on the one hand, the kind annotations

$x, y, z, \dots$	Term variable	$X, Y, Z, \dots$	Type variable
$s, t ::=$	Term	$A, B, C ::=$	Type
x	term variable	X	type variable
$\lambda x{:}A.\, t$	term abstraction	$\top$	top/maximum type
$s\, t$	term application	$\bot$	bottom/minimum type
$\lambda X{:}K.\, t$	type abstraction	$A \to B$	function type
$t\, A$	type application	$\forall X{:}K.\, A$	universal type
$u, v, w ::=$	Value	$\lambda X{:}K.\, A$	operator abstraction
$\lambda x{:}A.\, t$	term abstraction	$A\, B$	operator application
$\lambda X{:}K.\, t$	type abstraction	$J, K, L ::=$	Kind
$\Gamma, \Delta ::=$	Typing context	$A\,..\,B$	type interval
$\varnothing$	empty context	$(X{:}J) \to K$	dependent operator kind
$\Gamma, x{:}A$	term variable binding	$j, k, l ::=$	Simple kind
$\Gamma, X{:}K$	type variable binding	$*$	kind of proper types
		$j \to k$	operator kind

Figure 3.1 – Syntax of $F^{\omega}_{..}$.

in operator abstractions will play a crucial role when we prove weak normalization for types and kinds in the next chapter; on the other hand, they complicate the formulation of some subtyping rules (as we will see in Section 3.2) and that of certain commutativity properties (as we will see in Sections 4.1.2 and 4.2.2). A Curry-style (or domain-free) development of $F^{\omega}_{..}$ might avoid these complications but would require a different approach to proving normalization of types.

The main difference between $F^{\omega}_{..}$ and other calculi in the F_ω-family is reflected in its kind language. Firstly, the usual kind of proper types $*$ is replaced by the *interval kind* former $A\,..\,B$. Intuitively, an interval kind $A\,..\,B$ represents the collection of types bounded by the pair of types A, B. It is inhabited by all proper types $C : A\,..\,B$ that are both supertypes of A and subtypes of B. The degenerate case $\bot\,..\,\top$ represents the collection of *all* proper types since every such type is bounded by $\bot$ from below and by $\top$ from above. In other words, $\bot\,..\,\top$ is simply the kind of proper types, and hence we define the shorthand $* = \bot\,..\,\top$.

Secondly, most variants of F_ω, irrespective of whether they allow subtyping or not, have a *simple* kind language (as described by the non-terminal k in Fig. 3.1). In contrast, $F^{\omega}_{..}$ has a *dependent* kind language. In particular, the arrow kind $(X{:}J) \to K$ acts as a binder for the type variable X which may appear freely in the codomain K. The reason for this choice should be clear: since types are allowed to appear in kinds (as the bounds of intervals), one may wish to use the argument type X to constrain the codomain K. For example, the result of a type operator of kind $(X{:}*) \to \bot\,..\,X$ is guaranteed to be a subtype of its argument. Dependent kinds also play an important role when modeling *bounded type operators*. For example, consider a binary type operator of kind $(X{:}*) \to (Y{:}\bot\,..\,X) \to *$. Here, the interval kind $\bot\,..\,X$ of the

second argument Y ensures that the operator can only applied to types A, B if the latter is a subtype of the former. This idea is not new: Compagnoni and Goguen's $\mathscr{F}^{\omega}_{\leq}$, which features upper-bounded type operators, also allows arrow kinds to be dependent [18].

Following common convention, we identify expressions e (i.e. terms, types and kinds), up to *α-equivalence*. In other words, two expressions are considered equal if they only differ in the names of their bound term or type variables. We write $e \equiv e'$ when we wish to stress the fact that the expressions e and e' are α-equivalent. The set of free variables of an expression e is denoted by $\mathsf{fv}(e)$. We write $s[x := t]$ to denote the capture-avoiding substitution of t for the free term variable x in the term s, and $e[X := A]$ to denote the type variable substitution of A for X in the expression e. For hygiene, we adopt Barendregt's variable convention [10], i.e. we assume the names of bound variables in any given expression to be different from those of free variables. This avoids accidental name capture by substitutions and other syntactic operations. When we wish to stress that a variable X is fresh w.r.t. an expression e, we write $X \notin \mathsf{fv}(e)$.

We generally assume that the variables bound in a typing context Γ be *distinct* so that we may think of contexts as finite maps from sets of bound variables to the associated types and kinds. We write $\mathsf{dom}(\Gamma)$ for the *domain* of Γ, i.e. the set of variables bound in Γ, $\Gamma(x)$ for the type associated with the term variable x and $\Gamma(X)$ for the kind associated with the type variable X. Given two contexts Γ and Δ with disjoint domains $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \varnothing$, we define (Γ, Δ) as the context obtained by concatenating Γ and Δ. When spelling out contexts, we typically omit the empty context $\varnothing$, writing e.g. $\Gamma = x{:}A, Y{:}K$ instead of $\Gamma = \varnothing, x{:}A, Y{:}K$.

We abbreviate $(X{:}J) \to K$ by $J \to K$ whenever X does not appear freely in K. This convention, together with the shorthand $*$ for $\bot\,..\,\top$, allows us to informally treat the language of simple kind expressions k as a sub-language of the (dependent) kind expressions K. More formally, there is an injection $\iota(k)$ of simple kinds into kinds given by $\iota(*) = \bot\,..\,\top$ and $\iota(j \to k) = (X{:}\iota(j)) \to \iota(k)$, which we will usually omit, writing k instead of $\iota(k)$. In the opposite direction, we define a *simplification* map $|K|$ from kinds to simple kinds which forgets any dependencies in the kind K (see Fig. 3.2). Given a kind K, we say K *simplifies to* $|K|$ or that $|K|$ is its *simplification*. We say that a pair of kinds J, K, *simplify equally* if $|J| \equiv |K|$. It is easy to verify that simplification is left-inverse to the injection ι, i.e. we have $|\iota(k)| = k$.

A useful property of simple kinds is that they are stable under substitution. We will make ample use of the following lemma in proofs throughout Chapters 4 and 5.

Lemma 3.1 (stability of simplifications)**.** *Let A be a type, K a kind and X a type variable that occurs freely in K. Then* $|K[X := A]| \equiv |K|$.

Proof. By straightforward induction on the structure of K. □

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

$$* = \bot \mathrel{..} \top \qquad \emptyset = \top \mathrel{..} \bot$$

$$A \mathrel{..}_{A' \mathrel{..} B'} B = A \mathrel{..} B \qquad A \mathrel{..}_{(X:J) \to K} B = (X:J) \to A\,X \mathrel{..}_K B\,X \quad \text{for } X \notin \mathsf{fv}(A) \cup \mathsf{fv}(B)$$

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

Higher-order extrema $\boxed{\top_K}$ $\boxed{\bot_K}$ **Kind simplification** $\boxed{|K|}$

$$\top_{A \mathrel{..} B} = \top \qquad \top_{(X:J) \to K} = \lambda X{:}J.\top_K$$

$$\bot_{A \mathrel{..} B} = \bot \qquad \bot_{(X:J) \to K} = \lambda X{:}J.\bot_K$$

$$|A \mathrel{..} B| = * \qquad |(X:J) \to K| = |J| \to |K|$$

Bounded quantification and type operators

$$\forall X \leq A{:}K.B = \forall X{:}(\bot_K) \mathrel{..}_K A.B \qquad (X \leq A{:}J) \to K = (X{:}(\bot_J) \mathrel{..}_J A) \to K$$

$$\lambda X \leq A{:}K.t = \lambda X{:}(\bot_K) \mathrel{..}_K A.t \qquad \lambda X \leq A{:}K.B = \lambda X{:}(\bot_K) \mathrel{..}_K A.B$$

Figure 3.2 – Syntactic shorthands and encodings

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 $F_{\leq}$-style universal type $\forall X \leq A.B$ can be expressed as $\forall X{:}\bot \mathrel{..} A.B$ in $F^{\omega}_{..}$. If we wish to extend this principle to *higher-order* bounded quantification and type operators, we need corresponding higher-order interval kinds and extremal types. $F^{\omega}_{..}$ does not provide intrinsic syntax for these constructs. Instead, they are encoded via type abstraction and dependent kinds, as shown in Fig. 3.2.

The encoding of *higher-order maxima* $\top_K$ is standard (see e.g. [43, 18]), and that of *higher-order minima* $\bot_K$ follows the same principle. The encoding of *higher-order interval kinds* $A \mathrel{..}_K B$ resembles that of higher-order *singleton kinds* given by Stone and Harper in [49]. This is not a coincidence: in $F^{\omega}_{..}$, singleton kinds are simply a special case of interval kinds where the upper and lower bounds coincide. Adopting Stone and Harper's notation for singletons, we define $S(A : K) = A \mathrel{..}_K A$. The intuition behind all these encodings is grounded in the fact that higher-order subtyping is just the pointwise lifting of subtyping on proper types, i.e. two type operators A and B of kind $K \to *$ are subtypes $A \leq B$ precisely if applying them to any argument $C : K$ results in the subtyping relationship $A\,C \leq B\,C$ on proper types. By the same reasoning, upper and lower bounds on types, such as extrema or the bounds of type intervals, may be lifted pointwise to form higher-order bounds. We will make this intuition more precise in Section 3.6, where we give a number of admissible subtyping rules for higher-order interval kinds and extremal types.

Note that the extremal types $\top_K$ and $\bot_K$ for a given kind K need not inhabit K. To see this, consider the singleton kind $K = S(\bot : *) = \bot \mathrel{..} \bot$. Clearly, $\top_K = \top$ is not contained in K. But

CBV reduction $\boxed{s \longrightarrow_{\mathsf{v}} t}$

$$\frac{}{(\lambda x{:}A.\, t)\, v \longrightarrow_{\mathsf{v}} t[x := v]} \;(\text{R-AppAbs}) \qquad \frac{}{(\lambda X{:}K.\, t)\, A \longrightarrow_{\mathsf{v}} t[X := A]} \;(\text{R-TAppTAbs})$$

$$\frac{s \longrightarrow_{\mathsf{v}} s'}{s\, t \longrightarrow_{\mathsf{v}} s'\, t} \;(\text{R-App}_1) \qquad \frac{t \longrightarrow_{\mathsf{v}} t'}{v\, t \longrightarrow_{\mathsf{v}} v\, t'} \;(\text{R-App}_2) \qquad \frac{t \longrightarrow_{\mathsf{v}} t'}{t\, A \longrightarrow_{\mathsf{v}} t'\, A} \;(\text{R-TApp})$$

Figure 3.3 – Call-by-value reduction

what is the smallest kind that contains both $\top_K$ and $\bot_K$? Given a kind K, we define $*_K$ as the kind that takes the same arguments as K but forgets the bounds of its eventual return kind (see Fig. 3.2). For proper type intervals, $*_{A\,..\,B}$ is simply $*$. Equivalently, the kind $*_K$ can be seen as a higher-order interval, namely $*_K = (\bot_K)\,..{}_K\,(\top_K)$. In other words, $*_K$ is exactly the smallest higher-order interval wide enough to accommodate the higher-order extremal types for K. We will prove the equivalence of the two definitions in Section 3.6.

With higher-order intervals and extrema in place, it is straightforward to express $\mathscr{F}^{\omega}_{\le}$-style higher-order bounded operators and universal quantifiers (see Fig. 3.2).

$F^{\omega}_{..}$ also admits an encoding of (higher-order) existential quantifiers (aka weak sums). The encoding of unbounded existentials is a straightforward generalization of the Church-encoding for existential quantifiers in System F; we refer the interested reader to [43, Section 24.3] for details. Starting from unbounded existentials, higher-order type intervals may then be used to express bounded existential quantification just as for the universal case.

3.1.2 Structural operational semantics

We adopt a standard call-by-value (CBV) semantics for $F^{\omega}_{..}$. The *one-step CBV reduction* relation $\longrightarrow_{\mathsf{v}}$ is defined in Fig. 3.3, and we denote its reflexive, transitive closure, *CBV reduction*, by $\longrightarrow^*_{\mathsf{v}}$. Since these relations are untyped, and therefore largely independent from the type and kind language, they are essentially identical to corresponding CBV reduction relations defined for F_ω (see e.g. [43, Fig. 30-1]).

The choice of CBV as the evaluation strategy is only significant insofar as it forbids reduction under binders. As we will see in Section 3.8, the *subject reduction* property (also known as *preservation*) does only hold for *closed* well-typed terms in $F^{\omega}_{..}$, i.e. for terms typed in the empty context. Reductions under binders, such as term or type abstractions, on the other hand, are unsafe. Hence, we could have picked a call-by-name (CBN) semantics instead, but not one based on e.g. full β-reduction.

Subject reduction does hold for *open types and kinds*, however. We define the *one-step β-reduction* relation $\longrightarrow_\beta$ on types and kinds as the compatible closure of β-contraction of type operators w.r.t all the type and kind formers (see Fig. 3.4). We write $\longrightarrow^*_\beta$ for its reflexive,

Full β-reduction in types $\boxed{A \longrightarrow_\beta B}$

$$\frac{}{(\lambda X{:}K.\,A)\,B \longrightarrow_\beta A[X := B]}\ (\text{TR-AppAbs}) \qquad \frac{A \longrightarrow_\beta A'}{A \to B \longrightarrow_\beta A' \to B}\ (\text{TR-Arr}_1) \qquad \frac{B \longrightarrow_\beta B'}{A \to B \longrightarrow_\beta A \to B'}\ (\text{TR-Arr}_2)$$

$$\frac{K \longrightarrow_\beta K'}{\forall X{:}K.\,A \longrightarrow_\beta \forall X{:}K'.\,A}\ (\text{TR-All}_1) \qquad \frac{K \longrightarrow_\beta K'}{\lambda X{:}K.\,A \longrightarrow_\beta \lambda X{:}K'.\,A}\ (\text{TR-Abs}_1) \qquad \frac{A \longrightarrow_\beta A'}{A\,B \longrightarrow_\beta A'\,B}\ (\text{TR-App}_1)$$

$$\frac{A \longrightarrow_\beta A'}{\forall X{:}K.\,A \longrightarrow_\beta \forall X{:}K.\,A'}\ (\text{TR-All}_2) \qquad \frac{A \longrightarrow_\beta A'}{\lambda X{:}K.\,A \longrightarrow_\beta \lambda X{:}K.\,A'}\ (\text{TR-Abs}_2) \qquad \frac{B \longrightarrow_\beta B'}{A\,B \longrightarrow_\beta A\,B'}\ (\text{TR-App}_2)$$

Full β-reduction in kinds $\boxed{J \longrightarrow_\beta K}$

$$\frac{A \longrightarrow_\beta A'}{A\,..\,B \longrightarrow_\beta A'\,..\,B}\ (\text{TR-Intv}_1) \qquad \frac{B \longrightarrow_\beta B'}{A\,..\,B \longrightarrow_\beta A\,..\,B'}\ (\text{TR-Intv}_2)$$

$$\frac{J \longrightarrow_\beta J'}{(X{:}J) \to K \longrightarrow_\beta (X{:}J') \to K}\ (\text{TR-DArr}_1) \qquad \frac{K \longrightarrow_\beta K'}{(X{:}J) \to K \longrightarrow_\beta (X{:}J) \to K'}\ (\text{TR-DArr}_2)$$

Figure 3.4 – Full β-reduction in types and kinds

transitive closure, *β-reduction*, and $=_\beta$ for its equivalence closure, *β-equality*. We discuss subject reduction for types and kinds in more detail in Section 3.7.

3.2 Declarative typing and kinding

The rules for typing of terms, kinding of types, and for deriving type and kind (in)equalities are given in Figures 3.5 and 3.6. There are quite a number of judgments involved, so we give a summary below. We refer to this set of rules as the *declarative* system (or presentation) of $F^\omega_{..}$ as opposed to the *simplified* and *canonical* systems we will introduce in the next two chapters. The rules for term typing are again very similar to those of F_ω, while the kinding rules differ in a few key respects. The rules for subtyping, subkinding and the equality relations on types and kinds differ significantly from those of other systems. In the remainder of this section, we will discuss each of the judgments in turn, explaining the design of the rules, especially where they differ from those found in other systems.

Judgments. Figures 3.5 and 3.6 define the following judgments by mutual induction.

$\Gamma\ \mathsf{ctx}$	the context Γ is well-formed
$\Gamma \vdash K\ \mathsf{kd}$	the kind K is well-formed in context Γ
$\Gamma \vdash J \leq K$	J is a subkind of K in Γ
$\Gamma \vdash J = K$	J and K are equal kinds in Γ
$\Gamma \vdash A : K$	the type A has kind K in context Γ
$\Gamma \vdash A \leq B : K$	A is a subtype of B in K and Γ
$\Gamma \vdash A = B : K$	A and B are equal types of kind K in Γ
$\Gamma \vdash t : A$	the term t has type A in context Γ

We sometimes write $\Gamma \vdash \mathscr{J}$ to denote an arbitrary judgment in the a context Γ. This includes the context formation judgment $\Gamma\ \mathsf{ctx}$ which we may think of as a nullary relation in the context Γ.

Throughout this thesis, we assume the following *well-scopedness* conventions for judgments without mentioning them explicitly in individual definitions. If $\Gamma \vdash \mathscr{J}$ then

1. following our convention for typing contexts, all the variables bound in Γ are assumed distinct, in particular, if $\Gamma, X{:}K \vdash \mathscr{J}$ then $X \notin \mathsf{dom}(\Gamma)$;
2. only variables bound in Γ are allowed to occur freely to the right of the turnstile, i.e. $\mathsf{fv}(\mathscr{J}) \subseteq \mathsf{dom}(\Gamma)$.

These conventions apply in particular to all of the judgments listed above.

3.2.1 Context and kind formation

The rules for context formation $\Gamma\ \mathsf{ctx}$ are standard. Starting from the empty context, term and type variable bindings may be added provided the declared types and kinds are themselves well-formed. Since types and kinds may contain type variables as sub-expressions, the order of bindings in a context matters. The declared types and kinds of a binding may refer to type variables introduced earlier – those to the left of the binding – but not to the variable being introduced. As usual, the declared type of a term variable must be a proper type. The rules of the remaining judgments are set up so that they can only be derived in well-formed contexts.

In most variants of F_ω, kinds are simple and form a separate syntactic category. Such kinds are necessarily well-formed. But in $F^\omega_{..}$, kinds are type-dependent, so ill-formed kind expressions, such as $\bot\,..\,\lambda X{:}K.\,A$ or $(X{:}*) \to X\,\top$, may arise and need to be excluded from derivations through the use of a kind formation judgment $\Gamma \vdash K\ \mathsf{kd}$. Because types may appear in kinds, this judgment needs to be mutually defined with the kinding judgment $\Gamma \vdash A : K$ – as witnessed by the interval formation rule WF-INTV. A kind interval is well-formed if its lower and upper bounds are proper types. No effort is made to prevent the formation of intervals where the declared lower bound is not a subtype of the declared upper bound. For example, the empty kind $\emptyset = \top\,..\,\bot$ is a well-formed interval in any well-formed context, as demonstrated by the

Context well-formedness $\boxed{\Gamma\ \mathsf{ctx}}$

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

Kind well-formedness $\boxed{\Gamma \vdash K\ \mathsf{kd}}$

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

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

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

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

$$\frac{\Gamma \vdash J\ \mathsf{kd} \qquad \Gamma, X{:}J \vdash A : K}{\Gamma \vdash \lambda X{:}J.\, A : (X{:}J) \to K}\ (\textsc{K-Abs}) \qquad \frac{\Gamma \vdash A : (X{:}J) \to K \qquad \Gamma \vdash B : J}{\Gamma \vdash A\,B : K[X := B]}\ (\textsc{K-App})$$

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

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

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

$$\frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : * \qquad \Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x{:}A.\, t : A \to B}\ (\textsc{T-Abs}) \qquad \frac{\Gamma \vdash t : \forall X{:}K.\, A \qquad \Gamma \vdash B : K}{\Gamma \vdash t\,B : A[X := B]}\ (\textsc{T-TApp})$$

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

Figure 3.5 – Declarative presentation of $F^{\omega}_{..}$ – part 1

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

$$\frac{\Gamma \vdash A_2 \leq A_1 : * \qquad \Gamma \vdash B_1 \leq B_2 : *}{\Gamma \vdash A_1 \mathinner{..} B_1 \leq A_2 \mathinner{..} B_2} \text{(SK-INTV)}$$

$$\frac{\Gamma \vdash J_2 \leq J_1 \qquad \Gamma, X{:}J_2 \vdash K_1 \leq K_2 \qquad \Gamma \vdash (X{:}J_1) \to K_1 \ \mathsf{kd}}{\Gamma \vdash (X{:}J_1) \to K_1 \leq (X{:}J_2) \to K_2} \text{(SK-DARR)}$$

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

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash A \leq A : K} \text{(ST-REFL)}$$

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

$$\frac{\Gamma \vdash A : B \mathinner{..} C}{\Gamma \vdash A \leq \top : *} \text{(ST-TOP)}$$

$$\frac{\Gamma \vdash A : B \mathinner{..} C}{\Gamma \vdash \bot \leq A : *} \text{(ST-BOT)}$$

$$\frac{\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J}{\Gamma \vdash (\lambda X{:}J.\, A)\, B \leq A[X := B] : K[X := B]} \text{(ST-}\beta_1\text{)}$$

$$\frac{\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J}{\Gamma \vdash A[X := B] \leq (\lambda X{:}J.\, A)\, B : K[X := B]} \text{(ST-}\beta_2\text{)}$$

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

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

$$\frac{\Gamma \vdash A_2 \leq A_1 : * \qquad \Gamma \vdash B_1 \leq B_2 : *}{\Gamma \vdash A_1 \to B_1 \leq A_2 \to B_2 : *} \text{(ST-ARR)}$$

$$\frac{\Gamma \vdash K_2 \leq K_1 \qquad \Gamma, X{:}K_2 \vdash A_1 \leq A_2 : * \qquad \Gamma \vdash \forall X{:}K_1.\, A_1 : *}{\Gamma \vdash \forall X{:}K_1.\, A_1 \leq \forall X{:}K_2.\, A_2 : *} \text{(ST-ALL)}$$

$$\frac{\Gamma, X{:}J \vdash A_1 \leq A_2 : K \qquad \Gamma \vdash \lambda X{:}J_1.\, A_1 : (X{:}J) \to K \qquad \Gamma \vdash \lambda X{:}J_2.\, A_2 : (X{:}J) \to K}{\Gamma \vdash \lambda X{:}J_1.\, A_1 \leq \lambda X{:}J_2.\, A_2 : (X{:}J) \to K} \text{(ST-ABS)}$$

$$\frac{\Gamma \vdash A_1 \leq A_2 : (X{:}J) \to K \qquad \Gamma \vdash B_1 = B_2 : J}{\Gamma \vdash A_1\, B_1 \leq A_2\, B_2 : K[X := B_1]} \text{(ST-APP)}$$

$$\frac{\Gamma \vdash A : B_1 \mathinner{..} B_2}{\Gamma \vdash B_1 \leq A : *} \text{(ST-BND}_1\text{)}$$

$$\frac{\Gamma \vdash A : B_1 \mathinner{..} B_2}{\Gamma \vdash A \leq B_2 : *} \text{(ST-BND}_2\text{)}$$

$$\frac{\Gamma \vdash A_1 \leq A_2 : B \mathinner{..} C}{\Gamma \vdash A_1 \leq A_2 : A_1 \mathinner{..} A_2} \text{(ST-INTV)}$$

$$\frac{\Gamma \vdash A_1 \leq A_2 : J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash A_1 \leq A_2 : K} \text{(ST-SUB)}$$

Kind equality $\boxed{\Gamma \vdash J = K}$

$$\frac{\Gamma \vdash J \leq K \qquad \Gamma \vdash K \leq J}{\Gamma \vdash J = K} \text{(SK-ANTISYM)}$$

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

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

Figure 3.6 – Declarative presentation of $F^{\omega}_{::}$ – part 2

following lemma.

Lemma 3.2. *The kind of proper types $*$ and the empty kind $\varnothing$ are both well-formed in any well-formed context, i.e. the following are derivable.*

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash * \ \mathsf{kd}}\ (\textsc{Wf-Star}) \qquad\qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \varnothing\ \mathsf{kd}}\ (\textsc{Wf-Empty})$$

Proof. Both derivations use only K-BOT, K-TOP and WF-INTV. □

We say that the bounds of an interval $A\,..\,B$ are *consistent* in some context Γ if $\Gamma \vdash A \leq B : *$, i.e. if A is provably a subtype of B in Γ, and *inconsistent* otherwise. We say that the bounds of $A\,..\,B$ are *absurd* if they are inconsistent even as closed types, i.e. if $\vdash A : *$ and $\vdash B : *$ but $\vdash A \not\leq B : *$. For example, the bounds of $*$ are consistent in the empty context while those of $\varnothing$ are absurd. Given the context $\Gamma = X{:}*, Y{:}*$, the bounds of the interval $X\,..\,Y$ are inconsistent because we can cannot establish that $\Gamma \vdash X \leq Y : *$, but not absurd because X and Y are open types. We discuss advantages and drawbacks of allowing well-formed interval kinds with inconsistent bounds at the end of this chapter.

The formation rule WF-DARR for dependent arrow kinds is just the standard formation rule for dependent arrow types (aka dependent product types) lifted to the kind level.

3.2.2 Kinding and typing

The kinding rules K-VAR, K-TOP and K-BOT for variables and extremal types are standard. Since these three rules are the "leaves" of kinding derivations, each of them has a premise ensuring the well-formedness of its context.

The formation rules K-ARR and K-ALL for arrow and universal types are also standard. They simply ensure that all their sub-expressions are well-formed. The rule K-ALL resembles the formation rule for universal quantifiers in F_ω but differs from the corresponding rules found in systems with bounded polymorphism because bounds on the variable X are expressed in the kind K rather than featuring explicitly in the universal type constructor. Similarly, the introduction and elimination rules for type operators, K-ABS and K-APP, resemble more closely those found in F_ω than those in e.g. $F^\omega_{<:}$ or $\mathscr{F}^\omega_{\leq}$. The extra premise $\Gamma \vdash J\ \mathsf{kd}$ ensures that the domain J of an abstraction is a well-formed kind. It is usually omitted in systems where kinds are simple, though it does appear in presentations of F_ω that treat terms, kinds and types as a single syntactic category, such as those found in the Pure Type System (PTS) literature (see e.g. [10]).

The last two kinding rules, K-SING and K-SUB, are used to adjust the kind of a type. The subsumption rule K-SUB is just a kind-level analog of the rule T-SUB and reflects the intuition that the inhabitants of a kind J also inhabit any superkind K of J. The rule K-SING acts as an introduction rule for interval kinds or, more specifically, for singleton kinds. It states that any proper type A inhabits the (singleton) interval $A\,..\,A$ which is upper and lower-bounded

by A itself. Similar singleton introduction rules have been proposed by Stone and Harper for kinds [49] and by Aspinall for types [8]. Note that the singleton introduction rule potentially *narrows* the kind of a type, i.e. the kind $A \mathinner{..} A$ assigned to A in the conclusion might be a subkind of the kind $B \mathinner{..} C$ assigned to A in the premise. This is in contrast to the subsumption rule which can only *widen* the kind of a type, i.e. we have $J \leq K$.

The typing rules are 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. Examples are the rules T-VAR and T-TABS: the former has a premise ensuring the well-formedness of the context in which the variable is typed, the latter requires well-formedness of the domain of the type abstraction being introduced. The type abstraction and elimination rules T-ABS and T-TAPP resemble again more closely those found in F_ω than those found in calculi with bounded polymorphism. The rules T-ABS and T-APP are just the usual introduction and elimination rules for (simple) arrow types. Finally, the subsumption rule T-SUB allows the type A of a term to be widened to any proper supertype B of A, i.e. any type such that $\Gamma \vdash A \leq B : *$.

3.2.3 Subkinding and subtyping

Unlike most other calculi in the F_ω-family, $F^\omega_{..}$ features both subtyping and *subkinding*. The use of subkinding in $F^\omega_{..}$ is both natural and essential. It is natural since kinds are type-dependent. In particular, if a proper type A is contained in some type interval $B \mathinner{..} C$, then one naturally expects A to also be contained in a *wider* interval $B' \mathinner{..} C'$, where $B' \leq B$ and $C \leq C'$. This suggests a containment order on interval kinds – wider intervals contain smaller intervals – which generates subkinding and is captured by the rule SK-INTV. Stone and Harper define analogous rules for the special case of singleton kinds [49], which are admissible in $F^\omega_{..}$.

Lemma 3.3. *The following subkinding rules for singletons are admissible.*

$$\frac{\Gamma \vdash A : *}{\Gamma \vdash S(A : *) \leq *}\ (\text{SK-SINGSTAR}) \qquad\qquad \frac{\Gamma \vdash A_1 = A_2}{\Gamma \vdash S(A_1 : *) \leq S(A_2 : *)}\ (\text{SK-SING})$$

Proof. The rule SK-SINGSTAR is derivable using ST-TOP, ST-BOT and SK-INTV. The proof of SK-SING follows immediately from the premises of ST-ANTISYM – the only rule that can be used to derive $\Gamma \vdash A_1 = A_2$ – and SK-INTV. □

Subkinding is also essential. It is thanks to the interval subkinding rule SK-INTV that we can express bounded polymorphism and bounded type operators in $F^\omega_{..}$. Consider the example of a polymorphic term $t : \forall X \leq A{:}*. B$ and assume we are given some type argument $C : *$ such that $C \leq A : *$. We would like to apply t to C. Desugaring the type of t as per the encoding of bounded universal quantifiers discussed in Section 3.1.1, we obtain $t : \forall X{:}\bot \mathinner{..} A. B$. How can we apply this term to the type C even though its kind $*$ differs from the expected kind $\bot \mathinner{..} A$? Thanks to the singleton introduction rule K-SING discussed in the previous section, we know that C also has kind $C \mathinner{..} C$, and thanks to subkinding, it is then sufficient to show that

this (singleton) interval is contained in the interval $\bot\,..\,A$ – which is indeed the case. The full derivation is

$$\dfrac{\Gamma \vdash t : \forall X{:}\bot\,..\,A.B \qquad \dfrac{(\textsc{K-Sing})\;\dfrac{\Gamma \vdash C : *}{\Gamma \vdash C : C\,..\,C} \qquad \dfrac{\dfrac{\Gamma \vdash C : *}{\Gamma \vdash \bot \leq C : *}\;(\textsc{ST-Bot}) \qquad \Gamma \vdash C \leq A : *}{\Gamma \vdash C\,..\,C \leq \bot\,..\,A}\;(\textsc{SK-Intv})}{\Gamma \vdash C : \bot\,..\,A}\;(\textsc{K-Sub})}{\Gamma \vdash t\,C : B[X := C]}\;(\textsc{T-TApp})$$

We will generalize this principle to higher-order bounded quantifiers and type operators in Section 3.6.

The rule SK-DARR lifts the interval containment order through dependent arrow kinds. As usual, the dependent arrow constructor is contravariant in its domain and covariant in its codomain. Since arrow kinds are dependent, we have to extend the context with a binding for X when comparing the codomains; but what should be the declared kind of this binding? Since $\Gamma \vdash J_2 \leq J_1$, the kind $X : J_2$ corresponds to a stronger assumption: wherever $X : J_2$, we expect $X : J_1$ to hold as well, but the opposite need not be true. However, we would still like the codomain K_1 of the left-hand side $(X{:}J_1) \to K_1$ to be well-formed under the weaker assumption $X : K_1$. The extra premise $\Gamma \vdash (X{:}J_1) \to K_1$ kd ensures that this is the case. Aspinall and Compagnoni use a similar condition in their (S-π) rule for subtyping dependent product types [9].

Subtyping judgments $\Gamma \vdash A \leq B : K$ are indexed by a common kind K in which the types A and B are related. This common kind is relevant because two types A, B may be related when seen as inhabitants of some common kind K but not when seen as inhabitants of other, possibly distinct kinds $K_1 \neq K_2$. For example, the extremal types $\top$ and $\bot$ are related as proper types, i.e. $\vdash \bot \leq \top : *$, but not as inhabitants of their respective singleton kinds $\bot\,..\,\bot$ and $\top\,..\,\top$, i.e. letting $K_1 = \bot\,..\,\bot$ and $K_2 = \top\,..\,\top$, we have $\vdash \bot : K_1$ and $\vdash \bot : K_2$ but $\vdash K_1 \not\leq K_2$ and $\vdash K_2 \not\leq K_1$, so that $\bot$ and $\top$ are related in neither K_1 nor K_2.

For a given context Γ and kind K, the subtyping relation $\Gamma \vdash A \leq B : K$ is a preorder, i.e. it is reflexive and transitive. As is customary for declarative presentations of subtyping, we include a pair of rules, ST-REFL and ST-TRANS, to reflect this fact. Note that there are no such rules for subkinding. Instead, we will prove them admissible in Section 3.3.2. In Chapter 5, we will see that some (but not all) instances of ST-REFL and ST-TRANS can be eliminated.

The rules ST-TOP and ST-BOT establish $\top$ and $\bot$ as the maximum and minimum proper types w.r.t. subtyping. Both rules have a single premise $\Gamma \vdash A : B\,..\,C$ ensuring that the extremal types are only related to other proper types. In Section 3.6, we will show that similar rules are admissible for the higher-order extrema $\top_K$ and $\bot_K$ defined earlier. The kinding of the premises of ST-TOP and ST-BOT may be a bit surprising: the kind $B\,..\,C$ does not match the common kind $*$ in the conclusion. As we are about to see, every type inhabiting a proper type interval also inhabits $*$, so why not simply use the premise $\Gamma \vdash A : *$? It turns out that the proof

of the very fact that proper types inhabit $*$ depends crucially on the extra flexibility in ST-TOP and ST-BOT.

Lemma 3.4. *Types inhabiting interval kinds are proper types. If* $\Gamma \vdash A : B \,..\, C$*, then also* $\Gamma \vdash A : *$.

Proof. The result is derivable as

$$\dfrac{(\text{K-SING})\,\dfrac{\Gamma \vdash A : B\,..\,C}{\Gamma \vdash A : A\,..\,A} \qquad \dfrac{(\text{ST-BOT})\,\dfrac{\Gamma \vdash A : B\,..\,C}{\Gamma \vdash \bot \leq A : *} \quad \dfrac{\Gamma \vdash A : B\,..\,C}{\Gamma \vdash A \leq \top : *}\,(\text{ST-TOP})}{\Gamma \vdash A\,..\,A \leq \bot\,..\,\top}\,(\text{SK-INTV})}{\Gamma \vdash A : \bot\,..\,\top}\,(\text{K-SUB})$$

□

Note that the proof makes essential use of the rules ST-BOT and ST-TOP in a way that would be impossible if we were to strengthen their premises as suggested above.

The rules ST-β_1 and ST-β_2 correspond to β-contraction and reduction, respectively. Together, they ensure that β-convertible types are subtypes. The two separate rules are necessary because subtyping is not symmetric. Alternatively, we could have combined the two rules into a single type equality rule but that would have complicated the definition of type equality which, in its current form, is pleasingly simple and easy to work with.

Similarly, the rules ST-η_1 and ST-η_2 relate η-convertible types. Most presentations of F_ω do not include rules for η-conversion but they play an important role in $F^\omega_{..}$ for two reasons. Firstly, on a conceptual level, they are necessary to relate type operators to the higher-order extrema and interval types defined in Section 3.1.1. Hence they feature prominently in the proofs of properties associated with these encodings, in particular the admissible formation, (sub)kinding and (sub)typing rules (see Section 3.6). Secondly – and perhaps more importantly – on a metatheoretic level, the η-rules will prove essential in establishing an equivalence between the declarative presentation of subtyping given in this chapter and the alternative, canonical presentation we will give in Chapter 5. This equivalence is at the heart of our type safety proof for $F^\omega_{..}$. As we will see in in Chapter 5, canonical subtyping judgments relate only η-long β-normal types, which is why both β and η rules are necessary to establish the equivalence. There are some other variants of F_ω where η-conversion is of similar importance. Stone and Harper's $\lambda^{\Pi\Sigma S}_{\leq}$, admits an encoding of higher-order singleton types similar to our encoding of type intervals [49]. The system features η-like extensionality rules which are crucial to relate singleton kinds to their inhabitants. Abel and Rodriguez prove that subtyping is decidable for a variant of $F^\omega_{<:}$ with η-conversion [2]. Their proof makes essential use of an algorithmic subtyping judgment that only relates η-long β-normal types; a type equality judgment including η-conversion is used to relate the algorithmic judgment to a declarative one.

The rules ST-ARR and ST-ALL for subtyping instances of the arrow and universal type formers are again fairly standard. Both are contravariant in the domain and contravariant in the codomain. The rule for universals is similar to the subkinding rule for dependent arrow

kinds SK-DARR in that we compare the codomains under the stronger assumption $X : K_2$ and ensure well-formedness of the left-hand side via the additional premise $\Gamma \vdash \forall X{:}K_1.\, A_1 : *$.

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. As we will see in Section 3.3.2, the usual rules can be recovered via subtyping reflexivity and antisymmetry. Since we do not track the variance (or polarity) of type operators, applications of operators are considered subtypes only if their arguments agree up to type equality. Hence, the rule ST-APP uses subtyping to compare the heads of applications and type equality to compare their arguments. Since arrow kinds are dependent in $F^{\omega}_{..}$, a suitable type must be substituted in the conclusion for the free type variable X in the codomain K. The arguments B_1 and B_2 are both equally suitable candidates; we pick the former.

The rule for subtyping operator abstractions, ST-ABS, is maybe the most unusual when compared to other variants of F_ω. Typically, one would find a version of the following, weaker subtyping rule instead.

$$\frac{\Gamma \vdash J_1 = J_2 \qquad \Gamma, X{:}J_1 \vdash A_1 \leq A_2 : K}{\Gamma \vdash \lambda X{:}J_1.\, A_1 \leq \lambda X{:}J_2.\, A_2 : (X{:}J_1) \to K} \ (\text{ST-ABSWEAK})$$

The premises of the rule reflect the facts that (i) type equality is a congruence w.r.t. operator abstractions, and (ii) subtyping is lifted pointwise to type operators. The first premise may be omitted in a system with simple kinds or if there is a separate equality rule for operator abstractions.

The strengthened rule ST-ABS retains point (ii) but allows abstractions to be subtypes even if their domain annotations J_1 and J_2 do not match. Intuitively, these annotations are part of the signatures (i.e. the kinds) of the respective abstractions, not of the operators itself; therefore they should only matter for subtyping insofar as they ensure that the abstractions being related are of the same kind. This would be obvious if we used a Curry-style representation of operator abstractions and omitted domain annotations altogether. But is this intuition justified for our Church-style presentation? As the following example illustrates, it is.

Let Γ be some typing context containing the binding $X : * \to *$ and A a proper type in Γ, so that we have $\Gamma \vdash X : * \to *$ and $\Gamma \vdash A : *$. The abstract type operator X has declared kind $* \to *$, but by subsumption, it also has kind $\Gamma \vdash X : (Y{:}\bot \,..\, A) \to *$. That is, the unbounded type operator X may be used wherever an A-bounded type operator is required. By subsumption, ST-ETA$_1$ and ST-ETA$_2$, it follows that

$$\begin{array}{lllllll} \Gamma \vdash & \lambda Y{:}*.\, X\, Y & \leq & X & \leq & \lambda Y{:}\bot \,..\, A.\, X\, Y & : (Y{:}\bot \,..\, A) \to * \\ \Gamma \vdash & \lambda Y{:}\bot \,..\, A.\, X\, Y & \leq & X & \leq & \lambda Y{:}*.\, X\, Y & : (Y{:}\bot \,..\, A) \to * \end{array}$$

i.e. the two η-expansions of X are mutual subtypes, despite the fact that their type annotations differ (unless $A = \top$). Seeing as such subtyping relationships can already be established via

the η-rules, it seems reasonable to adopt a subtyping rule for abstractions that allows us to derive them directly.

The first premise of ST-Abs says that the bodies of the two abstractions are subtypes in the common codomain K, assuming a parameter in the common domain J. The two remaining premises ensure that both abstractions – irrespective of their domain annotations – inhabit the common arrow kind $(X{:}J) \to K$. We will show in Section 3.7 that the weaker rule ST-AbsWeak given above remains admissible.

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 an instance $A : B\,..\,C$ of a type interval $B\,..\,C$, ST-Bnd$_1$ and ST-Bnd$_2$ assert that the types A and B are indeed lower and upper bounds, respectively, of A. When A is a variable, we may use rule ST-Bnd$_2$ together with K-Var to derive judgments of the form $\Gamma \vdash X \leq C$, similar to those obtained using the familiar variable subtyping rule from $F_{\leq}$. In other words, the bound projection rules allow us to *reflect* subtyping assumptions into subtyping judgments. In fact, together with ST-Trans, they can be used to reflect arbitrary subtyping assumptions $\Gamma(X) = A\,..\,B$ – consistent or not – into corresponding subtyping judgments $\Gamma \vdash A \leq B$.

$$\dfrac{\text{(ST-Bnd}_1)\ \dfrac{\text{(K-Var)}\ \dfrac{\Gamma\ \mathsf{ctx} \qquad \Gamma(X) = A\,..\,B}{\Gamma \vdash X : A\,..\,B}}{\Gamma \vdash A \leq X : *} \qquad \dfrac{\dfrac{\Gamma\ \mathsf{ctx} \qquad \Gamma(X) = A\,..\,B}{\Gamma \vdash X : A\,..\,B}\ \text{(K-Var)}}{\Gamma \vdash X \leq B : *}\ \text{(ST-Bnd}_2)}{\Gamma \vdash A \leq B : *}\ \text{(ST-Trans)}$$

We will discuss the ramifications of such derivations for type safety in Section 3.8.

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: if a pair of types A, B is related in some kind J, they remain related in any superkind K of J. The rule ST-Intv, on the other hand, is the subtyping counterpart of the interval introduction rule K-Sing: if A is a subtype of B in some interval $C\,..\,D$, then surely A is still a subtype of B in the interval $A\,..\,B$ bounded by those very same types. Indeed, $A\,..\,B$ is the smallest interval in which the two types are related. Hence, as for the corresponding kinding rules, ST-Sub generally widens the kind of a subtyping judgment, while ST-Intv generally narrows it.

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. As their names suggest, these rules make subkinding and subtyping antisymmetric w.r.t. kind and type equality by definition. 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, and

such proofs are generally rather challenging (see e.g. [17]). In $F^{\omega}_{..}$, antisymmetry is not an admissible property, however. To see this, consider a pair of proper types A and B such that $\Gamma \vdash A : B \,..\, B$ in some context Γ. Since B is both a lower and an upper bound of A, we would like to identify the two types, i.e. we would like to conclude that $\Gamma \vdash A = B : *$. But without antisymmetry, this is not always possible. E.g. if A is a type variable $A = X$ such that $\Gamma(X) = B \,..\, B$, we can derive that X and B are mutual subtypes using the bound projection rules ST-BND$_1$ and ST-BND$_2$, but without antisymmetry we have no way to derive $\Gamma \vdash X = B : *$.

One might argue that, instead of antisymmetry, we should introduce a rule to reflect singleton instances into type equality judgments, such as the following rule due to Stone and Harper [49].

$$\frac{\Gamma \vdash A : S(B : *)}{\Gamma \vdash A = B : *}\ (\text{TEQ-SING})$$

Interestingly, antisymmetry of proper types can be derived immediately using this rule and some other rules about type intervals. Let A, B be a pair of (provably) proper types such that $\Gamma \vdash A \leq B : *$ and $\Gamma \vdash B \leq A : *$. Then

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A : *}{\Gamma \vdash A : A \,..\, A}\ (\text{K-SING}) \qquad \dfrac{\Gamma \vdash B \leq A : * \qquad \Gamma \vdash A \leq B : *}{\Gamma \vdash A \,..\, A \leq B \,..\, B}\ (\text{SK-INTV})}{\Gamma \vdash A : B \,..\, B}\ (\text{K-SUB})}{\Gamma \vdash A = B : *}\ (\text{TEQ-SING})$$

As we will see in Section 3.6, similar properties hold for inhabitants $A : S(B : K)$ of higher-order singletons, so this principle can be extended to type equalities in arbitrary kinds K. That is, assuming suitable congruence rules for kind and type equality, covering at least the kind formers, as well as operator abstractions and applications. In summary, replacing SK-ANTISYM and ST-ANTISYM with a rule like TEQ-SING would not break antisymmetry but "obfuscate" it, at the expense of additional type and kind equality rules, which would be necessary to recover the expressiveness of the current equality judgments. In light of this, the current set of rules seems preferable.

Because the antisymmetry rules are the only way to derive kind and type equalities, any property that holds for mutual subkinds or subtypes also holds for equal kinds or types, respectively. For example, admissibility of the following subsumption rule for type equality follows immediately from the rule ST-SUB and antisymmetry.

Corollary 3.5. *The following subsumption rule for type equality is admissible.*

$$\frac{\Gamma \vdash A = B : J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash A = B : K}\ (\text{TEQ-SUB})$$

In what follows, we sometimes need to relate the bindings appearing in two syntactically different contexts Γ and Δ. To this end, we define *context equality* $\Gamma = \Delta\ \text{ctx}$ as the pointwise lifting of type and kind equality to context bindings:

$$\frac{}{\varnothing = \varnothing\ \mathsf{ctx}} \qquad \frac{\Gamma = \Delta\ \mathsf{ctx} \qquad \Gamma \vdash J = K}{\Gamma, X{:}J = \Delta, X{:}K\ \mathsf{ctx}} \qquad \frac{\Gamma = \Delta\ \mathsf{ctx} \qquad \Gamma \vdash A = B : *}{\Gamma, x{:}A = \Delta, x{:}B\ \mathsf{ctx}}$$

3.3 Basic metatheoretic properties

With the dynamics and statics in place, we can begin our work on the metatheory of $F^{\omega}_{\cdot\cdot}$. In this section, we present some basic properties, which can be roughly divided into two categories. First, we show that all the judgments are preserved under various operations on contexts (weakening, substitution, narrowing); second, we establish some order-theoretic properties about the four binary relations on kinds and types (subkinding, subtyping, kind and type equality). As a warm-up, we prove our earlier claim that judgments can only be derived in well-formed contexts.

Lemma 3.6 (context validity)**.** *Let* Γ *be a context. If* $\Gamma \vdash \mathscr{J}$ *for any of the judgments defined above, then* $\Gamma\ \mathsf{ctx}$.

Proof. By (simultaneous) induction on the derivations of the various judgments. The cases for context formation judgments and rules that contain $\Gamma\ \mathsf{ctx}$ as a premise are trivial. For other rules, the result follows by applying the IH to any of the premises that do not extend the context. There is always at least one such premise. □

3.3.1 Substitution lemmas

The next three lemmas are the cornerstones of our metatheory. They establish that the various judgments of our theory are preserved under certain well-formed transformations on contexts: addition of a well-formed binding, i.e. *weakening* of the context, *substitution* of a term or type variable by an identically typed or kinded term or type, respectively, and substitution of a binding for one with a wider kind or type, i.e. *context narrowing*.

Lemma 3.7 (weakening)**.** *A judgment remains true if its context is extended by a well-formed binding. Let* Γ, Δ *be contexts and* $\Gamma, \Delta \vdash \mathscr{J}$ *for any of the judgments defined above. Then*

1. *for any type* A *and* $x \notin \mathsf{dom}(\Gamma, \Delta)$, *if* $\Gamma \vdash A : *$, *then* $\Gamma, x{:}A, \Delta \vdash \mathscr{J}$;
2. *for any kind* K *and* $X \notin \mathsf{dom}(\Gamma, \Delta)$, *if* $\Gamma \vdash K\ \mathsf{kd}$, *then* $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$.

Corollary 3.8 (iterated weakening)**.** *If* $\Gamma, \Delta\ \mathsf{ctx}$ *and* $\Gamma \vdash \mathscr{J}$, *then* $\Gamma, \Delta \vdash \mathscr{J}$.

Lemma 3.9 (substitution)**.** *A judgment remains true if a suitable well-formed expression is substituted for one of the variables bound in its context.*

1. *If* $\Gamma \vdash t : A$ *and* $\Gamma, x{:}A, \Delta \vdash \mathscr{J}$, *then* $\Gamma, \Delta \vdash \mathscr{J}[x := t]$.
2. *If* $\Gamma \vdash A : K$ *and* $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$, *then* $\Gamma, \Delta[X := A] \vdash \mathscr{J}[X := A]$.

Lemma 3.10 (context narrowing – weak version)**.** *A judgments remains true if the type or kind of one of the variables bound in its context is narrowed.*

1. *If* $\Gamma \vdash A : *$, $\Gamma \vdash A \leq B : *$ *and* $\Gamma, x{:}B, \Delta \vdash \mathscr{J}$, *then* $\Gamma, x{:}A, \Delta \vdash \mathscr{J}$.
2. *If* $\Gamma \vdash J\ \mathsf{kd}$, $\Gamma \vdash J \leq K$ *and* $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$, *then* $\Gamma, X{:}J, \Delta \vdash \mathscr{J}$.

Corollary 3.11 (context conversion)**.** *If* $\Gamma\ \mathsf{ctx}$, $\Gamma = \Delta\ \mathsf{ctx}$ *and* $\Delta \vdash \mathscr{J}$, *then* $\Gamma \vdash \mathscr{J}$.

The context narrowing lemma is a bit weaker than one might expect. In particular, the premises $\Gamma \vdash A : *$ and $\Gamma \vdash J\ \mathsf{kd}$ seem redundant – surely, if $\Gamma \vdash A \leq B : *$ then A and B ought to be proper types. This property – called *subtyping validity* – does indeed hold, but we are not quite ready to prove it yet. Indeed, one of the prerequisites of the proof is the context narrowing lemma itself. Once we have established subtyping validity, we will strengthen Lemma 3.10; until then we will have to make due with the weak version.

Lemmas 3.7, 3.9 and 3.10 are proven in that order, each by simultaneous induction on the derivations of the various judgments. The proofs of Lemmas 3.9 and 3.10 rely on Corollary 3.8 for the variable cases T-VAR and K-VAR. All three proofs are entirely standard, so we only present an excerpt from the proof of Lemma 3.9 to illustrate the basic strategy. In it, we will make use of the following helper lemma about substitutions.

Lemma 3.12 (substitutions commute)**.** *Let e be some arbitrary expression, A, B types and X, Y distinct type variables such that $X \notin \mathsf{fv}(B)$. Then* $e[X := A][Y := B] \equiv e[Y := B][X := A[Y := B]]$

Proof. By induction on the structure of e. □

Proof of Lemma 3.9. The two parts are proven separately, each by induction on the derivation of the second premise ($\Gamma, x{:}A, \Delta \vdash \mathscr{J}$ for the first part, $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$ for the second). For the context formation judgment, the proofs proceed by a local induction on the structure of Δ. We show the cases for K-VAR and ST-β_1 for the second part of the lemma. The other cases are similar.

- *Case* K-VAR. $\mathscr{J}$ is $X : J$ and we have $\Gamma, X{:}K, \Delta\ \mathsf{ctx}$ and $(\Gamma, X{:}K, \Delta)(Y) = J$. By the IH, we get $\Gamma, \Delta[X := A]\ \mathsf{ctx}$. We distinguish two sub-cases based on whether $Y = X$.
  - *Sub-case* $Y = X$. We want to show that $\Gamma, \Delta[X := A] \vdash A : K[X := A]$. By well-scopedness, X does not occur freely in K, so we have $K[X := A] \equiv K$. The desired result follows from applying iterated weakening (Corollary 3.8) to the premise $\Gamma \vdash A : K$.
  - *Sub-case* $Y \neq X$. We want to show that $\Gamma, \Delta[X := A] \vdash Y : J[X := A]$. Since the domains of Γ and Δ are disjoint, Y must appear either in Γ or Δ but not in both. If $Y \in \mathsf{dom}(\Gamma)$, then X does not occur freely in $J = \Gamma(Y)$ and hence $(\Gamma, \Delta[X := A])(Y) = J \equiv J[X := A]$. Otherwise $(\Gamma, \Delta[X := A])(Y) = (\Delta[X := A])(Y) = J[X := A]$. In either case we conclude by K-VAR.

- *Case* ST-β_1. $\mathscr{J}$ is $(\lambda Y{:}J_1.B_1)\,B_2 \le B_1[X := B_2] : J_2[X := B_2]$ and we have $\Gamma, X{:}K, \Delta, Y{:}J_1 \vdash B_1 : J_2$ and $\Gamma, X{:}K, \Delta \vdash B_2 : J_1$. By the IH we get

$$\begin{array}{lll}\Gamma, \Delta[X := A], Y{:}J_1[X := A] & \vdash B_1[X := A] : J_2[X := A] & \text{and} \\ \Gamma, \Delta[X := A] & \vdash B_2[X := A] : J_1[X := A]. & \end{array}$$

Applying ST-β_1, we obtain

$$\begin{array}{llll}\Gamma, \Delta[X := A] & \vdash & ((\lambda Y{:}J_1.B_1)\,B_2)[X := A] & \le \; B_1[X := A][Y := B_2[X := A]] \\ & & & : \; K[X := A][Y := B_2[X := A]]\end{array}$$

and using Lemma 3.12 twice, it follows that

$$\Gamma, \Delta[X := A] \vdash ((\lambda Y{:}J_1.B_1)\,B_2)[X := A] \le B_1[Y := B_2][X := A] : K[Y := B_2][X := A]$$

which concludes the case. □

3.3.2 Admissible order-theoretic rules

The rules ST-Refl and ST-Trans establish that subtyping is a preorder. Via ST-AntiSym, we can lift these properties to type equality, and show that the latter is symmetric. Together, these properties make type equality an equivalence relation.

Corollary 3.13. *Type equality is an equivalence, i.e. the following equality rules are admissible.*

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash A = A : K}\;(\text{TEq-Refl}) \qquad \frac{\Gamma \vdash A = B : K}{\Gamma \vdash B = A : K}\;(\text{TEq-Sym})$$

$$\frac{\Gamma \vdash A = B : K \qquad B = C : K}{\Gamma \vdash A = C : K}\;(\text{TEq-Trans})$$

The same is true of kind equality, though we first have to establish that subkinding is a preorder.

Lemma 3.14. *Subkinding is a preorder, i.e. the following subkinding rules are admissible.*

$$\frac{\Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash K \le K}\;(\text{KS-Refl}) \qquad \frac{\Gamma \vdash K \le J \qquad J \le L}{\Gamma \vdash K \le L}\;(\text{KS-Trans})$$

Proof. Separately for each rule, by structural induction on K for KS-Refl and on J for KS-Trans. For the type interval cases we use the corresponding order-theoretic properties of subtyping. The proof of transitivity uses context narrowing (Lemma 3.10) and context validity (Lemma 3.6) for subkinding in the case where $J = (X{:}J_1) \to J_2$. □

Corollary 3.15. *Kind equality is an equivalence, i.e. the following equality rules are admissible.*

$$\frac{\Gamma \vdash K \ \mathsf{kd}}{\Gamma \vdash K = K}\ (\text{KEQ-REFL}) \qquad \frac{\Gamma \vdash K = J \qquad J = L}{\Gamma \vdash K = L}\ (\text{KEQ-TRANS}) \qquad \frac{\Gamma \vdash K = J}{\Gamma \vdash J = K}\ (\text{KEQ-SYM})$$

In addition, the following variants of subtyping and subkinding reflexivity are also admissible, which makes the subtyping and subkinding relations partial orders w.r.t. type and kind equality.

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

Another consequence of these rules is that we can treat well-typed terms and well-kinded types up to type and kind equality, respectively.

Corollary 3.16 (conversion)**.** *The following are admissible.*

$$\frac{\Gamma \vdash A : J \qquad \Gamma \vdash J = K}{\Gamma \vdash A : K}\ (\text{K-CONV}) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B : *}{\Gamma \vdash t : B}\ (\text{T-CONV})$$

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

In light of their order-theoretic properties, we often call kind and type equality judgments *equations* and subkinding and subtyping judgments *inequations*. We sometimes use equational reasoning notation in proofs, i.e. we write

$$\Gamma \vdash A_1 = A_2 = \cdots = A_i \leq A_{i+1} \leq \cdots = A_n \equiv A_{n+1} \equiv \cdots = A_m : K$$

to denote chains of (in)equations where the use of the corresponding transitivity rules is left implicit. In so doing, we may freely mix the relations $\leq$, $=$ and $\equiv$ provided that they are defined on the same sort (i.e. kinds or types). Such chains are always interpreted as judgments of the weakest relation they contain.

3.4 Validity

In this section, we state and prove a number of *validity properties* for the various judgments defined earlier. Roughly, we say that a judgment is valid if all its parts are well-formed. For example, *subkinding validity* states that, if $\Gamma \vdash J \leq K$, then both J and K are actually well-formed kinds. We saw another example earlier: *context validity* (Lemma 3.6) states that the context Γ of any judgment $\Gamma \vdash \mathscr{J}$ is well-formed. Here is a summary of the validity properties that remain to be proven.

Lemma 3.17 (validity)**.** *The judgments defined in Figures 3.5 and 3.6 enjoy the following validity properties.*

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

These validity properties provide a "sanity check" for the static semantics developed in this chapter, but they also play a crucial role in the proofs of other important properties, such as subject reduction, soundness of type normalization and and type safety.

Unfortunately, the validity properties are 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 3.18 (functionality)**.** *Let* $\Gamma \vdash A_1 = A_2 : K$.

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

But a naive attempt at proving this lemma directly leads to a circular dependency on kinding and subtyping validity, in a way that is not easily resolved. In particular, it is not sufficient to simply prove the two statements simultaneously.

It is instructive to play through the critical cases encountered when attempting to prove Lemmas 3.17 and 3.18 directly to see where things go wrong and to better understand the solution described in the next section. We start with subtyping validity, attempting a proof by induction on subtyping derivations. For the case of the application rule ST-App, we are given $\Gamma \vdash A_1 \leq A_2 : (X{:}J) \to K$ and $\Gamma \vdash B_1 = B_2 : J$, and we would like to show that $\Gamma \vdash A_1\, B_1 : K[X := B_1]$ and $\Gamma \vdash A_2\, B_2 : K[X := B_1]$. We can already spot the source of trouble: the type B_1 that is being substituted for X in the kind of the second type application differs from the argument type B_2. By the IH, we get $\Gamma \vdash A_2 : (X{:}J) \to K$ and $\Gamma \vdash B_2 : J$, and applying K-App we obtain $\Gamma \vdash A_2\, B_2 : K[X := B_2]$ but, as expected, the kinds do not match up. If we could show that $\Gamma \vdash K[X := B_2] = K[X := B_1]$, then by K-Conv, we would be done. Enter functionality of kind formation.

For the functionality lemma, we attempt a proof by simultaneous induction on kind formation and kinding derivations and consider the case where the current kinding derivation ends in an instance of the operator abstraction rule K-Abs. In addition to the premise $\Gamma \vdash B_1 = B_2 : J$, we are given derivations for $\Gamma, X{:}J \vdash K_1$ kd and $\Gamma, X{:}J, Y{:}K_1 \vdash A : K_2$, and we want to show that

$$\Gamma \vdash (\lambda Y{:}K_1.\, A)[X := B_1] \;=\; (\lambda Y{:}K_1.\, A)[X := B_2] \;:\; ((Y{:}K_1) \to K_2)[X := B_1].$$

To do so, we would like to use the rule ST-ABS together with ST-ANTISYM but we first need to establish the right-hand validity of the above equation, i.e. that

$$\Gamma \vdash (\lambda Y{:}K_1.\, A)[X := B_2] : ((Y{:}K_1) \to K_2)[X := B_1].$$

Clearly, Lemma 3.17 would be helpful here: equation validity would give us $\Gamma \vdash B_2 : J$, from which we could obtain $\Gamma \vdash (\lambda Y{:}K_1.\, A)[X := B_2] : ((Y{:}K_1) \to K_2)[X := B_2]$ by Lemma 3.9. This is almost what we need. Again, we face a mismatch in kinds that could, in principle, be remedied by using functionality of kind formation together with K-CONV. Concretely, we would like to invoke the IH to derive $\Gamma \vdash ((Y{:}K_1) \to K_2)[X := B_1] = ((Y{:}K_1) \to K_2)[X := B_2]$. Alas, we do not have a suitable sub-derivation to do so. Although $\Gamma \vdash (Y{:}K_1) \to K_2$ kd follows from kinding validity, we cannot not apply the IH to this result because it is not a sub-derivation of our overall premise. Indeed, none of the sub-derivations we are given are sufficient to derive the required kind equation.

Note that we could finish the proof of this case if only (1) the rule ST-ABS had an additional premise $\Gamma, X{:}J, Y{:}K_1 \vdash K_2$ kd and (2) the IH was a bit stronger, so that we could use it to derive

$$\Gamma, Y{:}K_1[X := B_1] \vdash K_2[X := B_1] = K_2[X := B_2].$$

This, together with a similar use of the IH on the first premise of ST-ABS and some uses of Lemma 3.9, ST-REFL-TEQ and SK-DARR, would be enough to derive $\Gamma \vdash ((Y{:}K_1) \to K_2)[X := B_2] \leq ((Y{:}K_1) \to K_2)[X := B_1]$, which we could then put to use with K-SUB. Indeed, these are the basic ideas that will allow us to resolve the circular dependency between Lemma 3.17 and Lemma 3.18. We start by addressing point (1).

3.4.1 The extended system

We define a pair of *extended* kinding and subtyping judgments in Fig. 3.7 where some rules have been endowed with additional premises. Only the rules that differ from those defined previously are mentioned in the figure, and the additional premises are highlighted in gray. We call these extra premises *validity conditions*. Crucially, the validity conditions of an extended rule are redundant in the sense that they follow (more or less) directly from the remaining premises of the rule via Lemma 3.17. For example, the extended rule K-EXTABS carries the extra premise $\Gamma, X{:}J \vdash K$ kd which follows directly from applying kinding validity to the rule's second premise $\Gamma, X{:}J \vdash A : K$. Thanks to this invariant, the two sets of rules are in fact equivalent – every derivation of an extended kinding or subtyping judgment has a corresponding derivation that uses only original rules, and vice-versa. We give a formal equivalence proof in Section 3.4.2.

Since kinding and subkinding are defined mutually with all the other judgments of the declarative system (except typing), the extension indirectly affects those judgments as well. We call the entire set of extended judgments the *extended (declarative) system*, as opposed to the

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

$$\frac{\Gamma \vdash J \ \mathsf{kd} \qquad \Gamma, X{:}J \vdash A : K \qquad \Gamma, X{:}J \vdash K \ \mathsf{kd}}{\Gamma \vdash \lambda X{:}J.\, A : (X{:}J) \to K} \ \text{(K-ExtAbs)}$$

$$\frac{\Gamma \vdash A : (X{:}J) \to K \qquad \Gamma \vdash B : J \qquad \Gamma, X{:}J \vdash K \ \mathsf{kd} \qquad \Gamma \vdash K[X := B] \ \mathsf{kd}}{\Gamma \vdash A\, B : K[X := B]} \ \text{(K-ExtApp)}$$

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

$$\frac{\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J \qquad \Gamma \vdash A[X := B] : K[X := B] \qquad \Gamma, X{:}J \vdash K \ \mathsf{kd} \qquad \Gamma \vdash K[X := B] \ \mathsf{kd}}{\Gamma \vdash (\lambda X{:}J.\, A)\, B \le A[X := B] : K[X := B]} \ \text{(ST-Ext}\beta_1\text{)}$$

$$\frac{\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J \qquad \Gamma \vdash A[X := B] : K[X := B] \qquad \Gamma, X{:}J \vdash K \ \mathsf{kd} \qquad \Gamma \vdash K[X := B] \ \mathsf{kd}}{\Gamma \vdash A[X := B] \le (\lambda X{:}J.\, A)\, B : K[X := B]} \ \text{(ST-Ext}\beta_2\text{)}$$

$$\frac{\Gamma \vdash A_1 \le A_2 : (X{:}J) \to K \qquad \Gamma \vdash B_1 = B_2 : J \qquad \Gamma \vdash B_1 : J \qquad \Gamma, X{:}J \vdash K \ \mathsf{kd} \qquad \Gamma \vdash K[X := B_1] \ \mathsf{kd}}{\Gamma \vdash A_1\, B_1 \le A_2\, B_2 : K[X := B_1]} \ \text{(ST-ExtApp)}$$

Figure 3.7 – Extended declarative kinding and subkinding

original (declarative) system. We will sometimes distinguish the two systems by writing $\Gamma \vdash_{\mathsf{d}} \mathscr{J}$ for judgments of the original system and $\Gamma \vdash_{\mathsf{e}} \mathscr{J}$ for those of the extended system. Since the two systems are equivalent, this distinction only matters in a few key situations – notably the development in the remainder of this section. When we refer to the "declarative system" in the following chapters, we will always mean the original declarative system, unless otherwise noted.

The idea of extending a set of inference rules with redundant premises in order to simplify metatheoretic proofs is not new. For example, Harper and Pfenning use similar premises to establish validity properties for the typing judgments of a variant of LF [30]. Furthermore, some readers will have noticed that a few of the original declarative rules already carry redundant premises. For example, the first premise $\Gamma \vdash J \ \mathsf{kd}$ of the rule K-Abs could easily be reconstructed from its second premise $\Gamma, X{:}J \vdash A : K$ via context validity (Lemma 3.6). The rules Wf-DArr, K-All, T-Abs, and T-TAbs carry similar validity conditions. We include these premises primarily because their presence simplifies the proof of the substitution lemma (Lemma 3.9). Context validity, on the other hand, remains easily provable without them.

To prove the validity properties stated in Lemma 3.17 for both the original and extended systems, we use the following strategy:

1. prove that the validity properties hold for the extended judgments;
2. prove that the two systems are equivalent, i.e. that

(a) the extended rules are *sound* w.r.t to the original ones – we can drop the validity conditions without affecting the conclusions of any derivations – and that

(b) the extended rules are *complete* w.r.t. to the original ones – the additional validity conditions follow from the remaining premises of the extended rules via the validity properties proved in step 1;

3. prove that the validity properties hold for the original system via the equivalence – convert original derivations to extended derivations (via completeness), derive the property in question, convert the conclusion back (via soundness).

Before we continue, we should point out that some of the validity conditions introduced in Fig. 3.7 are not actually necessary for the proof of Lemmas 3.17 and 3.18 – some even complicate the proofs. However, we will face a similar cyclic dependency later on when attempting to prove the equivalence of the (original) declarative system and the *canonical system* of judgments introduced in Chapter 5. Rather than introducing yet another extension to the declarative system later, we opt for a combined system containing all of the extra conditions.

We start the development set out above by noting that all the basic metatheoretic properties established in Section 3.3 still hold for the extended system. The proofs carry over with minor adjustments to deal with the additional premises. Next, we prove a variant of the functionality lemma discussed above but using the extended kinding and subtyping rules.

Lemma 3.19 (functionality – extended version)**.** *Substitutions of equal types in well-formed expressions result in well-formed equations. Let* Γ, Δ, Σ *be contexts,* K *a kind and* A_1, A_2 *types, such that* $\Gamma \vdash A_1 : K$ *and* $\Gamma \vdash A_2 : K$*, the context* Γ, Σ *is well-formed and the following equations hold:*

$$\Gamma \vdash A_1 \;=\; A_2 : K \qquad\qquad \Gamma, \Sigma \;=\; \Gamma, \Delta[X := A_1]\ \mathsf{ctx} \qquad\qquad \Gamma, \Sigma \;=\; \Gamma, \Delta[X := A_2]\ \mathsf{ctx}.$$

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

Note that the lemma has been strengthened – so that it is applicable to any type variable binding in a context, not just the last one – and simultaneously weakened – by adding extra conditions on A_1, A_2 and the target context Γ, Σ. The latter are effectively validity conditions ensuring that the proof of the lemma does not depend on Lemma 3.17. The separate target context Σ is used to symmetrize the treatment of context extensions, which is helpful when dealing with kind annotations in contravariant positions.

Proof. The two parts are proven simultaneously, by induction on extended kind formation and kinding derivations, respectively. The proof of the first part is relatively straightforward, while the proof of the second part deserves some attention. We present a few key cases, the others are similar.

- *Case* K-VAR. We have $B = Y$, $J = \Gamma(Y)$ and $\Gamma, X{:}K, \Delta$ ctx. We distinguish two cases based on Y.
  - *Sub-case* $Y = X$. We have $J = K \equiv K[X := A_1]$ since $X \notin \mathsf{fv}(K)$. By iterated weakening, we get $\Gamma, \Sigma \vdash A_1 = A_2 : K[X := A_1]$ and we are done.
  - *Sub-case* $Y \neq X$. We have $(\Gamma, \Delta[X := A_1])(Y) \equiv J[X := A_1]$, either because $Y \in \mathsf{dom}(\Gamma)$ and $X \notin \mathsf{fv}(J)$, or because $Y \in \mathsf{dom}(\Delta)$ and $(\Delta[X := A_1])(Y) = J[X := A_1]$. Furthermore, since $\Gamma, \Sigma = \Gamma, \Delta[X := A_1]$ ctx we have

    $$\Gamma, \Sigma \vdash (\Gamma, \Sigma)(Y) = (\Gamma, \Delta[X := A_1])(Y) \equiv J[X := A_1].$$

    We conclude by K-VAR, K-CONV, and TEQ-REFL.
- *Case* K-ALL. We have $B = \forall Y{:}J_1.B_1$ and $J = *$ for some kind J_1 and type B_1, as well as $\Gamma, X{:}K, \Delta \vdash J_1$ kd and $\Gamma, X{:}K, \Delta, Y{:}J_1 \vdash B_1 : *$. We want to show that $(\forall Y{:}J_1.B_1)[X := A_1]$ and $(\forall Y{:}J_1.B_1)[X := A_2]$ are mutual subtypes. To do so, we first prove that

  $$\Gamma, \Sigma \vdash (\forall Y{:}J_1.B_1)[X := A_1] : * \qquad \Gamma, \Sigma \vdash (\forall Y{:}J_1.B_1)[X := A_2] : *$$
  $$\Gamma, \Sigma \vdash J_1[X := A_2] \leq J_1[X := A_1] \qquad \Gamma, \Sigma \vdash J_1[X := A_1] \leq J_1[X := A_2]$$
  $$\Gamma, \Sigma, X{:}J_1[X := A_2] \vdash B_1[X := A_1] \leq B_1[X := A_2] : *$$
  $$\Gamma, \Sigma, X{:}J_1[X := A_1] \vdash B_1[X := A_2] \leq B_1[X := A_1] : *$$

  then apply ST-ALL twice, and conclude with ST-ANTISYM. The two kinding judgments follow from the premises by the substitution lemma (Lemma 3.9), the two subkinding judgments by the IH. The last two subtyping judgments require some extra work.

  Note that the additional type variable bindings in the two judgments differ syntactically, so we will have to use the IH twice, with different target contexts. In each case we need to show that the kind of the additional binding is well-formed and equal to $J_1[X := A_1]$ and $J_1[X := A_2]$, respectively. Concretely, we need to show that

  $$\Gamma, \Sigma \vdash J_1[X := A_2] = J_1[X := A_1] \text{ kd} \qquad \Gamma, \Sigma \vdash J_1[X := A_2] \text{ kd}$$
  $$\Gamma, \Sigma \vdash J_1[X := A_2] = J_1[X := A_2] \text{ kd}$$

  for the first invocation of the IH, and three analogous statements for the second. The first equation follows from the two subkinding judgments above via ST-ANTISYM. The context formation judgment and the first equation follow from the substitution lemma and TEQ-REFL. This is sufficient to apply the IH and obtain the first of the two remaining subtyping judgments via ST-REFL-TEQ. The proof of the second one is similar.

- *Case* K-EXTAPP. We have $B = B_1\,B_2$ and $J = J_2[Y := B_2]$ for some B_1, B_2, J_2, as well as

$$\begin{array}{ll}
\Gamma \vdash A_1 : K, & \Gamma \vdash A_2 : K \\
\Gamma \vdash A_1 = A_2 : K, & \Gamma, \Sigma\ \mathsf{ctx} \\
\Gamma, \Sigma = \Gamma, \Delta[X := A_1]\ \mathsf{ctx}, & \Gamma, \Sigma = \Gamma, \Delta[X := A_2]\ \mathsf{ctx}, \\
\Gamma, X{:}K, \Delta \vdash B_1 : (Y{:}J_1) \to J_2, & \Gamma, X{:}K, \Delta \vdash B_2 : J_1, \\
\Gamma, X{:}K, \Delta, Y{:}J_1 \vdash J_2\ \mathsf{kd}, & \Gamma, X{:}K, \Delta \vdash J_2[Y := B_2]\ \mathsf{kd}
\end{array}$$

for some J_1. We want to establish that that $(B_1\,B_2)[X := A_1]$ and $(B_1\,B_2)[X := A_2]$ are mutual subtypes in $J_2[Y := B_2][X := A_1]$, i.e. that

$$\Gamma, \Sigma \vdash (B_1\,B_2)[X := A_1] \leq (B_1\,B_2)[X := A_2] : J_2[Y := B_2][X := A_1], \quad \text{and} \tag{3.1}$$

$$\Gamma, \Sigma \vdash (B_1\,B_2)[X := A_2] \leq (B_1\,B_2)[X := A_1] : J_2[Y := B_2][X := A_1]. \tag{3.2}$$

The first half is fairly straightforward. Applying the IH to the first two premises of K-EXTAPP yields corresponding equations, the first of which we turn into an inequation via ST-REFL-TEQ.

$$\begin{array}{l}
\Gamma, \Sigma \vdash B_1[X := A_1] \leq B_1[X := A_2] : ((Y{:}J_1) \to J_2)[X := A_1], \\
\Gamma, \Sigma \vdash B_2[X := A_1] = B_2[X := A_2] : J_1[X := A_1].
\end{array}$$

In order to apply ST-EXTAPP we also need to derive the following validity conditions:

$$\begin{array}{ll}
\Gamma, \Sigma \vdash B_2[X := A_1] : J_1[X := A_1], & \Gamma, \Sigma, Y{:}J_1[X := A_1] \vdash J_2[X := A_1]\ \mathsf{kd}, \\
\Gamma, \Sigma \vdash J_2[X := A_1][Y := B_2[X := A_1]]\ \mathsf{kd}. &
\end{array}$$

All three follow from premises of K-EXTAPP and the substitution lemma (Lemma 3.9), followed by a use of Corollary 3.11 to adjust the contexts. Adjusting the context of $\Gamma, \Delta[X := A_1], Y{:}J_1[X := A_1] \vdash J_2[X := A_1]\ \mathsf{kd}$, requires a bit more work because we need to prove that the kind of the extra binding $Y{:}J_1[X := A_1]$ is well-formed. To do so, we first invoke context validity (Lemma 3.6) on the second validity condition of K-EXTAPP, which gives us $\Gamma, X{:}K, \Delta \vdash J_1\ \mathsf{kd}$. Form this, we derive the desired well-formedness proof via the substitution lemma. By ST-EXTAPP and Lemma 3.12, we arrive at (3.1).

We have to work a bit harder to prove (3.2). Again, we want to apply ST-EXTAPP, and again, the first two premises follow from the IH – this time followed by a use of TEQ-SYM to adjust the direction – and ST-REFL-TEQ to turn the first equation into a subtyping statement. The validity conditions are

$$\begin{array}{ll}
\Gamma, \Sigma \vdash B_2[X := A_2] : J_1[X := A_1], & \Gamma, \Sigma, Y{:}J_1[X := A_1] \vdash J_2[X := A_1]\ \mathsf{kd}, \\
\Gamma, \Sigma \vdash J_2[X := A_1][Y := B_2[X := A_2]]\ \mathsf{kd}. &
\end{array}$$

We have already established the second condition; the third one follows from applying

the substitution lemma to the first two. So it remains to prove the first.

We start by deriving $\Gamma, \Sigma \vdash B_2[X := A_2] : J_1[X := A_2]$ via the substitution lemma and context conversion. Next, we would like to use the IH to show that $\Gamma, \Sigma \vdash J_1[X := A_2] = J_1[X := A_1]$ in order to adjust the kind of the previous judgment via K-CONV. But to do so, we need to find a derivation of $\Gamma, X{:}K, \Delta \vdash J_1$ kd that is a strict sub-derivation of our current instance of K-EXTAPP.

Fortunately, this is always possible, thanks to the validity condition $\Gamma, X{:}K, \Delta, Y{:}J_1 \vdash J_2$ kd. Since the contexts of kind formation judgments are always well-formed themselves (see Lemma 3.6), it suffices to traverse the derivation tree of this judgment upwards along kind formation and kinding rules until one arrives at a "leaf" – an instance of K-VAR, K-TOP or K-BOT – which holds a well-formedness derivation for the current context. That context formation derivation, in turn, contains a sub-derivation of the desired kind formation judgment. Readers who are skeptical of this somewhat informal argument are encouraged to state and prove a helper lemma that combines the IH with the "lookup procedure" just described. The lemma is proven simultaneously with the main lemma, by induction on kind formation and kinding derivations.

With all the validity conditions in place, we apply ST-EXTAPP to obtain

$$\Gamma, \Sigma \vdash (B_1\, B_2)[X := A_2] \le (B_1\, B_2)[X := A_1] : J_2[X := A_1][Y := B_2[X := A_2]].$$

To complete the proof of 3.2, we use ST-CONV and

$$\begin{aligned}
\Gamma, \Sigma \vdash\ & J_2[X := A_1][Y := B_2[X := A_2]] && \\
=\ & J_2[X := A_2][Y := B_2[X := A_2]] && \text{(by Lemma 3.9)}\\
\equiv\ & J_2[Y := B_2][X := A_2] && \text{(by Lemma 3.12)}\\
=\ & J_2[Y := B_2][X := A_1] && \text{(by the IH)}
\end{aligned}$$

using our earlier result $\Gamma, \Sigma \vdash J_1[X := A_2] = J_1[X := A_1]$ in the first step and the final validity condition of K-EXTAPP in the last. □

We are now ready to prove Lemma 3.17 in the extended system, simultaneously with the following lemma.

Lemma 3.20. *Subtypes inhabiting interval kinds are proper subtypes. If* $\Gamma \vdash A \le B : C\,..\,D$*, then also* $\Gamma \vdash A \le B : *$.

Proof of Lemma 3.17 and Lemma 3.20 – extended version. All the validity properties are proven simultaneously with Lemma 3.20, by induction on the derivations of the respective premises. The proof is now mostly routine, thanks to the validity conditions. The only interesting cases are those of of ST-APP, where we use the functionality lemma to adjust the kind of the right-hand validity proof, and ST-INTV, where we use Lemma 3.20. The proof of Lemma 3.20 uses subtyping validity in turn. □

Corollary 3.21. *Equal types in intervals are equal as proper types. If* $\Gamma \vdash A = B : C \mathinner{..} D$*, then also* $\Gamma \vdash A = B : *$.

3.4.2 Equivalence

The next and final step in our program for proving Lemma 3.17 is to establish the equivalence of the two declarative systems.

Lemma 3.22. *The original and extended declarative systems are equivalent:* $\Gamma \vdash_{\mathsf{d}} \mathscr{J}$ *iff* $\Gamma \vdash_{\mathsf{e}} \mathscr{J}$.

Proof. The proof of soundness ($\Leftarrow$) is nearly trivial – we simply forget all the validity conditions. The proof of completeness ($\Rightarrow$) is only slightly more involved: the sub-derivations of the original rules are translated to the extended system recursively, then the missing premises – the validity conditions – are proved from the original premises using the extended version of Lemma 3.17 and, where necessary, the substitution lemma. □

Thanks to this equivalence, all the validity properties laid out in Lemma 3.17 also hold for the original judgments of the declarative system. Our original functionality lemma (Lemma 3.18) and the following strengthened version of Lemma 3.10 follow as corollaries of validity and Lemmas 3.19 and 3.10, respectively.

Corollary 3.23 (context narrowing – strong version)**.**

1. *If* $\Gamma \vdash A \leq B : *$ *and* $\Gamma, x{:}B, \Delta \vdash \mathscr{J}$*, then* $\Gamma, x{:}A, \Delta \vdash \mathscr{J}$.
2. *If* $\Gamma \vdash J \leq K$ *and* $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$*, then* $\Gamma, X{:}J, \Delta \vdash \mathscr{J}$,

3.5 Congruence lemmas for type and kind equality

Another consequence of the validity properties established in the previous section, is that we are now able to prove a number of admissible *congruence rules* for kind and type equality. These follow the same structure as the corresponding subkinding and subtyping rules but are generally a bit simpler. Firstly, we no longer need to pay attention to the variance (or polarity) of constructor arguments because equality is symmetric. Secondly, the left-hand validity conditions present in the rules SK-DArr and ST-All become redundant in the corresponding equality rules because the kind annotations in the left- and right-hand sides are convertible. Finally, thanks to symmetry, only one rule is needed for β-conversion, and likewise for η-conversion.

Lemma 3.24. *Kind equality is a congruence with respect to the interval and dependent arrow kind formers, i.e. the following kind equality rules are admissible.*

$$\frac{\Gamma \vdash A_1 = A_2 : * \qquad \Gamma \vdash B_1 = B_2 : *}{\Gamma \vdash A_1 \mathinner{..} B_1 = A_2 \mathinner{..} B_2}\ (\textsc{KEq-Intv})$$

$$\frac{\Gamma \vdash J_1 = J_2 \qquad \Gamma, X{:}J_1 \vdash K_1 = K_2}{\Gamma \vdash (X{:}J_1) \to K_1 = (X{:}J_2) \to K_2}\ (\textsc{KEq-DArr})$$

Lemma 3.25. *Type equality is a congruence with respect to the various type formers and includes β and η-conversion, i.e. the following type equality rules are admissible.*

$$\frac{\Gamma \vdash K_1 = K_2 \qquad \Gamma, X{:}K_1 \vdash A_1 = A_2 : *}{\Gamma \vdash \forall X{:}K_1.\, A_1 = \forall X{:}K_2.\, A_2 : *} \; (\text{TEq-All})$$

$$\frac{\Gamma \vdash A_1 = A_2 : * \qquad \Gamma \vdash B_1 = B_2 : *}{\Gamma \vdash A_1 \to B_1 = A_2 \to B_2 : *} \; (\text{TEq-Arr})$$

$$\frac{\Gamma, X{:}J \vdash A_1 = A_2 : K \quad \Gamma \vdash \lambda X{:}J_1.\, A_1 : (X{:}J) \to K \quad \Gamma \vdash \lambda X{:}J_2.\, A_2 : (X{:}J) \to K}{\Gamma \vdash \lambda X{:}J_1.\, A_1 = \lambda X{:}J_2.\, A_2 : (X{:}J) \to K} \; (\text{TEq-Abs})$$

$$\frac{\Gamma \vdash A_1 = A_2 : (X{:}J) \to K \qquad \Gamma \vdash B_1 = B_2 : J}{\Gamma \vdash A_1\, B_1 = A_2\, B_2 : K[X := B_1]} \; (\text{TEq-App})$$

$$\frac{\Gamma \vdash A_1 = A_2 : B\,..\,C}{\Gamma \vdash A_1 = A_2 : A_1\,..\,A_1} \; (\text{TEq-Sing})$$

$$\frac{\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J}{\Gamma \vdash (\lambda X{:}J.\, A)\, B = A[X := B] : K[X := B]} \; (\text{TEq-}\beta)$$

$$\frac{\Gamma \vdash A : (X{:}J) \to K \qquad X \notin \mathsf{fv}(A)}{\Gamma \vdash \lambda X{:}J.\, A\, X = A : (X{:}J) \to K} \; (\text{TEq-}\eta)$$

Proof. The admissibility proofs of the above rules all follow the same basic pattern. We want to show that the left- and right-hand sides of the conclusions are mutual subkinds or subtypes, respectively. To do so, we employ the respective subkinding and subtyping rules, adjusting the kinds of additional bindings and subtyping judgments using context narrowing (Corollary 3.23), subsumption ST-Sub, conversion ST-Conv and functionality (Lemma 3.18) where necessary. When additional validity properties are required, Lemma 3.17 delivers the required well-formedness or well-kindedness proofs. □

3.6 Admissible rules for higher-order extrema and intervals

In this section, we state and prove admissible rules that justify the encodings of higher-order extremal types and interval kinds given in Section 3.1.1. Many of these rules are straightforward generalizations of the corresponding rules for the types $\top$, $\bot$ and for proper type intervals $A\,..\,B$. The remaining rules and lemmas mostly deal with the family of kinds $*_K$, which plays a crucial role in the other encodings and the proofs of their respective properties.

We start by stating and proving a formation rule for $*_K$.

Lemma 3.26. *The kind $*_K$ is well-formed whenever K is, i.e. the following is admissible.*

$$\frac{\Gamma \vdash K \;\mathsf{kd}}{\Gamma \vdash *_K \;\mathsf{kd}} \; (\text{Wf-KMax})$$

Proof. By induction on the structure of K. The base case uses Wf-Star. □

The kind $*_K$ is a widened version of K, i.e. the latter is always a subkind of the former. As a consequence, any type of kind K is also of kind $*_K$.

Lemma 3.27. *Any well-formed kind K is a subkind of $*_K$.*

$$\frac{\Gamma \vdash K \ \mathsf{kd}}{\Gamma \vdash K \leq *_K} \ (\text{SK-KMAX})$$

Proof. By straightforward induction on the structure of K. □

Corollary 3.28. *If $\Gamma \vdash A : K$, then also $\Gamma \vdash A : *_K$.*

The following two lemmas introduce admissible kinding rules for the higher-order extremal types, and prove that $\bot_K$ and $\top_K$ are in fact extrema in $*_K$, i.e. they are the least and greatest inhabitants of $*_K$, respectively.

Lemma 3.29. *Higher-order extremal types are well-formed if their index kind is.*

$$\frac{\Gamma \vdash K \ \mathsf{kd}}{\Gamma \vdash \top_K : *_K} \ (\text{K-TMAX}) \qquad \frac{\Gamma \vdash K \ \mathsf{kd}}{\Gamma \vdash \bot_K : *_K} \ (\text{K-TMIN})$$

Proof. Separately, by induction on the structure of K. The cases for dependent arrow kinds use WF-KMAX. □

Lemma 3.30. *The types $\top_K$ and $\bot_K$ are the maximal and minimal elements of $*_K$, respectively.*

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash A \leq \top_K : *_K} \ (\text{ST-TMAX}) \qquad \frac{\Gamma \vdash A : K}{\Gamma \vdash \bot_K \leq A : *_K} \ (\text{ST-TMIN})$$

Proof. Separately, by induction on the structure of K. Corollary 3.28 is used to adjust the kind of the premises where necessary. In the inductive step, we use ST-ABS and the η-rules ST-$\eta_{1,2}$. For example, for $K = (X{:}K_1) \to K_2$ we have

$$\begin{aligned}
\Gamma \vdash A &\leq \lambda X{:}K_1.\, A\, X && \text{for } X \notin \mathsf{fv}(A) && \text{(by ST-}\eta_2\text{)}\\
&\leq \lambda X{:}K_1.\, \top_{K_2} && && \text{(by the IH and ST-ABS)}\\
&\equiv \top_{(X:K_1)\to K_2} \ : *_{(X:K_1)\to K_2}. && && \text{(by definition)}
\end{aligned}$$

□

Having generalized the properties of the extremal types to their higher-order counterparts, we now turn to interval kinds. We start with an admissible formation rule for higher-order intervals.

Lemma 3.31. *Higher-order interval kinds are well-formed if their bounds are.*

$$\frac{\Gamma \vdash A : K \qquad \Gamma \vdash B : K}{\Gamma \vdash A \mathrel{..}_K B \ \mathsf{kd}} \ (\text{WF-HOINTV})$$

Proof. By induction on the structure of K. The inductive step uses kinding validity, K-VAR and K-APP to expand the bounds. □

The subkinding rule SK-INTV for proper type intervals also generalizes straightforwardly to intervals over arbitrary type operators.

Lemma 3.32. *Higher-order interval kinds are widened in accordance with their bounds.*

$$\frac{\Gamma \vdash A_2 \leq A_1 : K \qquad \Gamma \vdash B_1 \leq B_2 : K}{\Gamma \vdash A_1 \mathbin{..}_K B_1 \leq A_2 \mathbin{..}_K B_2} \text{(SK-HOINTV)}$$

Proof. By induction on the structure of K. The inductive step uses subtyping validity, K-VAR, TEQ-REFL and ST-APP to expand the bounds, and WF-HOINTV to establish well-formedness of the left-hand side. □

Next, we would like to prove an admissible higher-order singleton introduction rule that generalizes K-SING. Ideally, we would like to show that any well-kinded type $\Gamma \vdash A : K$ inhabits its corresponding singleton kind $S(A : K)$. This is not necessarily true, however. Consider the case of an operator variable X with declared type $\Gamma(X) = * \to *$. The singleton kind corresponding to X is $S(X : * \to *) = (Y{:}*) \to X\,Y \mathbin{..} X\,Y$, so we would like to prove that $\Gamma \vdash X : (Y{:}*) \to X\,Y \mathbin{..} X\,Y$. Which kinding rules could we use to adjust the kind of X to the desired singleton kind? Since K-SING can only be applied to proper types, our only option is to use the subsumption rule K-SUB. But unfortunately, the declared kind $* \to *$ of X is a *strict supertype* of the singleton kind $(Y{:}*) \to X\,Y \mathbin{..} X\,Y$, so this cannot work.

We can, however, assign the desired singleton kind to the *η-expansion* of X, i.e. to $\lambda Y{:}*.\,X\,Y$. Unlike X, the application $X\,Y$ in the body of the η-expansion is a proper type, so we can use K-SING to narrow its kind. The full derivation is

$$\cfrac{\Gamma \vdash * \;\mathsf{kd} \qquad \cfrac{\cfrac{\Gamma, Y{:}* \vdash X : * \to * \qquad \Gamma, Y{:}* \vdash Y : *}{\Gamma, Y{:}* \vdash X\,Y : *}\text{(K-APP)}}{\Gamma, Y{:}* \vdash X\,Y : X\,Y \mathbin{..} X\,Y}\text{(K-SING)}}{\Gamma \vdash \lambda Y{:}*.\,X\,Y : (Y{:}*) \to X\,Y \mathbin{..} X\,Y}\text{(K-ABS)}$$

This principle generalizes to arbitrary well-kinded types: the η-expansion of a well-kinded type $\Gamma \vdash A : K$ always inhabits the corresponding singleton kind $S(A : K)$.

Given a type A, we define the *weak η-expansion* $\bar{\eta}_K(A)$ of A as $\bar{\eta}_{B \mathbin{..} C}(A) = A$ and $\bar{\eta}_{(X:J) \to K}(A) = \lambda X{:}J.\,\bar{\eta}_K(A\,X)$ where, as usual, we assume that $X \notin \mathsf{fv}(A)$. We call this expansion "weak" because the argument X in the definition $\lambda X{:}J.\,\bar{\eta}_K(A\,X)$ of the arrow case is not η-expanded further. This means that the result is not η-long. This is sufficient for the purpose of this section; we will define a stronger version in the next chapter.

As expected, a type of kind K is equal to its weak η-expansion in K.

Lemma 3.33. *Weak η-expansion is sound, i.e. if $\Gamma \vdash A : K$, then $\Gamma \vdash A = \bar{\eta}_K(A) : K$.*

Proof. By induction on the structure of K, using TEQ-η and TEQ-ABS in the inductive case. □

Lemma 3.34. *The η-expansions of type operators inhabit their higher-order singleton intervals.*

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash \bar{\eta}_K(A) : S(A : K)} \ (\text{K-HoSing})$$

Proof. By induction on the structure of K. The base case follows from K-Sing, the inductive step from the usual combination of kinding validity, K-Var, K-App and Wf-HoIntv. □

Corollary 3.35. *If $\Gamma \vdash B_1 \leq A : K$ and $\Gamma \vdash A \leq B_2 : K$, then $\Gamma \vdash \bar{\eta}_K(A) : B_1 \mathrel{..}_K B_2$.*

Having found ways to form, widen and populate higher-order intervals, we still need a way to put their bounds to use. To this end, we introduce two *higher-order bound projection* rules, which generalize the corresponding rules ST-Bnd$_1$ and ST-Bnd$_2$ for proper type intervals.

Lemma 3.36 (higher-order bound projection)**.** *Inhabitants of a higher-order interval are supertypes of its lower bound and subtypes of its upper bound.*

$$\frac{\Gamma \vdash A : B_1 \mathrel{..}_K B_2 \qquad \Gamma \vdash A : K \qquad \Gamma \vdash B_1 : K}{\Gamma \vdash B_1 \leq A : K} \ (\text{ST-HoBnd}_1) \qquad \frac{\Gamma \vdash A : B_1 \mathrel{..}_K B_2 \qquad \Gamma \vdash A : K \qquad \Gamma \vdash B_2 : K}{\Gamma \vdash A \leq B_2 : K} \ (\text{ST-HoBnd}_2)$$

These rules are a bit weaker than one might expect. In particular, the additional premises $\Gamma \vdash A : K$, $\Gamma \vdash B_1 : K$ and $\Gamma \vdash B_2 : K$, might seem redundant. They are necessary because we cannot, in general, invert well-formedness judgments about higher-order intervals. That is, $\Gamma \vdash B_1 \mathrel{..}_K B_2\ \mathsf{kd}$ does *not* imply $\Gamma \vdash B_1 : K$ and $\Gamma \vdash B_2 : K$, nor does $\Gamma \vdash A : B_1 \mathrel{..}_K B_2$ imply $\Gamma \vdash A : K$. To see this, consider the kind $K = \bot \mathrel{..}_\varnothing \top$, where $\varnothing = \top \mathrel{..} \bot$ is the empty interval (note the absurd bounds). The kind K is well-formed and inhabited by both $\bot$ and $\top$, yet clearly $\bot, \top$ are not inhabitants of $\varnothing$. Note that the formation rule Wf-HoIntv for higher-order intervals is not to blame: although K is well-formed, we cannot prove this fact using Wf-HoIntv. There are simply more well-formed higher-order intervals than can be derived using Wf-HoIntv.

Proof of Lemma 3.36. Separately, by induction on the structure of K. In the base case, we use the interval projection rules ST-Bnd$_{1,2}$ as well as ST-Intv and ST-Sub to adjust the kinds of the resulting inequations. In the inductive step, we use ST-Abs and the η-rules ST-$\eta_{1,2}$. For example, for the left-hand case and $K = (X{:}K_1) \to K_2$ we have

$$\begin{aligned}
\Gamma \vdash B_1 &\leq \lambda X{:}K_1.\, B_1\, X && \text{(by ST-}\eta_2\text{)}\\
&\leq \lambda X{:}K_1.\, A\, X && \text{(by the IH and ST-Abs)}\\
&\leq A \qquad : (X{:}K_1) \to K_2. && \text{(by ST-}\eta_1\text{)}
\end{aligned}$$

□

Thanks to the admissible kinding and subtyping rules for higher-order intervals and extrema, we can now easily derive judgments for forming, introducing or eliminating bounded universal quantifiers over arbitrary type operators.

For example, well-formedness of the higher-order universal quantifier $\forall X \le A{:}K.\,B$ can be derived as

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{\dfrac{\Gamma \vdash A : K}{\Gamma \vdash K\ \mathsf{kd}}\ (\text{kinding validity})}{\Gamma \vdash \bot_K : *_K}\ (\text{K-TMin})
      \qquad
      \dfrac{\Gamma \vdash A : K}{\Gamma \vdash A : *_K}\ (\text{Corollary 3.28})
    }{\Gamma \vdash \bot_K \mathrel{..}_{*_K} A\ \mathsf{kd}}\ (\text{Wf-HoIntv})
  }{\Gamma \vdash \bot_K \mathrel{..}_K A\ \mathsf{kd}}\ (\text{Lemma 3.37})
  \qquad
  \Gamma, X{:}\bot_K \mathrel{..}_K A \vdash B : *
}{\Gamma \vdash \forall X \le A{:}K.\,B : *}\ (\text{K-All})
$$

The derivation uses the following lemma for simplifying interval kinds; its proof is by structural induction on the index K.

Lemma 3.37. *Let A, B be types and K a kind. Then $A \mathrel{..}_{*_K} B \equiv A \mathrel{..}_K B$*

Similar derivations exist for the introduction and elimination rules.

Corollary 3.38 (bounded quantification)**.** *The following rules for the formation, introduction and elimination of bounded universal quantifiers are admissible.*

$$
\dfrac{\Gamma \vdash A : K \qquad \Gamma, X{:}\bot_K \mathrel{..}_K A \vdash B : *}{\Gamma \vdash \forall X \le A{:}K.\,B : *}\ (\text{K-AllBnd})
$$

$$
\dfrac{\Gamma \vdash A : K \qquad \Gamma, X{:}\bot_K \mathrel{..}_K A \vdash t : B}{\Gamma \vdash \lambda X \le A{:}K.\,t : \forall X \le A{:}K.\,B}\ (\text{T-TAbsBnd})
$$

$$
\dfrac{\Gamma \vdash t : \forall X \le A{:}K.\,B \qquad \Gamma \vdash C \le A : K}{\Gamma \vdash t\,(\bar{\eta}_K(C)) : B[X := \bar{\eta}_K(C)]}\ (\text{T-TAppBnd})
$$

Similar rules for the formation, abstraction and elimination of bounded operators are also admissible.

Note that we need to η-expand the type argument C in the elimination rule T-TAppBnd before it can by applied to t. This is because C has kind K, while the polymorphic expression t expects an argument of kind $\bot_K \mathrel{..}_K C$. As discussed earlier, C is not guaranteed to inhabit that kind but its η-expansion is – via K-HoSing and a subsequent widening of its kind from $C \mathrel{..}_K C$ to $\bot_K \mathrel{..}_K C$.

There is an alternative encoding of higher-order bounded quantification (and bounded type operators) that separates the declaration of type variables from that of the subtyping constraints imposed by their bounds, at the cost of using an auxiliary type variable with potentially inconsistent bounds. Assume a partition of the set of type variable names into two distinct sets of *operator names* denoted by $X_{\mathsf{n}}, Y_{\mathsf{n}}, \ldots$ and *constraint names* denoted by $X_{\mathsf{c}}, Y_{\mathsf{c}}, \ldots$ We may then encode an upper-bounded type variable binding $X \le A : K$ as a pair of bindings

$X_{\mathsf{n}} : K,\ X_{\mathsf{c}} : X_{\mathsf{n}} \mathrel{..}_K A$, separating the declaration of the operator name X from the subtyping constraint $X \leq A$. For example, the encoding of bounded universal quantifiers according to this scheme would be $\forall X \leq A{:}K.\, B = \forall X_{\mathsf{n}}{:}K.\, \forall X_{\mathsf{c}}{:}X_{\mathsf{n}} \mathrel{..}_K A.\, B$ where $X_{\mathsf{c}} \notin \mathsf{fv}(B)$.

The advantage of this encoding is a cleaner separation between the uses of bounded variable bindings in kinding and subtyping. Whenever we want to refer to the original type variable X or its kind, we simply use X_{n}. When we require a proof of the fact that $X \leq A$ we obtain one from X_{c} via ST-HoBind$_1$, ST-HoBind$_2$, and ST-Trans. The same is true when we instantiate type parameters. For example, a type application $t\,C$, where t has type $\forall X \leq A{:}K.\, B$ and $C \leq A$, is now desugared to $t\,C\,(\bar{\eta}_K(C))$, i.e. only the second type argument, which corresponds to the constraint parameter X_{c}, needs to be η-expanded, while the argument C for the parameter X_{n} can be left as is. Since X_{c} does not occur freely in the codomain B of the desugared universal type, the overall type of the desugared application is just $B[X_{\mathsf{n}} := C]$. A clear drawback of this encoding is the necessary duplication of bindings and the corresponding introduction and elimination forms (abstraction, application). In addition, the kind $X_{\mathsf{n}} \mathrel{..}_K A$ of the constraint X_{c} has inconsistent bounds in general, which can be problematic.

In Section 3.1.1, we mentioned an alternative definition for the family of kinds $*_K$, namely $*_K = (\bot_K) \mathrel{..}_K (\top_K)$. The original definition, given in Fig. 3.2, has the advantage of being independent of the definition of the higher-order extrema $\top_K$ and $\bot_K$. This allowed us to prove properties such as Wf-KMax and Lemma 3.27 admissible without appealing to any of the properties of higher-order extrema, and thereby avoid some cyclic dependencies in the proofs of the latter. The alternative definition, on the other hand, seems more intuitive. To conclude the section, we show that the two definitions are equal for well-formed kinds K.

Lemma 3.39. *The kind $*_K$ is equal to the higher-order interval bounded by $\bot_K$ and $\top_K$. If $\Gamma \vdash K\ \mathsf{kd}$, then $\Gamma \vdash *_K = (\bot_K) \mathrel{..}_K (\top_K)$.*

Proof of Lemma 3.39. By induction on the structure of K. The base case is immediate. In the inductive step, we use SK-HoIntv and the β-rule TEq-β.

Let $K = (X{:}K_1) \to K_2$. We want to show that

$$\Gamma \vdash (X{:}K_1) \to *_{K_2} \;=\; (Y{:}K_1) \to (\bot_{(X:K_1)\to K_2}\, Y) \mathrel{..}_{K_2} (\top_{(X:K_1)\to K_2}\, Y)$$

for some Y that does not occur freely in $\mathsf{fv}(\bot_{(X:K_1)\to K_2})$ or $\mathsf{fv}(\top_{(X:K_1)\to K_2})$. By the IH, we have $\Gamma \vdash *_{K_2} = (\bot_{K_2}) \mathrel{..}_{K_2} (\top_{K_2})$ but we need to adjust the bounds of the right-hand interval. We use the following equation for the lower bound, and an similar one for the upper bound.

$$\begin{aligned}
\Gamma \vdash \bot_{(X:K_1)\to K_2}\, Y \;&\equiv\; \lambda X{:}K_1.\, \bot_{K_2}\, Y && \text{(by definition)}\\
&=\; \bot_{K_2}[X := Y] && \text{(by K-TMin and TEq-}\beta\text{)}\\
&\equiv\; \bot_{K_2} \qquad : *_{K_2}. && (\alpha\text{-renaming})
\end{aligned}$$

By SK-REFL-KEQ, SK-HOINTV and SK-ANTISYM we obtain

$$\Gamma \vdash (\bot_{K_2}) \mathrel{..}_{*_{K_2}} (\top_{K_2}) \;=\; (\bot_{(X:K_1)\to K_2}\, Y) \mathrel{..}_{*_{K_2}} (\top_{(X:K_1)\to K_2}\, Y)$$

which we re-index using Lemma 3.37. We conclude by KEQ-REFL and KEQ-DARR. □

3.7 Subject reduction for well-kinded types

For most versions of F_ω, subject reduction is easy to prove at the type level because their kind languages are essentially the same as the type language of the simply-typed lambda calculus. In $F^\omega_{..}$, the proof is complicated by the presence of type-dependent kinds and subkinding. However these complications are minor. Despite type dependency, the structure of kinds is still rather simple, as witnessed by the subkinding rules. There are only two *shapes* of kinds – intervals and arrows – with exactly one subkinding rule per shape: SK-INTV for relating intervals and SK-DARR for relating arrows. There is no danger of relating kinds of different shapes simply because there are no subkinding rules to do so. Contrast this with subtyping, where we can, in principle, relate proper types of any shape using the bound projection rules ST-BND$_{1,2}$ and transitivity.

Thanks to the simple structure of subkinding, it is easy to prove the following generation lemma.

Lemma 3.40 (generation of kinding for operator abstractions)**.** *The following is admissible.*

$$\frac{\Gamma \vdash \lambda X{:}L.\, A : (X{:}J) \to K}{\Gamma, X{:}J \vdash A : K}$$

Proof. By induction on kinding derivations. There are only two relevant cases, K-ABS and K-SUB. The former is immediate, the latter proceeds by case analysis on subkinding derivations. We have $\Gamma \vdash \lambda X{:}L.\, A : L'$ and $\Gamma \vdash L' \leq (X{:}J) \to K$ for some L'. The subkinding judgment must have been derived using SK-DARR since that is the only rule relating arrow kinds. Hence $L' = (X{:}J') \to K'$ for some kinds J', K' such that $\Gamma \vdash J \leq J'$ and $\Gamma, X{:}J \vdash K' \leq K$. By the IH, we get $\Gamma, X{:}J' \vdash A : K'$. The result follows by context narrowing (Corollary 3.23) and K-SUB. □

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

Lemma 3.41.

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

The proof uses a weak congruence rule TEQ-ABSWEAK for operator abstractions analogous to the weak subtyping rule ST-ABSWEAK discussed in Section 3.2, as well as the following variant of TEQ-β.

$$\frac{\Gamma \vdash J_1 = J_2 \qquad \Gamma, X{:}J_1 \vdash A_1 = A_2 : K}{\Gamma \vdash \lambda X{:}J_1.\, A_1 = \lambda X{:}J_2.\, A_2 : (X{:}J_1) \to K} \text{(TEQ-ABSWEAK)}$$

$$\frac{\Gamma, X{:}J \vdash A : K \qquad \Gamma \vdash B : J \qquad \Gamma \vdash \lambda X{:}L.\, A : (X{:}J) \to K}{\Gamma \vdash (\lambda X{:}L.\, A)\, B = A[X := B] : K[X := B]} \text{(TEQ-}\beta'\text{)}$$

Both rules are easily proven admissible using the validity conditions and congruence rules introduced in Section 3.4.

Proof of Lemma 3.41. The two parts are proven simultaneously, by induction on kind formation and kinding derivations, respectively, followed by a case-analysis on the last rule used to derive the β-step. Most cases are straightforward. They follow the general strategy of first applying the IH to the premise that corresponds to the sub-expression taking a reduction step, before concluding with the relevant congruence rule and KEQ-REFL or TEQ-REFL where necessary. In the case for K-ABS we use TEQ-ABSWEAK. The only interesting case is that for K-APP when the reduction step is a β-contraction. There we use the generation lemma for operator abstraction (Lemma 3.40) and the rule TEQ-β' discussed above. □

Subject reduction for kinds and types follows now almost trivially from Lemma 3.41.

Theorem 3.42 (subject reduction for kinding)**.** *Full β-reduction preserves well-formedness of kinds and well-kindedness of types.*

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* $\Gamma \vdash B : K$.

Proof. By repeated application of Lemma 3.41 and equation validity. □

As we will see in the next section, things are not quite as simple at the term level.

3.8 Type safety

Following Wright and Felleisen's syntactic approach [53], we would like to establish the safety of our calculus by proving the two properties of *progress* and *preservation* (aka subject reduction).

Proposition 3.1 (type safety)**.** *Well-typed terms do not get stuck.*

(progress) *If* $\vdash t : A$*, then either* $t = v$ *for some value* v*, or* $t \longrightarrow_{\mathsf{v}} t'$ *for some term* t'.
(preservation) *If* $\Gamma \vdash t : A$ *and* $t \longrightarrow_{\mathsf{v}} t'$*, then* $\Gamma \vdash t' : A$.

Unfortunately, preservation for well-typed terms does not hold in its usual form in $F^\omega_{::}$. Note that unlike the progress property, which only holds in the empty context, preservation is expected to hold in arbitrary contexts, i.e. for *open terms*. Indeed, this is the convention we adopted for the type-level subject reduction theorem (Theorem 3.42) in the previous section. But reduction of open terms is not safe in $F^\omega_{::}$. The culprit are type variable bindings with inconsistent bounds.

Consider the following example. Assume v and A such that $\vdash v : A$ and $\vdash A \not\leq A \to B : *$ for any B, i.e. v is a closed value of type A that cannot be applied to other terms of type A. For example, take 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,[1] and hence $\vdash \forall X{:}*.\, X \to X \not\leq (\forall X{:}*.\, X \to X) \to B : *$. We will give a formal proof of this fact in Chapter 5 (see Lemma 5.30).

Since v cannot be applied to other As, the application $t = (\lambda x{:}A.\, x)\; v\; v$ is ill-typed as a closed term. The inner application $(\lambda x{:}A.\, x)\; v$ is just the identity function on A applied to v; its type is again A, which – by assumption – is not a type that takes arguments of type A itself, hence we cannot apply it to v again. Yet, as the following derivation illustrates, we can show that t is well-typed in the context $\Gamma = X : (A \to A) \,..\, (A \to A \to A)$.

$$
\dfrac{
  \dfrac{
    \dfrac{\vdots}{\Gamma \vdash \lambda x{:}A.\, x : A \to A}
    \quad
    \dfrac{
      \dfrac{\dfrac{\vdots}{\Gamma \vdash X : (A \to A) \,..\, (A \to A \to A)}}{\Gamma \vdash A \to A \leq X : *}\;(\textsc{ST-Bnd}_1)
      \quad
      \dfrac{\dfrac{\vdots}{\Gamma \vdash X : (A \to A) \,..\, (A \to A \to A)}}{\Gamma \vdash X \leq A \to A \to A : *}\;(\textsc{ST-Bnd}_2)
    }{\Gamma \vdash A \to A \leq A \to A \to A : *}\;(\textsc{ST-Trans})
  }{\Gamma \vdash \lambda x{:}A.\, x : A \to A \to A}\;(\textsc{T-Sub})
  \quad
  \Gamma \vdash v : A
}{
  \dfrac{\Gamma \vdash (\lambda x{:}A.\, x)\; v : A \to A \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x{:}A.\, x)\; v\; v : A}\;(\textsc{T-App})
}\;(\textsc{T-App})
$$

Note that both $A \to A$ and $A \to A \to A$ are proper types, so the interval $(A \to A) \,..\, (A \to A \to A)$ is well-formed, as is the context Γ. But since $\vdash A \not\leq A \to A : *$, the bounds of the interval are absurd – assuming that subtyping is well-behaved at least for closed types.

To see how this example breaks preservation, consider what happens when t takes a reduction step. By R-App$_1$ and R-AppAbs we have

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

Since $\Gamma \vdash (\lambda x{:}A.\, x)\; v\; v : A$, we would expect the resulting term $v\; v$ to also have type A, but that is not the case. In fact, the application $v\; v$ is ill-typed, even in Γ. The assumption $X : (A \to A) \,..\, (A \to A \to A)$ is useless here, since v does not have type $A \to A$.

If we take $v = \lambda X{:}*.\, \lambda x{:}X.\, x$ as suggested earlier, then $v\; v$ is not only ill-typed, it is also stuck. Type abstractions expect type arguments, but v is a term, so it cannot be applied to itself. Hence $v\; v$ is neither a value nor can it be reduced further.

This example illustrates that it is not safe to evaluate open terms in $F^{\omega}_{\cdot\cdot}$. But what about closed terms? If we close off the term t from our example using a type abstraction, we obtain the value $\lambda X{:}(A \to A) \,..\, (A \to A \to A).\, (\lambda x{:}A.\, x)\; v\; v$. Because it is a value, there are no applicable

[1] There are systems where universal types are subtypes of their instantiations, i.e. $\forall X{:}K.\, A \leq A[X := B]$ for some B, but this is not the case for $F^{\omega}_{\cdot\cdot}$, where universals have to be eliminated explicitly using K-App. Other candidates for v and A would be $v = \lambda x{:}\bot.\, x$ with $A = \bot \to \bot$ or $A = \top$.

CBV reduction rules, and preservation holds trivially. And, assuming the bounds of the binding $X : (A \to A) \mathinner{..} (A \to A \to A)$ are indeed absurd, we cannot possibly supply a type argument to this polymorphic term either: if there were some closed type $\vdash C : (A \to A) \mathinner{..} (A \to A \to A)$, then we would have $\vdash A \to A \leq A \to A \to A : *$ by the bound projection rules and transitivity, and hence the interval would have consistent bounds after all. At least superficially, it looks like preservation might hold for closed terms.

Throughout the next two chapters, we will formalize this intuition and work our way towards a proof of the following weakened version of preservation.

Theorem 3.43 (preservation – weak version)**.** *CBV reduction preserves the types of closed terms. If* $\vdash t : A$ *and* $t \longrightarrow_{\mathsf{v}} t'$*, then* $\vdash t' : A$.

To conclude the chapter, let us briefly explore the challenges involved in proving the weak preservation theorem and some strategies to address them. Following the approach used to prove Lemma 3.41, we start by stating a generation lemma for term and type abstractions. Since the subtyping rules are more complex than those for subkinding, our generation lemma for typing is a bit weaker than the one for kinding. In particular, rather than yielding a single typing judgment, it concludes with a pair of typing and subtyping judgments.

Lemma 3.44 (generation of typing for term and type abstraction)**.**

1. *If* $\Gamma \vdash \lambda x{:}A.\, t : B$*, then* $\Gamma, x{:}A \vdash t : C$ *and* $\Gamma \vdash A \to C \leq B : *$ *for some* C.
2. *If* $\Gamma \vdash \lambda X{:}K.\, t : A$*, then* $\Gamma, X{:}K \vdash t : B$ *and* $\Gamma \vdash \forall X{:}K.\, B \leq A : *$ *for some* B.

Note that the lemma puts lower bounds on the types of the abstractions but does not say anything about their shapes. For example, we do not know if the type B of a term abstraction is actually an arrow type, only that it is lower-bounded by one.

Next, assume we are attempting a proof of the preservation theorem by induction on typing derivations and case analysis on CBV reduction rules. The interesting cases are again those where β-reductions occur. For example, consider the case of T-APP when the reduction step is an instance of R-APPABS. We have $t = (\lambda x{:}B.\, s)\; v$ with $\vdash \lambda x{:}B.\, s : C \to A$ and $\vdash v : C$ for some B and C. By generation, $x{:}B \vdash s : D$ and $\vdash B \to D \leq C \to A : *$ for some D. To continue, we would like to show that $\vdash C \leq B : *$ and $\vdash D \leq A : *$. As expected, this is where things get challenging.

The rules ST-ARR and ST-ALL tell us that the type formers for arrows and universals *preserve* the subtyping order – they are both antitone in their first argument and monotone in the second one. What we need to show now is that these constructors also *embed* the subtyping order, i.e. we would like to show that the following rules are admissible:

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

We will refer to these properties as *inversion of subtyping*, although, strictly speaking, it is not the subtyping relation that is being inverted but the rules for subtyping the arrow and universal type formers. There are several features of the subtyping rules that severely complicate the proof of subtyping inversion.

1. The presence of the bound projection rules ST-BND$_{1,2}$, together with transitivity, allow derivations of the form $A \leq X \leq B : *$ where A and B need not even be of the same shape. For example, under suitably absurd assumptions we can derive

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

2. 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.

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

3. The subsumption rule ST-SUB may change the kinds of related types in the middle of a subtyping derivation.

We address these points as follows. Firstly, we avoid issues caused by absurd bounds by proving subtyping inversion only in the empty context, i.e. only for closed types. That is sufficient for proving weak preservation and, as the above example illustrates, it is the best we can do, unless we fundamentally change the way abstractions over type variables with inconsistent bounds are introduced and eliminated. Secondly, we eliminate uses of the β and η rules by adopting an alternative presentation of subtyping which we dub *canonical subtyping*. Canonical subtyping judgments only relate types in *η-long β-normal form*, which means that the β or η rules cannot occur in a canonical subtyping derivation, otherwise at least one side of the corresponding inequation would not be in normal form. The canonical presentation of subtyping also deals with the last issue in the above list. In canonical subtyping derivations, subsumption is relegated to certain carefully chosen positions, similarly to the way subtyping is handled in algorithmic or bidirectional typing.

The core challenge of the proof of subtyping inversion will thus consist in establishing that the canonical and declarative presentation of subtyping are equivalent, and in particular that every well-formed kind and every well-kinded type has a normal form. This will be the topic of the next two chapters.

4 Normalization of types

The declarative subtyping rules described in the previous chapter contain a great deal of redundancy. One source of redundancy is the computational nature of types and kinds themselves. Subtyping relates $\beta\eta$-convertible types and kinds, which are semantically equal but differ in their syntactic structure. This impedes any direct, structural approach to establishing subtyping inversion, even for closed types.

In this chapter we take a first step towards a canonical presentation of subtyping by eliminating this source of redundancy. We show that types and kinds in $F^{\omega}_{..}$ are weakly normalizing and can thus be put into a canonical form: their $\beta\eta$-normal form. We define a *bottom-up normalization* procedure based on *hereditary substitution* and prove its soundness, i.e. that any given type is judgmentally equal to the normal form computed for it by this procedure.

The chapter is divided into two parts. Section 4.1 defines the hereditary substitution and normalization functions on raw, i.e. unkinded, types and establishes basic properties. Section 4.2 introduces a set of *simplified kinding* judgments which provide a syntactic characterization of normal types and allow us to establish important properties about hereditary substitutions and normal forms, notably a pair of commutativity lemmas (Lemmas 4.17 and 4.26) that play an important role in the development of the next chapter.

4.1 Normalization of raw types and kinds

In this section, we define a bottom-up normalization procedure on raw types and kinds. A key ingredient is *hereditary substitution* [52]. Roughly, hereditary substitution differs from ordinary substitution in that it immediately eliminates any β-redexes created when substituting an abstraction for a variable at the head of an application. Thanks to this strategy, hereditary substitution preserves well-kinded normal forms, as we will show in the second part of this chapter. The challenge in defining a hereditary substitution function adapted to our dependently kinded setting is, of course, ensuring its totality, i.e. that it terminates on all inputs.

$F, G ::=$	Head	$\boldsymbol{D}, \boldsymbol{E} ::=$	Spine
X	type variable	ϵ	empty spine
$\top$	top/maximum type	$D, \boldsymbol{E}$	concatenation
$\bot$	bottom/minimum type	$J, K, L ::=$	Kind
$D \to E$	function type	$D \,..\, E$	type interval
$\forall X{:}K.\,E$	universal type	$(X{:}J) \to K$	dependent operator kind
$\lambda X{:}K.\,E$	operator abstraction	$\gamma, \delta ::=$	Simple kinding context
$D, E ::= F\,\boldsymbol{E}$	Elimination	$\varnothing$	empty context
		$\gamma, X{:}k$	type variable binding

Figure 4.1 – Alternative syntax for types

Before we present our variant of hereditary substitution and the resulting normalization procedure for types and kinds, we first establish a few preliminaries.

4.1.1 Syntax

We begin by introducing an alternative syntax for types that is better suited to our definition of hereditary substitution. The key difference between this presentation of types and the the more standard one given in the previous chapter is that the arguments of repeated applications are grouped together in sequences called *spines*. The syntactic structure of spines closely matches that of the kinds of operators being applied to them, which is key to our presentation of hereditary substitution. Using this alternative syntax, every type is represented as an application (or elimination) of an operator to zero or more arguments. Hence we call this presentation of types *elimination form*. The grammar of elimination forms is given in Fig. 4.1.

Applications form a separate syntactic category, called *eliminations*, and are composed of a *head* followed by a sequence of arguments, called the *spine*. A head is any type form that is not an application. Intuitively, heads are atomic operators that may be applied to a sequence of type arguments to form compound types. This is true even for proper type constants such as $\top$ and $\bot$, which are considered nullary operators. Every head F has an associated elimination form $F\epsilon$ where it is applied to an empty spine. We often omit ϵ and informally treat heads as a subset of eliminations. Spines are just sequences of eliminations. We adopt vector notation for spines, writing $\boldsymbol{E}$ for the sequence $\boldsymbol{E} = E_1, E_2, \ldots, E_n$. We sometimes use vector notation for other sorts of expressions as well, e.g. $\boldsymbol{A}$ denotes a sequence of types. We write $(\boldsymbol{D}, \boldsymbol{E})$ for the concatenation of two sequences $\boldsymbol{D}$ and $\boldsymbol{E}$. The alternative representation extends to kinds as well, though we do not use separate notation to distinguish normal kinds or from those in elimination form. When the distinction is important, it is usually clear from the context.

The two representations of types are isomorphic. Given a type A, we write $\overline{A}$ for its elimination form. Conversely, we write $\underline{E}$, $\underline{\boldsymbol{E}}$ and $\underline{F}$ and for the type, or sequence of types, underlying an elimination form E, a spine $\boldsymbol{E}$ or a head F, respectively. More often though, we omit the

Weak equality of heads $\boxed{F \approx G}$

$$\frac{}{X \approx X}\ (\text{WEQ-VAR}) \qquad \frac{}{\top \approx \top}\ (\text{WEQ-TOP}) \qquad \frac{}{\bot \approx \bot}\ (\text{WEQ-BOT})$$

$$\frac{D_1 \approx D_1 \qquad E_1 \approx E_2}{D_1 \to E_1 \approx D_2 \to E_2}\ (\text{WEQ-ARR}) \qquad \frac{K_1 \approx K_2 \qquad E_1 \approx E_2}{\forall X{:}K_1.\, E_1 \approx \forall X{:}K_2.\, E_2}\ (\text{WEQ-ALL}) \qquad \frac{|K_1| \equiv |K_2| \qquad E_1 \approx E_2}{\lambda X{:}K_1.\, E_1 \approx \lambda X{:}K_2.\, E_2}\ (\text{WEQ-ABS})$$

Weak equality of eliminations and spines $\boxed{D \approx E}$ $\boxed{\boldsymbol{D} \approx \boldsymbol{E}}$

$$\frac{F_1 \approx F_2 \qquad \boldsymbol{E}_1 \approx \boldsymbol{E}_2}{F_1\, \boldsymbol{E}_1 \approx F_2\, \boldsymbol{E}_2}\ (\text{WEQ-ELIM}) \qquad \frac{}{\epsilon \approx \epsilon}\ (\text{WEQ-EMPTY}) \qquad \frac{D_1 \approx D_2 \qquad \boldsymbol{E}_1 \approx \boldsymbol{E}_2}{D_1, \boldsymbol{E}_1 \approx D_2, \boldsymbol{E}_2}\ (\text{WEQ-CONS})$$

Weak equality of kinds $\boxed{J \approx K}$

$$\frac{D_1 \approx D_2 \qquad E_1 \approx E_2}{D_1 \,..\, E_1 \approx D_2 \,..\, E_2}\ (\text{WEQ-INTV}) \qquad \frac{J_1 \approx J_2 \qquad K_1 \approx K_2}{(X{:}J_1) \to K_1 \approx (X{:}J_2) \to K_2}\ (\text{WEQ-DARR})$$

Figure 4.2 – Weak type equality

explicit markings and freely mix both representations, knowing that appropriate conversions could always be inserted. For example, we write $D\,E$ instead of $\overline{\underline{D}\,\underline{E}}$, $A\,\boldsymbol{B}$ instead of $\overline{\underline{A}\,\underline{\boldsymbol{B}}}$ and $F\,\boldsymbol{D}\,\boldsymbol{E}$ instead of $F\,(\boldsymbol{D}, \boldsymbol{E})$.

We also define a notion of *simple kinding contexts*, to be used primarily in Section 4.2. Simple kinding contexts γ, δ are best thought of as typing contexts consisting exclusively of type variable bindings $X{:}k$ with simple kind annotations k (as opposed to full kind annotations K). Their grammar is given in Fig. 4.1. As for full contexts, we assume that the variables bound in a simple context are all distinct. We write $\mathsf{dom}(\gamma)$ for the set of variables bound in γ and (γ, δ) for the concatenation of two simple contexts when $\mathsf{dom}(\gamma) \cap \mathsf{dom}(\delta) = \varnothing$.

4.1.2 Weak equality

So far we have considered two equality relations on types: syntactic equality, or α-equivalence $A \equiv B$, and judgmental type equality $\Gamma \vdash A = B : K$. Syntactic equality is simple but restrictive; judgmental type equality is flexible but complex. We now define an additional equality relation $A \approx B$ on types, called *weak equality*, which trades some of the simplicity of syntactic equality for some of the flexibility of judgmental equality.

Just as syntactic equality, weak equality is defined directly on raw types and kinds. However, unlike syntactic equality, weak equality identifies type operator abstractions up to simplification of their domain annotations, as shown in the following inference rule:

$$\frac{|J| \equiv |K| \qquad A \approx B}{\lambda X{:}J.\,A \approx \lambda X{:}K.\,B}\ (\text{WEq-Abs})$$

Weak equality is the smallest congruence (w.r.t. to all the type and kind formers) including the above rule. The complete set of inference rules is given in Fig. 4.2. Since we will mostly use weak equality to relate types and kinds in elimination form, we present inference rules for equality of heads, spines, eliminations and kinds.

The definition of weak equality includes no explicit inference rules for reflexivity, transitivity or symmetry, but these are easily proven admissible.

Lemma 4.1. *Weak equality is an equivalence, i.e. it is reflexive, transitive and symmetric.*

$$\frac{}{E \approx E}\ (\text{WEq-Refl}) \qquad \frac{E_1 \approx E_2 \qquad E_2 \approx E_3}{E_1 \approx E_3}\ (\text{WEq-Trans}) \qquad \frac{E_1 \approx E_2}{E_2 \approx E_1}\ (\text{WEq-Sym})$$

Although these three rules only cover the case of eliminations, analogous properties hold for kinds, heads and spines, i.e. weak equality is an equivalence on both types and kinds.

Proof. The three properties of reflexivity, transitivity and symmetry are proven separately; each simultaneously on the structure of kinds, eliminations, heads and spines, with a case analysis of the final rules used in the corresponding weak equality derivations. □

One may wonder why we did not adopt an even weaker notion of equality that ignores domain annotations altogether and identifies abstractions such as $\lambda X{:}*.\,E$ and $\lambda X{:}* \to *.\,E$. The answer is that weak equality is designed to fit well with the notion of *simplified kinding* introduced in Section 4.2, which ignores dependencies in kinds but preserves the simple kinding structure of types. The following property of weak equality will also prove useful in this setting.

Lemma 4.2. *Weakly equal kinds simplify equally. If $K_1 \approx K_2$, then $|K_1| \equiv |K_2|$.*

Proof. By straightforward induction on the derivation of $K_1 \approx K_2$. □

4.1.3 Hereditary substitution in raw types

In the next chapter, we will introduce a set of canonical rules that are defined directly on normal forms. Since kinds in $F^{\omega}_{\because}$ 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 $(Y\ A)[Y := \lambda X{:}K.\,B] \equiv (\lambda X{:}K.\,B)\ A$ 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. Thankfully, there is an operation that does precisely that: *hereditary substitution* [52].

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

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

$\boxed{\boldsymbol{D}[X^k := E]}$

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

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

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

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

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

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

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

Figure 4.3 – Hereditary substitution

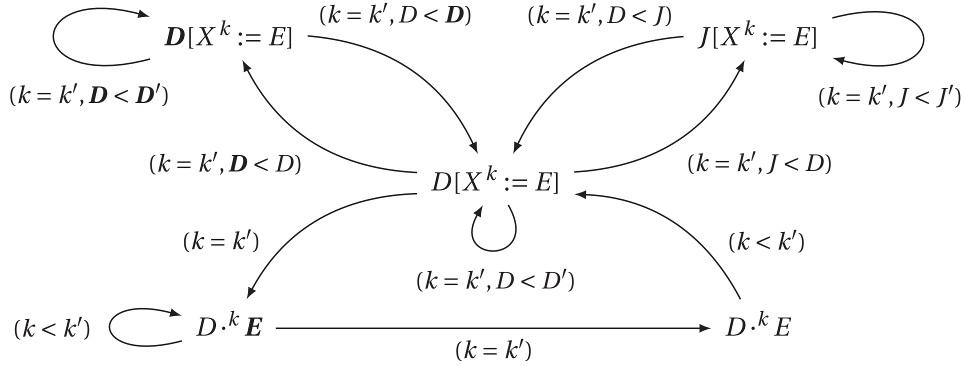

Figure 4.4 – Recursive structure of hereditary substitution

Our definition of hereditary substitution is given in Fig. 4.3. Hereditary substitution in raw kinds, eliminations and spines is defined mutually with *reducing application* of eliminations to eliminations and spines by recursion on the structure of the simple kind parameter k. The definitions of hereditary substitution in kinds, eliminations on spines proceed by inner recursion on the structure of the parameters K, D and $\boldsymbol{D}$, respectively. The mutually recursive structure of the five functions is illustrated as a call graph in Fig. 4.4. Nodes represent functions while edges represent calls among them. The edge labels indicate which parameter (if any) decreases during the corresponding call. Note that there are two calls where no parameter decreases: in the case where $D = X\,\boldsymbol{V}$, hereditary substitution in eliminations $D[X^k := E]$ is defined in terms of a reducing application $E \cdot^k (\boldsymbol{D}[X^k := E])$ where the simple kind k remains unchanged; there is a similar case in the definition of $D \cdot^k \boldsymbol{E}$. But as Fig. 4.4 illustrates, at least one of the relevant parameters decreases along every cycle in the call graph, i.e. there are no recursive calls where all the parameters remain the same. Hence the five functions remain structurally recursive, ensuring their totality.

There is only one recursive case in the definition of reducing applications to spines $D \cdot^k \boldsymbol{E}$, namely that where the simple kind parameter k and the spine parameter $\boldsymbol{E}$ are both compound expressions, i.e. where we have $k = k_1 \to k_2$ and $\boldsymbol{E} = E, \boldsymbol{E}'$. This is where spines shine. Because the structure of spines – like that of arrow kinds – is right-associative, we can recursively unwind $\boldsymbol{E}$ and k at the same time. The recursive call $(D \cdot^k E) \cdot^{k_2} \boldsymbol{E}'$ matches the tail $\boldsymbol{E}'$ of the spine with the domain k_2 of the arrow kind. This is precisely why we choose to define hereditary substitutions on elimination forms.

Our presentation of hereditary substitution differs slightly from others in the literature. The reasons for this are twofold. Firstly, our definition of hereditary substitution needs to cover dependent kinds, which is not the case for some other systems; secondly, we wanted our definition to be easily mechanized using a theorem prover. Like Keller and Altenkirch, we define hereditary substitution by structural recursion and mutually with reducing application [31]. Their paper describes an Agda implementation of hereditary substitution for simply typed lambda terms; i.e. it has already been mechanized in a theorem prover. However, their

implementation uses an intrinsically typed representation of lambda terms, which does not immediately generalize to a system with dependent types (or kinds). We choose, instead, to define hereditary substitution directly on raw, i.e. unkinded, type expressions. As a consequence, there are degenerate cases and the results of hereditary substitutions are not necessarily normal.

In that respect, our definition is closer to that given by Abel and Rodriguez who also define hereditary substitution directly on raw types [2]. Their definition also relies on the structure of kinds to ensure termination, but it requires tracking the kinds of both input types and result types, as well as their relationship to each other, in order to establish termination. This is necessary because their definition, unlike ours or that of Keller and Altenkirch, does not collect arguments of repeated applications in spines.

Neither of these definitions cover dependent types or kinds. Our approach of distinguishing between dependent and simple kinds and defining hereditary substitution by recursion on the latter was inspired by Harper and Licata's formalization of Canonical LF [28]. However, rather than defining a recursive function, they give an inductive definition of hereditary substitution as a collection of relations on raw but normal types and kinds. The relations are functional but partial, thus avoiding degenerate cases. On the other hand, functionality and termination have to be established separately for their definition.

We extend hereditary substitution pointwise to contexts, i.e. given a context Γ, a type variable $Y \notin \mathsf{dom}(\Gamma)$, a simple kind k and a type E, we define $\Gamma[Y^k := E]$ as

$$\begin{aligned}
\varnothing[Y^k := E] &= \varnothing \\
(\Gamma, x{:}D)[Y^k := E] &= \Gamma[Y^k := E],\ x : D[Y^k := E] \\
(\Gamma, X{:}K)[Y^k := E] &= \Gamma[Y^k := E],\ X : K[Y^k := E].
\end{aligned}$$

Next, we describe some simple properties of hereditary substitution. Since it is defined pointwise on spines, hereditary substitution commutes with spine concatenation.

Lemma 4.3. *Let $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ be spines, then $(\boldsymbol{D}_1, \boldsymbol{D}_2)[X^k := E] \equiv (\boldsymbol{D}_1[X^k := E], \boldsymbol{D}_2[X^k := E])$, for any X, k and E.*

Just as for ordinary substitution, simplified kinds are stable under hereditary substitution.

Lemma 4.4 (stability of simplifications under hereditary substitution)**.** *Let J be a kind, X a type variable, k a simple kind and A a type. Then $\lfloor J[X^k := A] \rfloor \equiv \lfloor J \rfloor$.*

Proof. By straightforward induction on the structure of J. □

The following lemma shows that hereditary substitutions of weakly equal types preserve weak equality.

Lemma 4.5. *Weak equality is a congruence w.r.t. hereditary substitution and reducing application. Let $E_1 \approx E_2$,*

1. *if $K_1 \approx K_2$, then $K_1[X^k := E_1] \approx K_2[X^k := E_2]$;*
2. *if $D_1 \approx D_2$, then $D_1[X^k := E_1] \approx D_2[X^k := E_2]$;*
3. *if $\boldsymbol{D}_1 \approx \boldsymbol{D}_2$, then $\boldsymbol{D}_1[X^k := E_1] \approx \boldsymbol{D}_2[X^k := E_2]$;*
4. *if $D_1 \approx D_2$, then $E_1 \cdot^k D_1 \approx E_2 \cdot^k D_2$;*
5. *if $\boldsymbol{D}_1 \approx \boldsymbol{D}_2$, then $E_1 \cdot^k \boldsymbol{D}_1 \approx E_2 \cdot^k \boldsymbol{D}_2$.*

Proof. The structure of the proof mirrors that of the recursive definitions of hereditary substitution and reducing application. All five parts are proven simultaneously, by induction on the structure of k. Parts 1–3 proceed by an inner induction on the derivations of $K_1 \approx K_2$, $D_1 \approx D_2$ and $\boldsymbol{D}_1 \approx \boldsymbol{D}_2$, respectively. Parts 4 and 5 proceed by a case analysis on the final rules used to derive $E_1 \approx E_2$ and $\boldsymbol{D}_1 \approx \boldsymbol{D}_2$, respectively. Since weak equality of eliminations is necessarily derived using WEQ-ELIM, part 2 proceeds by a case analysis on the final rule used to derive $F_1 \approx F_2$, where F_1 and F_2 are the heads, respectively, of $E_1 = F_1 \boldsymbol{D}_1$ and $E_2 = F_2 \boldsymbol{D}_2$. In the case for WEQ-ABS, we use stability of kind simplification under hereditary substitution (Lemma 4.4). In the case for WEQ-VAR, we use the IH twice: first for part 3 to derive $\boldsymbol{D}_1[X^k := E_1] \approx \boldsymbol{D}_2[X^k := E_2]$, then for part 5, to derive $E_1 \cdot^k (\boldsymbol{D}_1[X^k := E_1]) \approx E_2 \cdot^k (\boldsymbol{D}_2[X^k := E_2])$. In the second instance, k does not decrease nor is $\boldsymbol{D}_1[X^k := E_1] \approx \boldsymbol{D}_2[X^k := E_2]$ a sub-derivation of the current premise. This use of the IH is nevertheless justified because any subsequent use of the IH for part 2 in the proof of part 5 must occur after the use of the IH for part 4, at which point k has necessarily decreased. □

Since the essential difference between ordinary and hereditary substitution is that the latter reduces newly created β-redexes, one would expect the results of ordinary and hereditary substitutions to be β-convertible. As the following lemma shows, this is indeed the case.

Lemma 4.6. *Ordinary substitutions and applications in types β-reduce to hereditary substitutions and reducing applications, respectively. Let E be an elimination, X a type variable and k a simple kind, then*

1. *$K[X := E] \longrightarrow^*_\beta K[X^k := E]$ for any kind K;*
2. *$D[X := E] \longrightarrow^*_\beta D[X^k := E]$ for any type D;*
3. *for any D_1, D_2 and $\boldsymbol{D}$, if $D_1[X := E] \longrightarrow^*_\beta D_2$, then $(D_1\, \boldsymbol{D})[X := E] \longrightarrow^*_\beta D_2\, (\boldsymbol{D}[X^k := E])$;*
4. *$E\, D \longrightarrow^*_\beta E \cdot^k D$ for any type D;*
5. *$E\, \boldsymbol{D} \longrightarrow^*_\beta E \cdot^k \boldsymbol{D}$ for any spine $\boldsymbol{D}$.*

Proof. All five parts are proven simultaneously, by induction on the structure of k. Parts 1–3 proceed by an inner induction on the structure of K, D and $\boldsymbol{D}$, respectively. Parts 4 and 5 proceed by a case analysis on E and $\boldsymbol{D}$, respectively. All parts rely on compatibility of β-reduction with the kind and type formers. □

η-expansion of neutral types $\boxed{\eta_K(E)}$

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

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

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

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

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

Figure 4.5 – Normalization of types

As an immediate consequence of the previous lemma and subject reduction, the results of hereditary substitutions in well-formed kinds and well-kinded types are judgmentally equal to their ordinary counterparts.

Corollary 4.7 (soundness of hereditary substitution)**.** *Let* $\Gamma \vdash A : K$*, then*

1. *if* $\Gamma, X{:}K, \Delta \vdash J\ \mathsf{kd}$*, then* $\Gamma, \Delta[X := A] \vdash J[X := A] = \underline{\overline{J}[X^{|K|} := \overline{A}]}$*;*
2. *if* $\Gamma, X{:}K, \Delta \vdash B : J$*, then* $\Gamma, \Delta[X := A] \vdash B[X := A] = \underline{\overline{B}[X^{|K|} := \overline{A}]} : J[X := A]$.

4.1.4 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 given by Abel and Rodriguez [2], adjusted to also cover 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. 4.5.

The η-expansion $\eta_K(E)$ of a type E of kind K is defined by recursion on the structure of K.

It is defined for any type E, though it is designed primarily to expand *neutral forms*, i.e. eliminations of the form $E = X\,\boldsymbol{V}$ where $\boldsymbol{V}$ is a spine consisting entirely of $\beta\eta$-normal forms. In the definition of nf, η-expansion is used exclusively to expand type variables, which are indeed neutral types. The η-expansion $\eta_K(E)$ is similar to the weak η-expansion $\bar{\eta}_K(A)$ defined in the previous chapter, except that $\eta_K(E)$ also expands newly introduced argument variables, so that, when applied to a neutral type, the result is η-long.

Normalization $\mathsf{nf}_\Gamma(A)$ and $\mathsf{nf}_\Gamma(K)$ of raw types A and raw kinds K in a 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 simple kind, or equivalently, the simple kind of X. Since the normalization function is defined directly on raw, unkinded types, the only way to obtain this information is from the declared kind K of X 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 be normal.

We extend normalization pointwise to contexts, i.e. we define $\mathsf{nf}_\Gamma(\Delta)$ as

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

We drop the subscript Γ if it is the empty context, i.e. $\mathsf{nf}(\Delta) = \mathsf{nf}_\varnothing(\Delta)$.

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 $\mathsf{nf}(\Omega) \equiv (X\,X)[X^* := \lambda X{:}*.\,X\,X] \equiv (\lambda X{:}*.\,X\,X) \cdot^* (\lambda X{:}*.\,X\,X) \equiv \Omega$. However, we will show in Section 4.1.5 that, for well-kinded types $\Gamma \vdash A : K$ and well-formed kinds $\Gamma \vdash K\ \mathsf{kd}$, the type $\mathsf{nf}_\Gamma(A)$ and kind $\mathsf{nf}_\Gamma(K)$ are guaranteed to be η-long β-normal forms.

Before we can do so, we need to establish some basic properties of η-expansion and normalization. Unsurprisingly, simplified kinds are stable under normalization.

Lemma 4.8 (stability of simplifications under normalization)**.** *Let Γ be a context and K a kind. Then $|\mathsf{nf}_\Gamma(K)| \equiv |K|$.*

Proof. By straightforward induction on the structure of K. □

Extending kind simplification pointwise to contexts, we define $|\Gamma|$ as

$$|\varnothing| = \varnothing \qquad |\Gamma, x{:}A| = |\Gamma| \qquad |\Gamma, X{:}K| = |\Gamma|, X{:}|K|$$

It is easy to see that context lookup commutes with simplification, i.e. $|\Gamma|(X) = |\Gamma(X)|$, and that

simplified contexts are stable under (hereditary) substitution and normalization, i.e.

$$|\Gamma[X := A]| = |\Gamma| \qquad |\Gamma[X^k := A]| = |\Gamma| \qquad |\mathsf{nf}(\Gamma)| = |\Gamma|$$

The following two lemmas show that η-expansion and normalization preserve weak equality. Importantly, this is true even if the corresponding kinds and contexts, respectively, are not themselves weakly equal but simplify equally – a much weaker requirement.

Lemma 4.9. *Weak equality is preserved by η-expansion along kinds that simplify equally. If $|J| \equiv |K|$ and $D \approx E$, then $\eta_J(D) \approx \eta_K(E)$.*

Proof. By induction on the structure of J and case analysis on the final rule used to derive $D \approx E$. □

Lemma 4.10. *Kinds and types normalize weakly equally in contexts that simplify equally. Let Γ and Δ be contexts such that $|\Gamma| \equiv |\Delta|$. Then*

1. $\mathsf{nf}_\Gamma(K) \approx \mathsf{nf}_\Delta(K)$ *for any kind K, and*
2. $\mathsf{nf}_\Gamma(A) \approx \mathsf{nf}_\Delta(A)$ *for any type A.*

Proof. Simultaneously, by induction on the structure of K and A, respectively. In the type variable case $A = X$ we use Lemma 4.9; in the operator application case $A = A_1\, A_2$ we use Lemma 4.5.2; in the cases for dependent operator kinds, universal types and operator abstraction, we use Lemma 4.2. □

4.1.5 Soundness of normalization

To conclude this section, we show that η-expansion and normalization do not fundamentally alter the meaning of a type or kind. Well-kinded types and well-formed kinds are judgmentally equal to their normalized counterparts.

Lemma 4.11 (soundness of η-expansion)**.** *Well-kinded neutral types are equal to their η-expansions. Let $\Gamma \vdash X\,\boldsymbol{V} : K$, then $\Gamma \vdash X\,\boldsymbol{V} = \eta_K(X\,\boldsymbol{V}) : K$.*

Proof. By induction on the structure of K. In the case for $K = (Y{:}K_1) \to K_2$ we use kinding validity and a case analysis on the final rule of the resulting kind formation derivation to obtain $\Gamma \vdash K_1$ kd and $\Gamma, Y{:}K_1 \vdash K_2$ kd. By context validity, K-VAR and the IH, we have $\Gamma, Y{:}K_1 \vdash Y = \eta_{K_1}(Y) : K_2$. By the weakening lemma, TEQ-REFL, TEQ-APP and a second use of the IH, we obtain $\Gamma, Y{:}K_1 \vdash X\,\boldsymbol{V}\,Y = \eta_{K_2}(X\,\boldsymbol{V}\,(\eta_{K_1}(Y))) : K_1$, and hence

$$\begin{aligned} \Gamma \vdash X\,\boldsymbol{V} &= \lambda Y{:}K_1.\, X\,\boldsymbol{V}\,Y && \text{(by TEQ-}\eta\text{)} \\ &= \lambda Y{:}K_1.\, \eta_{K_2}(X\,\boldsymbol{V}\,(\eta_{K_1}(Y))) : K. && \text{(by KEQ-REFL and TEQ-ABSWEAK)} \end{aligned}$$

□

Lemma 4.12 (soundness of normalization). *Well-formed kinds and well-kinded types are equal to their normal forms.*

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

Proof. Simultaneously by induction kind formation, kinding and context formation derivations, respectively. Most cases are routine. The interesting cases are K-VAR and K-APP. To avoid clutter, we omit the subscript $\mathsf{nf}(\Gamma)$ in the remainder of the proof, writing e.g. $\mathsf{nf}(A)$ instead of $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$.

- *Case* K-VAR. We have $A = X$ and $\Gamma(X) = K$ for some X and K, as well as Γ ctx. By the IH for part 3, we obtain $\Gamma = \mathsf{nf}(\Gamma)$ ctx and hence $\Gamma \vdash K \equiv \Gamma(X) = \mathsf{nf}(\Gamma)(X) \equiv \mathsf{nf}(K)$. By K-VAR and K-CONV we get $\Gamma \vdash X : \mathsf{nf}(K)$, and by Lemma 4.11, $\Gamma \vdash X = \eta_{\mathsf{nf}(K)}(X) : \mathsf{nf}(K)$. We conclude by KEQ-SYM and K-CONV.
- *Case* K-APP. We have $A = A_1\, A_2$ with $\Gamma \vdash A_1 : (X{:}K_1) \to K_2$ and $\Gamma \vdash A_2 : K_1$. Applying the IH twice, we obtain $\Gamma \vdash A_1 = \mathsf{nf}(A_1) : (X{:}K_1) \to K_2$ and $\Gamma \vdash A_2 = \mathsf{nf}(A_1) : K_1$, and hence $\Gamma \vdash A_1\, A_2 = (\mathsf{nf}(A_1))\,(\mathsf{nf}(A_2)) : K_2[X := A_2]$ by TEQ-APP. Next, we distinguish two cases: $\mathsf{nf}(A_1) = \lambda X{:}J.\, E$ and $\mathsf{nf}(A_1) \neq \lambda X{:}J.\, E$. In the latter case, we are done. Otherwise, we use equation validity, generation of kinding for operators (Lemma 3.40) and kinding validity to derive $\Gamma \vdash \mathsf{nf}(A_1) : K_1$, $\Gamma, X{:}K_1 \vdash E : K_2$ and $\Gamma, X{:}K_1 \vdash K_2$ kd. It follows that

$$
\begin{aligned}
\Gamma \vdash (\lambda X{:}J.\, E)\,(\mathsf{nf}(A_2)) \;&=\; E[X := \mathsf{nf}(A_2)] && \text{(by TEQ-}\beta'\text{)}\\
&=\; E[X^{|K_1|} := \mathsf{nf}(A_2)] \;:\; K_2[X := \mathsf{nf}(A_2)]. && \text{(by Corollary 4.7)}
\end{aligned}
$$

  Functionality (Lemma 3.18) and TEQ-SYM give us $\Gamma \vdash K_2[X := \mathsf{nf}(A_2)] = K_2[X := A_2]$, which allows us to adjust the kind of the previous equation via TEQ-CONV, and to conclude the case by TEQ-TRANS. □

4.2 Simple kinding of normal types

The function nf assigns to each raw type A in a given context Γ a unique type $\mathsf{nf}_\Gamma(A)$. But as we have seen, the type $\mathsf{nf}_\Gamma(A)$ may not be $\beta\eta$-normal if A is ill-kinded. In this section, we prove the converse: whenever A is well-kinded in Γ, the type $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$ is a η-long β-normal form. To do so, we first introduce a set of *simplified kinding* judgments. Roughly, a simplified kinding judgment $\gamma \vdash E : k$ establishes that the type E is a normal form of *simple* kind k in the *simple* context γ. Given $\gamma \vdash E : k$, we say that E is a *simply (well-)kinded normal form*, or just that E is *simply kinded*. As we are about to show, every well-kinded type $\Gamma \vdash A : K$ has a simply well-kinded normal form $|\Gamma| \vdash E : |K|$, namely $E = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$ (see Lemma 4.20 below).

It is important to note that the converse is not true: not every simply kinded type is well-kinded. Because kind simplification forgets dependencies, there are necessarily some ill-kinded types

that are considered simply well-kinded according to the judgments we are about to introduce. However, every simply kinded type is guaranteed to be an η-long β-normal form and, as we will see in this section, simple kinding is preserved by operations such as hereditary substitution and η-expansion. Hence simple kinding allows us to prove important properties about these operations on $\beta\eta$-normal forms without subjecting ourselves to the complexity of fully dependent kinds.

To enhance readability, we use the following naming conventions for normal forms: the metavariables U, V, W denote normal types, while M and N denote neutral types.[1] No special notation is used for normal kinds.

Judgments. Fig. 4.6 defines the following judgments by mutual induction.

$\gamma \vdash K\ \mathsf{kds}$	the kind K is simply well-formed and normal in γ
$\gamma \vdash V : k$	the type V is a normal form of simple kind k in γ
$\gamma \vdash_{\mathsf{ne}} N : k$	the type N is a neutral form of simple kind k in γ
$\gamma \vdash j : \boldsymbol{V} : k$	applying an operator of simple kind j to the normal spine $\boldsymbol{V}$ yields a type of simple kind k in γ.

The judgments for simple kind formation and kinding follow the syntactic structure of normal kinds and types. A type V is a $\beta\eta$-normal form $\gamma \vdash V : k$ of simple kind k if it is either a proper type introduced by one of the basic type formers applied to normal arguments (rules SK-Top, SK-Bot, SK-Arr, and SK-All), an operator abstraction with a normal body (rule SK-Abs), or a simply kinded neutral type (rule SK-Ne). Simply kinded neutral forms $\gamma \vdash_{\mathsf{ne}} N : k$ are eliminations headed by an abstract type operator, i.e. a type variable X, which is applied to a spine of normal types $\boldsymbol{V}$ (rule SK-VarApp). Finally, a simply well-formed normal kind $\gamma \vdash K\ \mathsf{kds}$ is either a type interval bounded by normal types (rule SWf-Intv) or a dependent arrow with normal domain and codomain (rule SWf-DArr). Note that type operator abstractions are the only normal forms of (simple) arrow kind. This ensures that normal types are always η-long.

The simple spine kinding judgment $\gamma \vdash j : \boldsymbol{V} : k$ is different from the other judgment forms in that it is a quaternary rather than a ternary relation. The simple kinds j and k should be read as inputs and outputs, respectively, of such judgments: when a type of kind j is applied to the spine $\boldsymbol{V}$ (the subject of the judgment), the resulting type is of kind k – as exemplified by the rule SK-VarApp.

There is no formation judgment for simple contexts γ since such contexts only contain simple type variable bindings, i.e. bindings assigning simple kinds to type variables, and there is no such thing as an ill-formed simple kind.

[1] This is just notation. We do not consider normal forms a separate syntactic category, e.g. the letters U, V, W are metavariables denoting types (typically in elimination form) rather than non-terminals in some grammar of normal forms.

Simplified well-formedness of kinds $\boxed{\gamma \vdash K \ \mathsf{kds}}$

$$\frac{\gamma \vdash U : * \qquad \gamma \vdash V : *}{\gamma \vdash U \,..\, V \ \mathsf{kds}} \ (\text{SWF-INTV}) \qquad \frac{\gamma \vdash J \ \mathsf{kds} \qquad \gamma, X{:}|J| \vdash K \ \mathsf{kds}}{\gamma \vdash (X{:}J) \to K \ \mathsf{kds}} \ (\text{SWF-DARR})$$

Kinding of neutral types $\boxed{\gamma \vdash_{\mathsf{ne}} N : k}$

$$\frac{\gamma(X) = j \qquad \gamma \vdash j : \boldsymbol{V} : k}{\gamma \vdash_{\mathsf{ne}} X\,\boldsymbol{V} : k} \ (\text{SK-VARAPP})$$

Simple spine kinding $\boxed{\gamma \vdash j : \boldsymbol{V} : k}$

$$\frac{}{\gamma \vdash k : \epsilon : k} \ (\text{SK-EMPTY}) \qquad \frac{\gamma \vdash U : j \qquad \gamma \vdash k : \boldsymbol{V} : l}{\gamma \vdash j \to k : U, \boldsymbol{V} : l} \ (\text{SK-CONS})$$

Simple kinding of normal types $\boxed{\gamma \vdash V : k}$

$$\frac{}{\gamma \vdash \top : *} \ (\text{SK-TOP}) \qquad \frac{}{\gamma \vdash \bot : *} \ (\text{SK-BOT})$$

$$\frac{\gamma \vdash U : * \qquad \gamma \vdash V : *}{\gamma \vdash U \to V : *} \ (\text{SK-ARR}) \qquad \frac{\gamma \vdash K \ \mathsf{kds} \qquad \gamma, X{:}|K| \vdash V : *}{\gamma \vdash \forall X{:}K.\, V : *} \ (\text{SK-ALL})$$

$$\frac{\gamma \vdash J \ \mathsf{kds} \qquad \gamma, X{:}|J| \vdash V : k}{\gamma \vdash \lambda X{:}J.\, V : |J| \to k} \ (\text{SK-ABS}) \qquad \frac{\gamma \vdash_{\mathsf{ne}} N : *}{\gamma \vdash N : *} \ (\text{SK-NE})$$

Figure 4.6 – Simplified kinding

Because kinding is simplified, there is no notion of subkinding or kind equality, and hence no need for a subsumption rule. As a consequence, the simple kind formation and kinding rules are syntax-directed; in a given context, the rules unambiguously assign simple kinds to any normal type. Since neutral types N are always of the form $N = X\,\boldsymbol{V}$ for some X and $\boldsymbol{V}$, the judgment from $\gamma \vdash_{\mathsf{ne}} N : k$ is somewhat superfluous and the rule SK-VARAPP could, in principle, be merged into SK-NE. However, the rule SK-NE only covers proper neutral types, i.e. those of simple kind $*$. We keep the separate judgment for neutrals as it is convenient when reasoning about higher-order neutral types (see e.g. the admissible rule SK-NEAPP and Lemma 4.18 below).

Another important property of simplified kinding is that none of the rules involve substitutions in kinds. This substantially simplifies the proofs of some key lemmas about hereditary substitutions and reducing applications discussed later on in this section, such as Lemma 4.16 which states that hereditary substitutions preserve simple kinding (and thus normal forms) and that

simple kinding of reducing applications is admissible. It is also important in establishing admissibility of the following simple rules about spines and neutral types.

Lemma 4.13. *The following simple kinding rules for spine concatenation and application of neutrals are admissible.*

$$\frac{\gamma \vdash j : \boldsymbol{U} : k \qquad \gamma \vdash k : \boldsymbol{V} : l}{\gamma \vdash j : \boldsymbol{U}, \boldsymbol{V} : l} \text{ (SK-CONCAT)} \qquad \frac{\gamma \vdash j : \boldsymbol{U} : k \to l \qquad \gamma \vdash V : k}{\gamma \vdash j : \boldsymbol{U}, V : l} \text{ (SK-SNOC)}$$

$$\frac{\gamma \vdash_{\mathsf{ne}} N : j \to k \qquad \gamma \vdash V : j}{\gamma \vdash_{\mathsf{ne}} N\, V : k} \text{ (SK-NEAPP)}$$

Proof. The proofs are done separately for each of the three rules in the order the rules are listed. The proof for SK-CONCAT is by induction on the derivation of the first premise. The rule SK-CONS is derivable from SK-SNOC as a special case where $\boldsymbol{V} = V, \epsilon$, using SK-EMPTY and SK-CONS. The proof of SK-NEAPP starts with a case analysis on the final rule used to derive $\Gamma \vdash_{\mathsf{ne}} N : j \to k$. The only rule for deriving such judgments is SK-VARAPP, hence N must be of the form $N = X\,\boldsymbol{U}$ with $\gamma(X) = l$ and $\gamma \vdash l : \boldsymbol{U} : j \to k$. We conclude by SK-SNOC and SK-VARAPP. □

4.2.1 Simply-kinded hereditary substitution

Before we can prove that hereditary substitutions preserve simple kinding, we first need to establish the usual weakening properties for simple kind formation and kinding.

Lemma 4.14 (weakening)**.** *A simple judgment remains true if its context is extended by an additional binding. Let γ, δ be simple contexts, k a simple kind and $X \notin \mathsf{dom}(\gamma,\delta)$. If $\gamma,\delta \vdash \mathcal{J}$ for any of the simple judgments defined above, then $\gamma, X{:}k, \delta \vdash \mathcal{J}$.*

Proof. Simultaneously for all four judgments, by induction on the derivation of $\gamma,\delta \vdash \mathcal{J}$. □

Corollary 4.15 (Iterated weakening)**.** *Given a pair γ, δ of disjoint simple contexts, if $\gamma \vdash \mathcal{J}$, then $\gamma,\delta \vdash \mathcal{J}$.*

Lemma 4.16 (hereditary substitution)**.** *Hereditary substitutions and reducing applications preserve the simple kinds of types as well as simple well-formedness of kinds. Let γ, δ be simple contexts and X such that $X \notin \mathsf{dom}(\gamma,\delta)$. Assume further that $\gamma \vdash V : k$ for some V and k. Then*

1. *if $\gamma, X{:}k, \delta \vdash J$ kds, then $\gamma,\delta \vdash J[X^k := V]$ kds;*
2. *if $\gamma, X{:}k, \delta \vdash U : j$, then $\gamma,\delta \vdash U[X^k := V] : j$;*
3. *if $\gamma, X{:}k, \delta \vdash_{\mathsf{ne}} N : j$, then $\gamma,\delta \vdash N[X^k := V] : j$ as a normal form;*
4. *if $\gamma, X{:}k, \delta \vdash j : \boldsymbol{U} : l$, then $\gamma,\delta \vdash j : \boldsymbol{U}[X^k := V] : l$;*
5. *if $k = k_1 \to k_2$ and $\gamma \vdash U : k_1$, then $\gamma \vdash V \cdot^{k_1 \to k_2} U : k_2$;*
6. *if $\gamma \vdash k : \boldsymbol{U} : j$, then $\gamma \vdash V \cdot^{k} \boldsymbol{U} : j$.*

Note that hereditary substitutions preserve the simple kinds of neutral types but not neutrality itself.

Proof. All six parts are proven simultaneously by induction on the structure of k. Parts 1–4 proceed by an inner induction on the simple formation or kinding derivations for J, U, N and $\boldsymbol{U}$, respectively. Parts 5 and 6 proceed by a case analysis on the final rules used to derive $\gamma \vdash V : k_1 \to k_2$ and $\gamma \vdash k : \boldsymbol{V} : j$, respectively; for part 5, the only applicable rule is SK-ABS. For part 3, in the case for SK-VARAPP when $N = X\,\boldsymbol{U}$, we use iterated weakening (Corollary 4.15) and the IH (for 4), respectively, to obtain $\gamma, \delta \vdash V : k$ and $\gamma, \delta \vdash k : \boldsymbol{U}[X^k := V] : j$. To conclude the case, we apply the IH again (for 6). In this second use of the IH, k does not decrease nor is $\gamma, \delta \vdash k : \boldsymbol{U}[X^k := V] : j$ a strict sub-derivation of the current premise. However, in order to use the IH for part 3 again from within the proof of part 6, we must go through part 5, at which point k necessarily decreases. Again, the structure of the proof mirrors that of the mutually recursive definitions of hereditary substitution and reducing application. □

Thanks to Lemma 4.16, we can now prove the following commutativity lemma about hereditary substitutions, which will play an important role in the proof of Lemma 4.26 below and in the development of the next chapter.

Lemma 4.17 (commutativity of hereditary substitutions)**.** *Hereditary substitutions of simply kinded types commute; hereditary substitutions of simply kinded types commute with simply kinded reducing applications. Let $\gamma_1 \vdash U : j$ and $\gamma_1, X{:}k, \gamma_2 \vdash V : k$. Then*

1. *if $\gamma_1, X{:}j, \gamma_2, Y{:}k, \gamma_3 \vdash J$ kds, then*

$$J[Y^k := V][X^j := U] \equiv J[X^j := U][Y^k := V[X^j := U]];$$

2. *if $\gamma_1, X{:}j, \gamma_2, Y{:}k, \gamma_3 \vdash W : l$, then*

$$W[Y^k := V][X^j := U] \equiv W[X^j := U][Y^k := V[X^j := U]];$$

3. *if $\gamma_1, X{:}j, \gamma_2, Y{:}k, \gamma_3 \vdash_{\mathsf{ne}} N : l$, then*

$$N[Y^k := V][X^j := U] \equiv N[X^j := U][Y^k := V[X^j := U]];$$

4. *if $\gamma_1, X{:}j, \gamma_2, Y{:}k, \gamma_3 \vdash l_1 : \boldsymbol{W} : l_2$, then*

$$\boldsymbol{W}[Y^k := V][X^j := U] \equiv \boldsymbol{W}[X^j := U][Y^k := V[X^j := U]];$$

5. *if $k = k_1 \to k_2$ and $\gamma_1, X{:}j, \gamma_2 \vdash W : k_1$, then*

$$(V \cdot^{k_1 \to k_2} W)[X^j := U] \equiv (V[X^j := U]) \cdot^{k_1 \to k_2} (W[X^j := U]);$$

6. *if $\gamma_1, X{:}j, \gamma_2 \vdash k : \boldsymbol{W} : l$, then $(V \cdot^k \boldsymbol{W})[X^j := U] \equiv (V[X^j := U]) \cdot^k (\boldsymbol{W}[X^j := U])$.*

Proof. All six parts are proven simultaneously by simultaneous induction on the structures

of j and k. Simultaneous structural induction on j and k means roughly that it is sufficient for either one of j or k to decrease in an induction step. More formally, denote by $\sqsubseteq$ the sub-expression order on simple kinds, then the simultaneous induction order $<$ on unordered pairs $\{j, k\}$ of simple kinds is defined as $\{j_1, j_2\} < \{k_1, k_2\}$ if $j_1 \sqsubset k_1$ and $j_2 \sqsubseteq k_2$. Importantly, $<$ is defined over *unordered* pairs which allows us to exchange j and k in an induction step. Parts 1–4 proceed by an inner induction on the simple formation or kinding derivations for J, W, N and $\boldsymbol{W}$, respectively.

As usual, the interesting cases are those for part 3, when $N = Y\,\boldsymbol{W}$ and $N = X\,\boldsymbol{W}$.

- *Case* SK-VarApp, $N = Y\,\boldsymbol{W}$. We have $\gamma_1, X{:}j, \gamma_2, Y{:}k, \gamma_3 \vdash k : \boldsymbol{W} : l$. By Lemma 4.16.4, we obtain $\gamma_1, X{:}j, \gamma_2, \gamma_3 \vdash k : \boldsymbol{W}[Y^k := V] : l$, and hence we have

$$
\begin{aligned}
&(Y\,\boldsymbol{W})[Y^k := V][X^j := U] \\
&\quad \equiv (V \cdot^k (\boldsymbol{W}[Y^k := V]))[X^j := U] && \text{(by definition)} \\
&\quad \equiv (V[X^j := U]) \cdot^k (\boldsymbol{W}[Y^k := V][X^j := U]) && \text{(by the IH for 6)} \\
&\quad \equiv (V[X^j := U]) \cdot^k (\boldsymbol{W}[X^j := U][Y^k := V[X^j := U]]) && \text{(by the IH for 4)} \\
&\quad \equiv (Y\,\boldsymbol{W})[X^j := U][Y^k := V[X^j := U]]. && \text{(by definition)}
\end{aligned}
$$

- *Case* SK-VarApp, $N = X\,\boldsymbol{W}$. We have $\gamma_1, X{:}j, \gamma_2, Y{:}k, \gamma_3 \vdash j : \boldsymbol{W} : l$. By Lemma 4.16.4, we obtain $\gamma_1, X{:}j, \gamma_2, \gamma_3 \vdash j : \boldsymbol{W}[Y^k := V] : l$, and hence we have

$$
\begin{aligned}
&(X\,\boldsymbol{W})[Y^k := V][X^j := U] \\
&\quad \equiv (X\,(\boldsymbol{W}[Y^k := V]))[X^j := U] && \text{(by definition)} \\
&\quad \equiv U \cdot^j (\boldsymbol{W}[Y^k := V][X^j := U]) && \text{(by definition)} \\
&\quad \equiv U \cdot^j (\boldsymbol{W}[X^j := U][Y^k := V[X^j := U]]) && \text{(by the IH for 4)} \\
&\quad \equiv (U[Y^k := V[X^j := U]]) \cdot^j (\boldsymbol{W}[X^j := U][Y^k := V[X^j := U]]) && \text{(as } Y \notin \mathsf{fv}(U)\text{)} \\
&\quad \equiv (U \cdot^j (\boldsymbol{W}[X^j := U]))[Y^k := V[X^j := U]] && \text{(by the IH for 6)} \\
&\quad \equiv (X\,\boldsymbol{W})[X^j := U][Y^k := V[X^j := U]]. && \text{(by definition)}
\end{aligned}
$$

Note that, in the second case, we switched the roles of the simple kinds j and k when invoking the IH for part 6. □

4.2.2 Simplification and normalization of kinding

Thanks to Lemma 4.12 we know that the definition of the normalization function nf is sound, i.e. that well-formed kinds $\Gamma \vdash K\ \mathsf{kd}$ and well-kinded types $\Gamma \vdash A : K$ are convertible with $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(K)$ and $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$, respectively. But we have yet to establish that $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(K)$ and $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$ are actually normal forms. In this section, we prove a more general result, namely that, whenever $\Gamma \vdash K\ \mathsf{kd}$ and $\Gamma \vdash A : K$, it follows that $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(K)$ is a simply well-formed normal kind and and $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$ is a simply well-kinded normal type.

As a first step, we show that the simple kinds of variables and, more generally, of neutral types are preserved by η-expansion.

Lemma 4.18. *η-expansion preserves the simple kinds of neutral types. Assume $\gamma \vdash K$ kds and $\gamma \vdash_{\mathsf{ne}} N : |K|$. Then $\gamma \vdash \eta_K(N) : |K|$.*

Proof. By induction on the structure of K. The case fore $K = (X{:}K_1) \to K_2$ proceeds by case analysis on the final rules used to derive $\gamma \vdash (X{:}K_1) \to K_2$ kds and $\gamma \vdash_{\mathsf{ne}} N : |K_1| \to |K_2|$ and uses the weakening lemma (Lemma 4.14) as well as SK-NEAPP. □

Next, we require a syntactic notion of normal contexts. We define the *simple context formation* judgment Γ ctxs as the pointwise lifting of simple kind formation and kinding to bindings:

$$\frac{}{\varnothing \ \mathsf{ctxs}} \qquad \frac{\Gamma \ \mathsf{ctxs} \qquad |\Gamma| \vdash K \ \mathsf{kds}}{\Gamma, X{:}K \ \mathsf{ctxs}} \qquad \frac{\Gamma \ \mathsf{ctxs} \qquad |\Gamma| \vdash V : *}{\Gamma, x{:}V \ \mathsf{ctxs}}$$

Since simple kind formation and kinding is defined on normal kinds and types, a simply well-formed context Γ is also normal. Conversely, if we lookup the declared kind or type of a variable in a simply well-formed context, the result is guaranteed to be a normal form.

Lemma 4.19. *The declared kinds and types of variables in a simply well-formed context Γ are simply well-formed and well-kinded, respectively, in $|\Gamma|$, i.e*

$$\frac{\Gamma, X{:}K, \Delta \ \mathsf{ctxs}}{|\Gamma, X{:}K, \Delta| \vdash K \ \mathsf{kds}} \ (\text{SC-TPLOOKUP}) \qquad \frac{\Gamma, x{:}V, \Delta \ \mathsf{ctxs}}{|\Gamma, x{:}V, \Delta| \vdash V : *} \ (\text{SC-TMLOOKUP})$$

Proof. Both parts are proven separately by structural induction on Δ and case analysis on the final rule used to derive the premise. In the inductive case, we use the weakening lemma for simple kind formation. □

With Lemmas 4.18 and 4.19 at hand, it is easy to show that nf does indeed produce normal forms.

Lemma 4.20 (normalization and simplification)**.** *Well-formed kinds and well-kinded types have simply well-formed and simply kinded normal forms, respectively.*

1. *If $\Gamma \vdash K$ kd, then $|\mathsf{nf}(\Gamma)| \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma)}(K)$ kds.*
2. *If $\Gamma \vdash A : K$, then $|\mathsf{nf}(\Gamma)| \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A) : |K|$.*
3. *If Γ ctx, then $\mathsf{nf}(\Gamma)$ ctxs.*

The proof uses the following helper lemma about simplified subkinds, which is proven by straightforward induction on subkinding derivations.

Lemma 4.21. *Subkinds simplify equally. If $\Gamma \vdash J \le K$, then $|J| \equiv |K|$.*

Proof of Lemma 4.20. Simultaneously by induction on declarative kind formation, kinding, and context formation derivations. The only interesting cases are K-SUB (where we use Lemma 4.21), K-VAR and K-APP. In the case for K-VAR, where $A = X$, we use the IH for part 3 and SC-TPLOOKUP to obtain $|\mathsf{nf}(\Gamma)| \vdash K\,\mathsf{kds}$ for $K = \mathsf{nf}(\Gamma)(X)$, and we conclude by Lemma 4.18. In the case for K-APP, where $A = A_1\,A_2$, we start by applying the IH to obtain $|\mathsf{nf}(\Gamma)| \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A_1) : |K_1| \to |K_2|$, and $|\mathsf{nf}(\Gamma)| \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A_2) : |K_1|$. The first of these judgments must be derived using SK-ABS because that is the only simple kinding rule assigning an arrow kind to a type. Hence $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A_1) = \lambda X{:}J.\,V$ for some J and V such that $|J| \equiv |K_1|$ and $|\mathsf{nf}(\Gamma)|, X{:}|K_1| \vdash V : |K_2|$. We conclude by the hereditary substitution lemma for normal types (Lemma 4.16.2) and Lemma 4.4. □

Commutativity of normalization and substitution

We are now almost ready to introduce the canonical presentation of $F^{\omega}_{\cdot\cdot}$. Our final task in this chapter is to establish another commutativity property that will play a crucial role in proving equality of declarative and canonical subtyping: the fact that normalization commutes with substitution.

In the past few sections, we have seen that well-formed kinds and well-kinded types have normal forms (Lemma 4.20) and that these normal forms are convertible to the kinds and types they were computed from (Lemma 4.12). By validity, context conversion and kind conversion, this means that every declarative subtyping judgment $\Gamma \vdash A \leq B : K$ has an associated judgment $\mathsf{nf}(\Gamma) \vdash \mathsf{nf}(A) \leq \mathsf{nf}(B) : \mathsf{nf}(K)$ relating the normal forms of the original expressions.

Our goal for the next chapter is to come up with a set of canonical rules for deriving such judgments which are defined directly on normal forms – similar to the simple kinding and kind formation judgments introduced in this chapter. To establish equivalence of the two sets of rules will require a canonical rule – possibly a derivable or admissible one – for every declarative rule. But some of the declarative rules, such as the subtyping rules ST-$\beta_{1,2}$ for β-conversions, or the kinding rule K-APP for applications, involve substitutions, which do not preserve normal forms. To see why this is a problem, consider the declarative rule K-APP:

$$\frac{\Gamma \vdash A : (X{:}J) \to K \qquad \Gamma \vdash B : J}{\Gamma \vdash A\,B : K[X := B]}$$

By soundness of normalization (Lemma 4.12), type equation validity (Lemma 3.17), and context conversion (Corollary 3.11), we know that the following is also admissible:

$$\frac{\mathsf{nf}(\Gamma) \vdash U : (X{:}J') \to K' \qquad \mathsf{nf}(\Gamma) \vdash V : J'}{\mathsf{nf}(\Gamma) \vdash \mathsf{nf}(A\,B) : \mathsf{nf}(K[X := B])}$$

where $U = \mathsf{nf}(A)$, $V = \mathsf{nf}(B)$, $J' = \mathsf{nf}(J)$ and $K' = \mathsf{nf}(K)$. By Lemma 4.20, we know that U and V are simply well-kinded normal types, and that J' and K' are simply well-formed normal

kinds. For our canonical application rule, we would like to express the type $\mathsf{nf}(A\,B)$ and the kind $\mathsf{nf}(K[X := B])$ in the conclusion directly using U, V, J' and K'. This is relatively straightforward for the application $\mathsf{nf}(A\,B)$ because we know that U must be an operator abstraction $U = \lambda X{:}L.\,W$; after all, U has simple kind $|J| \to |K|$ and normal forms are η-long. We also know that $|L| \equiv |J| \equiv |J'|$ (see the proof of Lemma 4.20 for details). Hence $\mathsf{nf}(A\,B) \equiv W[X^{|J'|} := V]$ by definition of nf, and we are done.

Things are more complicated for the normal kind $\mathsf{nf}(K[X := B])$. The definition of the normalization function nf does not tell us anything immediately useful about substitutions. Indeed, we know that substitutions do not preserve normal forms, e.g. $(Y\,V)[Y := \lambda X{:}J'.\,W]$ is not a normal form, even if $Y\,V$ and $\lambda X{:}J'.\,W$ are. However, Corollary 4.7.1 tells us that substitutions in kinds are judgmentally equal to hereditary substitutions, i.e. $\Gamma \vdash K[X := B] = K[X^{|J'|} := B]$, and Lemma 4.16.1 tells us that hereditary substitutions preserve normal forms, all of which suggests that $\mathsf{nf}(K[X := B])$ should be equal to $K'[X^{|J'|} := V]$. This is indeed the case; one can show that $\Gamma \vdash \mathsf{nf}(K[X := B]) = K'[X^{|J'|} := V]$. But there is a caveat: the two normal forms are not *syntactically* equal, i.e. $\mathsf{nf}(K[X := B]) \not\equiv K'[X^{|J'|} := V]$. Similarly, $\mathsf{nf}(A[X := B]) \not\equiv \mathsf{nf}(A)[X^k := \mathsf{nf}(B)]$ for types A and B in general.

This fact is best illustrated through the case of type variables, i.e. when $A = X$ and we have $\mathsf{nf}_\Gamma(X[X := B]) \equiv \mathsf{nf}_\Gamma(B)$ and $(\mathsf{nf}_\Gamma(X))[X^k := \mathsf{nf}(B)] \equiv (\eta_{\Gamma(X)}(X))[X^k := \mathsf{nf}(B)]$. We would like to show that $(\eta_K(X))[X^k := V]$ is syntactically equal to V at least when all the involved types and kinds are well-kinded and well-formed, i.e. when $\Gamma \vdash X : K$, $\Gamma \vdash V : K$ and $k = |K|$. But this is not the case. The culprit is a mismatch of kind annotations in operator abstractions, as illustrated by the following counterexample.

Let $J_1 = \top\,..\,\top$ and $J_2 = *$ so that $\Gamma \vdash J_1 \leq J_2$ for any context Γ. Let $U = \top$, $V = \lambda X{:}J_2.\,U$ and $K = (X{:}J_1) \to *$ so that $\Gamma \vdash V : (X{:}J_2) \to *$, $\Gamma \vdash (X{:}J_2) \to * \leq K$ and hence $\Gamma \vdash V : K$. Then

$$
\begin{aligned}
\eta_K(Y) &\equiv \lambda Z{:}J_1.\,Y\,Z \\
\eta_K(Y)[Y^{|K|} := V] &\equiv (\lambda Z{:}J_1.\,Y\,Z)[Y^{|K|} := V] \\
&\equiv \lambda Z{:}J_1.\,(Y\,Z)[Y^{|K|} := V] && \text{(because } Y \notin \mathsf{fv}(J_1)\text{)} \\
&\equiv \lambda Z{:}J_1.\,V \cdot^{|K|} Z \\
&\equiv \lambda Z{:}J_1.\,\lambda X{:}J_2.\,U \cdot^{|J_1| \to *} Z \\
&\equiv \lambda Z{:}J_1.\,U[X^{|J_1|} := Z] \\
&\equiv \lambda X{:}J_1.\,U && \text{(as } X, Z \notin \mathsf{fv}(U)\text{)} \\
&\not\equiv \lambda X{:}J_2.\,U \equiv V. && \text{(because } J_1 \not\equiv J_2\text{)}
\end{aligned}
$$

So we are forced to conclude that $\eta_K(Y)[Y^{|K|} := V] \not\equiv V$ in general. The problem, as illustrated by this example, is that the domain annotation J_2 of the type operator abstraction $V = \lambda X{:}J_2.\,U$ is not necessarily preserved by the hereditary substitution. It is replaced by the domain J_1 of the declared kind $K = (X{:}J_1) \to U$ of Y, which need not be *syntactically* equal to J_1.

However, we do have $\Gamma \vdash J_1 \leq J_2$ and thus $|J_1| \equiv |J_2|$. The solution, therefore, is to be more lenient when comparing domain annotations in operator abstractions: the *weak* equation $\eta_K(Y)[Y^{|K|} := V] \approx V$ does hold. In fact, it holds for any simply well-formed kind K and simply well-kinded type U, as the following lemma shows.

Lemma 4.22.

1. *Let* $\gamma, X{:}\, j, \delta \vdash K\ \mathsf{kds}$, $\gamma \vdash U : j$ *and* $\gamma, X{:}\, j, \delta \vdash_{\mathsf{ne}} X\,\boldsymbol{V} : |K|$. *Then*

$$(\eta_K(X\,\boldsymbol{V}))[X^j := U] \;\approx\; (X\,\boldsymbol{V})[X^j := U].$$

Let $\gamma \vdash K\ \mathsf{kds}$*, then*

2. *if* $\gamma, X{:}|K|, \delta \vdash J\ \mathsf{kds}$*, then* $J[X^{|K|} := \eta_K(X)] \;\approx\; J$*;*
3. *if* $\gamma, X{:}|K|, \delta \vdash V : j$*, then* $V[X^{|K|} := \eta_K(X)] \;\approx\; V$*;*
4. *if* $\gamma, X{:}|K|, \delta \vdash j : \boldsymbol{V} : l$*, then* $\boldsymbol{V}[X^{|K|} := \eta_K(X)] \;\approx\; \boldsymbol{V}$*;*
5. *if* $K = (X{:}K_1) \to K_2$*,* $\gamma \vdash_{\mathsf{ne}} N : |K|$ *and* $\gamma \vdash V : |K_1|$*, then* $\eta_K(N) \cdot^{|K|} V \;\approx\; \eta_{K_2[X^{|K_1|} := V]}(N\,V)$*;*
6. *if* $\gamma \vdash_{\mathsf{ne}} N : |K|$ *and* $\gamma \vdash |K| : \boldsymbol{V} : *$*, then* $\eta_K(N) \cdot^{|K|} \boldsymbol{V} \;\approx\; N\,\boldsymbol{V}$*.*

Corollary 4.23. *If* $\gamma, X{:}\, j, \delta \vdash K\ \mathsf{kds}$ *and* $\gamma \vdash U : j$*, then* $(\eta_K(X))[X^j := U] \;\approx\; U$*.*

It is in this lemma that we see the true usefulness of weak equality. While syntactic equality is too strict for this particular commutativity property, using judgmental type and kind equality would have forced us to formulate its premises in terms of declarative kinding. This would have resulted in a weaker lemma with a more complicated proof. In the next chapter we will see that weak equations can be converted into judgmental ones provided the related types or kinds are well-kinded or well-formed, respectively. Hence, weak equality affords us a relatively straightforward proof of this lemma (and the next) with a minimal overhead in complexity.

In the proof of Lemma 4.22, we employ the following two helper lemmas. Both are proven by easy inductions: the first, on the structure of the kind K, for the second, on the derivation of $\gamma \vdash j : \boldsymbol{V} : k$.

Lemma 4.24. *Let* $X \neq Y$*, then* $(\eta_K(X\,\boldsymbol{D}))[Y^j := E] \;\equiv\; \eta_{K[Y^j := E]}(X\,(\boldsymbol{D}[Y^j := E]))$ *for any* K*,* $\boldsymbol{D}$*,* j *and* E*.*

Lemma 4.25. *Let* $\gamma \vdash U : j$*,* $\gamma \vdash j : \boldsymbol{V} : k$ *and* $\gamma \vdash k : \boldsymbol{W} : l$*, then* $U \cdot^j (\boldsymbol{V}, \boldsymbol{W}) \;\equiv\; (U \cdot^j \boldsymbol{V}) \cdot^k \boldsymbol{W}$*.*

Proof of Lemma 4.22. All six parts are proven simultaneously by induction on the structure of K. Parts 2–4 proceed by an inner induction on the simple formation and kinding derivations for J, V and $\boldsymbol{V}$, respectively. We show a few key cases, the remainder of the proof is routine.

- *Part 1,* $K = (Y{:}K_1) \to K_2$. By inspection of the formation and kinding rules, we must have $\gamma, X{:}\, j, \delta \vdash K_1\ \mathsf{kds}$, $\gamma, X{:}\, j, \delta, Y{:}|K_1| \vdash K_2\ \mathsf{kds}$ and $\gamma, X{:}\, j, \delta \vdash j : \boldsymbol{V} : |K|$. By Lemma 4.16 we have

$$\gamma, \delta \vdash (X\,\boldsymbol{V})[X^j := U] : |K_1| \to |K_2| \qquad \text{and} \qquad \gamma, \delta \vdash K_1[X^j := U]\ \mathsf{kds}.$$

The final kinding rule used to derive the first of these judgments must be SK-ABS since that is the only rule assigning simple arrow kinds to normal types. Therefore, the following must hold for some J and W:

$$(X\,\boldsymbol{V})[X^j := U] = \lambda Y{:}J.\,W \tag{4.1}$$
$$|K_1| = |J| \tag{4.2}$$
$$\gamma, \delta, Y{:}|J| \vdash W : |K_2|. \tag{4.3}$$

By weakening (Lemma 4.14), Lemma 4.18 and SK-NEAPP we also have

$$\gamma, X{:}\,j, \delta, Y{:}|K_1| \vdash_{\mathsf{ne}} X\,\boldsymbol{V}\,(\eta_{K_1}(Y)) : |K_2|$$

and hence

$$\begin{aligned}
&(\eta_{K_2}(X\,\boldsymbol{V}\,(\eta_{K_1}(Y))))[X^j := U] \\
&\quad\approx (X\,\boldsymbol{V}\,(\eta_{K_1}(Y)))[X^j := U] && \text{(by IH for 1)} \\
&\quad\equiv U \cdot^j ((\boldsymbol{V}, (\eta_{K_1}(Y)))[X^j := U]) && \text{(by definition)} \\
&\quad\equiv (U \cdot^j (\boldsymbol{V}[X^j := U])) \cdot^{|K|} ((\eta_{K_1}(Y))[X^j := U]) && \text{(by Lemmas 4.3 and 4.25)} \\
&\quad\equiv (U \cdot^j (\boldsymbol{V}[X^j := U])) \cdot^{|K|} (\eta_{K_1[X^j:=U]}(Y)) && \text{(by Lemma 4.24)} \\
&\quad\equiv ((X\,\boldsymbol{V})[X^j := U]) \cdot^{|K_1|\to|K_2|} (\eta_{K_1[X^j:=U]}(Y)) && \text{(by definition)} \\
&\quad\equiv (\lambda Y{:}J.\,W) \cdot^{|K_1|\to|K_2|} (\eta_{K_1[X^j:=U]}(Y)) && \text{(by (4.1))} \\
&\quad\equiv W[Y^{|K_1|} := (\eta_{K_1[X^j:=U]}(Y))] && \text{(by definition)} \\
&\quad\approx W && \text{(by Lemma 4.2, (4.3) and IH for 3)}
\end{aligned}$$

We conclude that

$$\begin{aligned}
&(\eta_K(X\,\boldsymbol{V}))[X^j := U] \\
&\quad\equiv \lambda Y{:}K_1[X^j := U].\,(\eta_{K_2}(X\,\boldsymbol{V}\,(\eta_{K_1}(Y))))[X^j := U] && \text{(by definition)} \\
&\quad\approx \lambda Y{:}J.\,W && \text{(by Lemma 4.2, (4.2), the above and WEQ-ABS)} \\
&\quad\equiv (X\,\boldsymbol{V})[X^j := U]. && \text{(by (4.1))}
\end{aligned}$$

- *Part 3, case* SK-NE. The rule SK-NE has only one premise which must have been derived using SK-VARAPP, so $V = N = Y\,\boldsymbol{V}$ and we have $\gamma, X{:}|K|, \delta \vdash j : \boldsymbol{V} : *$ with $(\gamma, X{:}|K|, \delta)(Y) = j$. We distinguish two cases: $Y = X$ and $Y \neq X$ but consider only the first case here; the second case is simpler. Since $Y = X$, we have $j = |K|$, and

$$\begin{aligned}
&V[X^{|K|} := \eta_K(X)] \\
&\quad\equiv \eta_K(X) \cdot^{|K|} (\boldsymbol{V}[X^{|K|} := \eta_K(X)]) && \text{(by definition)} \\
&\quad\approx \eta_K(X) \cdot^{|K|} \boldsymbol{V} && \text{(by WEQ-REFL, IH for 4 and Lemma 4.5.4)} \\
&\quad\approx X\,\boldsymbol{V}. && \text{(by SK-VARAPP and the IH for 6)}
\end{aligned}$$

- *Part 5,* $K = (X{:}K_1) \to K_2$. By inspection of the formation and kinding rules, we must have $\gamma \vdash K_1$ kds and $N = Y\,\boldsymbol{U}$ with $\gamma \vdash \gamma(Y) : \boldsymbol{U} : |K|$.

$$\begin{aligned}
\eta_K(N) \cdot^{|K|} V &\equiv \lambda X{:}K_1.\,\eta_{K_2}(Y\,\boldsymbol{U}\,(\eta_{K_1}(X))) \cdot^{|K|} V && \text{(by definition)}\\
&\equiv \eta_{K_2}(Y\,\boldsymbol{U}\,(\eta_{K_1}(X)))[X^{|K_1|} := V] && \text{(by definition)}\\
&\equiv \eta_{K_2[X^{|K_1|}:=V]}(Y\,((\boldsymbol{U},(\eta_{K_1}(X)))[X^{|K_1|} := V])) && \text{(by Lemma 4.24)}\\
&\equiv \eta_{K_2[X^{|K_1|}:=V]}(Y\,\boldsymbol{U}\,((\eta_{K_1}(X))[X^{|K_1|} := V])) && \text{(as } X \notin \mathsf{fv}(\boldsymbol{U})\text{)}\\
&\approx \eta_{K_2[X^{|K_1|}:=V]}(Y\,\boldsymbol{U}\,V). && \text{(by IH for 1 and Lemma 4.9)}
\end{aligned}$$

The use of the IH in the last step corresponds to Corollary 4.23. □

With Lemma 4.22 in place, we are ready to prove that normalization weakly commutes with substitution. In the following, $\Gamma \approx \Delta$ denotes the pointwise lifting of weak equality to contexts.

Lemma 4.26. *Substitution weakly commutes with normalization of well-formed kinds and well-kinded types. Let* $\Gamma \vdash A : J$ *and* $V = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$*, then*

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

Proof. Simultaneously by induction on declarative kind formation, kinding and context formation derivations. In the case for K-VAR where $B = Y$, we use Corollary 4.23 if $Y = X$; otherwise, we use the IH for part 3, Lemma 4.2, and Lemma 4.10 to derive $\mathsf{nf}_{\Gamma,\Delta[X:=A]}(Y) \approx \eta_{K'}(Y)$, where $K' = (\mathsf{nf}_{\mathsf{nf}(\Gamma,X:J,\Delta)}(K))[X^{|J|} := V]$. We conclude the case with Lemma 4.24. In the case for K-APP, we use Lemma 4.5.2 and Lemma 4.17.2. □

The very last lemma of this chapter will be used in our equivalence proof in the next chapter to show that subtyping rules for η-conversion of normal operators are admissible in canonical kinding.

Lemma 4.27. *If* $\Gamma \vdash A : (X{:}J) \to K$ *with* $X \notin \mathsf{fv}(A)$*, then* $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(\lambda X{:}J.\,A\,X) \approx \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$.

Proof. The proof uses Lemma 4.20 to obtain $|\mathsf{nf}(\Gamma)| \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma)}(A) : |J| \to |K|$ and proceeds by case analysis on the final rule used to derive this simple kinding judgment; the only applicable rule is SK-ABS. The remainder of the proof uses equational reasoning very similar to that used in the proof of Lemma 4.22.1. □

5 The canonical system

After having characterized the normal forms of kinds and types in the previous chapter, we now turn to the normalization – or rather, the canonization – of derivations of judgments relating such normal forms. In this chapter, we introduce a *canonical system* of judgments covering the kind and type level of $F^{\omega}_{..}$. We begin by describing the judgment forms and inference rules of the canonical system, comparing and contrasting them against their declarative counterparts.

In the remainder of the chapter, we develop the necessary metatheory to establish equivalence of the canonical and declarative presentations. As a first step, we prove soundness of the canonical system in Section 5.1. Next, we state and prove a *hereditary substitution lemma* in Section 5.2, which establishes that canonical judgments are preserved by hereditary substitutions. This lemma is a key ingredient in proving completeness of the canonical system. We prove completeness in Section 5.3, after showing that canonical subtyping can be inverted at the top-level in Section 5.4. We conclude the chapter by completing the type safety proof laid out at the end of Chapter 3.

Context well-formedness $\boxed{\Gamma\ \mathsf{ctx}}$

$$\frac{}{\varnothing\ \mathsf{ctx}}\ \text{(CC-Empty)} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash K\ \mathsf{kd}}{\Gamma, X{:}K\ \mathsf{ctx}}\ \text{(CC-TmBind)} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash V \rightrightarrows V\,..\,V}{\Gamma, x{:}V\ \mathsf{ctx}}\ \text{(CC-TpBind)}$$

Kind well-formedness $\boxed{\Gamma \vdash K\ \mathsf{kd}}$

$$\frac{\Gamma \vdash U \rightrightarrows U\,..\,U \qquad \Gamma \vdash V \rightrightarrows V\,..\,V}{\Gamma \vdash U\,..\,V\ \mathsf{kd}}\ \text{(CWf-Intv)} \qquad \frac{\Gamma \vdash J\ \mathsf{kd} \qquad \Gamma, X{:}J \vdash K\ \mathsf{kd}}{\Gamma \vdash (X{:}J) \to K\ \mathsf{kd}}\ \text{(CWf-DArr)}$$

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

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

Kinding of neutral types $\boxed{\Gamma \vdash_{\mathsf{ne}} N : K}$

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

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

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

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

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

Figure 5.1 – Canonical presentation of $F^{\omega}_{..}$ – part 1

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

$$\frac{\Gamma \vdash U_2 \leq U_1 \qquad \Gamma \vdash V_1 \leq V_2}{\Gamma \vdash U_1 \,..\, V_1 \leq U_2 \,..\, V_2} \text{ (CSK-INTV)}$$

$$\frac{\Gamma \vdash J_2 \leq J_1 \qquad \Gamma, X{:}J_2 \vdash K_1 \leq K_2 \qquad \Gamma \vdash (X{:}J_1) \to K_1 \ \mathsf{kd}}{\Gamma \vdash (X{:}J_1) \to K_1 \leq (X{:}J_2) \to K_2} \text{ (CSK-DARR)}$$

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

$$\frac{\Gamma \vdash V \rightrightarrows V \,..\, V}{\Gamma \vdash V \leq \top} \text{ (CST-TOP)} \qquad \frac{\Gamma \vdash V \rightrightarrows V \,..\, V}{\Gamma \vdash \bot \leq V} \text{ (CST-BOT)}$$

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

$$\frac{\Gamma \vdash_{\mathsf{var}} X : K \qquad \Gamma \vdash K \rightrightarrows \boldsymbol{V}_1 = \boldsymbol{V}_2 \rightrightarrows U \,..\, W}{\Gamma \vdash X\, \boldsymbol{V}_1 \leq X\, \boldsymbol{V}_2} \text{ (CST-NE)}$$

$$\frac{\Gamma \vdash K_2 \leq K_1 \qquad \Gamma, X{:}K_2 \vdash V_1 \leq V_2 \qquad \Gamma \vdash \forall X{:}K_1.\, V_1 \rightrightarrows \forall X{:}K_1.\, V_1 \,..\, \forall X{:}K_1.\, V_1}{\Gamma \vdash \forall X{:}K_1.\, V_1 \leq \forall X{:}K_2.\, V_2} \text{ (CST-ALL)}$$

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

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

$$\frac{\Gamma \vdash V_1 \leq V_2 \qquad \Gamma \vdash V_1 \leftleftarrows U \,..\, W \qquad \Gamma \vdash V_2 \leftleftarrows U \,..\, W}{\Gamma \vdash V_1 \leq V_2 \leftleftarrows U \,..\, W} \text{ (CST-INTV)}$$

$$\frac{\Gamma, X{:}J \vdash V_1 \leq V_2 \leftleftarrows K \qquad \Gamma \vdash \lambda X{:}J_1.\, V_1 \leftleftarrows (X{:}J) \to K \qquad \Gamma \vdash \lambda X{:}J_2.\, V_2 \leftleftarrows (X{:}J) \to K}{\Gamma \vdash \lambda X{:}J_1.\, V_1 \leq \lambda X{:}J_2.\, V_2 \leftleftarrows (X{:}J) \to K} \text{ (CST-ABS)}$$

Kind equality $\boxed{\Gamma \vdash J = K}$

$$\frac{\Gamma \vdash J \ \mathsf{kd} \qquad \Gamma \vdash K \ \mathsf{kd} \qquad \Gamma \vdash J \leq K \qquad \Gamma \vdash K \leq J}{\Gamma \vdash J = K} \text{ (CSK-ANTISYM)}$$

Type equality $\boxed{\Gamma \vdash U = V \leftleftarrows K}$

$$\frac{\Gamma \vdash K \ \mathsf{kd} \qquad \Gamma \vdash U \leq V \leftleftarrows K \qquad \Gamma \vdash V \leq U \leftleftarrows K}{\Gamma \vdash U = V \leftleftarrows K} \text{ (CST-ANTISYM)}$$

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

$$\frac{}{\Gamma \vdash K \rightrightarrows \epsilon = \epsilon \rightrightarrows K} \text{ (SPEQ-EMPTY)}$$

$$\frac{\Gamma \vdash U_1 = U_2 \leftleftarrows J \qquad \Gamma \vdash K[X^{|J|} := U_1] \rightrightarrows \boldsymbol{V}_1 = \boldsymbol{V}_2 \rightrightarrows L}{\Gamma \vdash (X{:}J) \to K \rightrightarrows U_1, \boldsymbol{V}_1 = U_2, \boldsymbol{V}_2 \rightrightarrows L} \text{ (SPEQ-CONS)}$$

Figure 5.2 – Canonical presentation of $F^{\omega}_{..}$ – part 2

Judgments. Figures 5.1 and 5.2 define the following judgments by mutual induction.

$\Gamma\ \mathsf{ctx}$	the context Γ is well-formed and normal
$\Gamma \vdash K\ \mathsf{kd}$	the kind K is well-formed and normal in Γ
$\Gamma \vdash J \leq K$	J is a normal subkind of K in Γ
$\Gamma \vdash J = K$	the normal kinds J and K are equal in Γ
$\Gamma \vdash_{\mathsf{var}} X : K$	the variable X has kind K in Γ
$\Gamma \vdash_{\mathsf{ne}} N : K$	the neutral type N has kind K in Γ
$\Gamma \vdash V \rightrightarrows K$	the normal type V has synthesized kind K in Γ
$\Gamma \vdash V \leftleftarrows K$	the normal type V kind checks against K in Γ
$\Gamma \vdash U \leq V$	U is a proper normal subtype of V in Γ
$\Gamma \vdash U \leq V \leftleftarrows K$	U is a normal subtype of V in K and Γ
$\Gamma \vdash U = V \leftleftarrows K$	U and V are equal normal types of kind K in Γ
$\Gamma \vdash J \rightrightarrows \boldsymbol{V} \rightrightarrows K$	applying an operator of kind J to the normal spine $\boldsymbol{V}$ yields a type of kind K in Γ.
$\Gamma \vdash J \rightrightarrows \boldsymbol{U} = \boldsymbol{V} \rightrightarrows K$	applying an operator of kind J to the equal spines $\boldsymbol{U}$ and $\boldsymbol{V}$ yields types of kind K in Γ.

Like the simplified formation and kinding judgments introduced in the previous chapter, the canonical judgments are defined directly on normal forms. But unlike the simplified judgments, the canonical judgments use the full expressivity of the kind language not only in their subjects, but also in their contexts and predicates. Like the declarative system, the canonical system therefore contains judgments for forming contexts and kinds, for kinding types, for subkinding and subtyping, and for identifying types and kinds.

The judgments for kinding and comparing normal types and spines are bidirectional. The double arrows $\rightrightarrows$ and $\leftleftarrows$ indicate whether a kind participating in such a judgment is considered an input or an output of the judgment. Kinds appearing at the tail end of a double arrow ($K \rightrightarrows$ or $\leftleftarrows K$) are inputs, i.e they are used to determine whether the judgment holds. Kinds appearing at the front end of a double arrow ($\rightrightarrows K$), on the other hand, are outputs, i.e they are uniquely determined by the inputs of the judgment. The contexts and subjects of judgments are always considered inputs.

The *kind synthesis* judgment $\Gamma \vdash V \rightrightarrows K$ assigns a unique kind K to a normal type V in the context Γ. We say that K is the synthesized kind of V in Γ. The rules of the kind synthesis judgment are similar to those of the declarative kinding judgment, except that the synthesized kinds are more precise (i.e. smaller w.r.t. the subtyping order), that there is no subsumption rule, and that the rules for kinding variables and applications have been combined into a single rule CK-SING for kinding neutral proper types. A quick inspection of the kind synthesis rules reveals that synthesized kinds are singletons. In particular, the synthesized kind of a proper type V is just the singleton interval $V \mathinner{..} V$.

Lemma 5.1. *If $\Gamma \vdash U \rightrightarrows V \mathinner{..} W$, then $V \equiv U$ and $W \equiv U$.*

The *kind checking* judgment $\Gamma \vdash V \leftleftarrows K$ has only one inference rule: the subsumption rule CK-SUB. This in contrast to other bidirectional systems where introduction forms, such as abstractions, typically lack type annotations, so that their kinds cannot be readily synthesized and must be checked instead. In $F^{\omega}_{\cdot\cdot}$, the only introduction forms at the type level are operator abstractions with Church-style kind annotations. Thanks to these annotations, the kinds of operator abstractions can be synthesized, making CK-SUB the only kind checking rule. The operational interpretation of CK-SUB is that, in order to check that a normal type V has kind K in a context Γ, we first synthesize the singleton kind J of V, then compare J and K to make sure the former is a subtype of the latter. If that is the case, we say that V kind checks against K, or simply that V is canonically well-kinded

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. This is due to the subsumption rule CV-SUB which allows us to widen the kind J of a variable to a superkind K of J. The additional premise $\Gamma \vdash K$ kd is a validity condition. The rule CV-SUB is not actually necessary for variable kinding, but as we will see in section Section 5.1, its presence considerably simplifies the proofs of the context narrowing and hereditary substitution lemmas for canonical judgments by rendering the former independent of the latter. Since the kind K is used in an input position in CV-SUB but in an output position in CV-VAR, variable kindings are neither kind synthesis nor kind checking judgments.

The *canonical spine kinding* judgment $\Gamma \vdash J \rightrightarrows \boldsymbol{V} \rightrightarrows K$ resembles its simple cousin from the previous chapter, except that both its input kind J and its output kind K may be fully dependent. This is reflected in the rule CK-CONS, where the head U of the spine $U, \boldsymbol{V}$ is hereditarily substituted for X in the codomain K of the overall input kind $(X{:}J) \to K$ to obtain the input kind $K[X^{|J|} := U]$ for kinding the remainder of the spine $\boldsymbol{V}$. The use of hereditary (rather than ordinary) substitution ensures that the resulting kind remains normal. To guarantee that U is a suitable substitute for X, it is first checked against the declared kind J of X in the first premise. The additional premise $\Gamma \vdash J$ kd is again a validity condition.

There is only one rule, CK-NE, for neutral kinding $\Gamma \vdash_{\mathsf{ne}} N : K$, which simply combines variable kinding and spine kinding. Since the overall kind of a neutral type is determined by the ambiguous kind of its head, the neutral kinding judgment is not directed either.

The canonical formation rules for contexts and kinds are almost identical to their declarative counterparts. The only difference is the form of the premises establishing a type V as a well-kinded proper type. We use a kind synthesis judgment of the form $\Gamma \vdash V \rightrightarrows V \mathrel{..} V$ for such premises rather than the perhaps more intuitive kind checking judgment $\Gamma \vdash V \leftleftarrows *$. The two are equivalent, as we will see in the next section, but a derivation of the latter always contains a derivation of the former as a strict sub-derivation, so we might as well pick the judgment with the smaller derivation to characterize proper types.

The rules for canonical subtyping are divided into two judgments: *subtyping of proper types* $\Gamma \vdash U \le V$ on one hand, and *checked* or *kind-directed subtyping* $\Gamma \vdash U \le V \leftleftarrows K$ on the other. As the name implies, the proper subtyping judgment $\Gamma \vdash U \le V$ relates only proper types,

and is therefore unkinded. It resembles the subtyping judgment of $F_{\leq}$ except for the lack of an explicit reflexivity rule, which we will prove admissible in the next section, and a variable subtyping rule, which is replaced by the more general bound projection rules CST-BND$_1$ and CST-BND$_2$ and the rule CST-NE for subtyping neutrals.[1]

The most interesting of these three rules is CST-NE. It says that two neutral types $X\,\boldsymbol{V}_1$ and $X\,\boldsymbol{V}_2$ headed by a common type variable X are subtypes if their spines are canonically equal, i.e. if they consist of point-wise equal normal types. Importantly, the spines $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ need not be syntactically equal: in $F^{\omega}_{\because}$, even normal types may be judgmentally equal yet differ syntactically. For example, two variables X and Y maybe equal if Y is of singleton kind $X \mathinner{..} X$, i.e. we have $\Gamma, X{:}*, Y{:}X \mathinner{..} X, \Delta \vdash X = Y : *$. And using inconsistent bounds, arbitrary pairs U, V of identically kinded normal types may be identified judgmentally, e.g. by bound projection, transitivity and antisymmetry, we have

$$\Gamma,\ X{:}\,U \mathinner{..} V,\ Y{:}\,V \mathinner{..} U,\ \Delta \vdash U\ =\ V\ :\ *$$

for any pair of proper types U and V. This last example illustrates that there is no easy way for the normalization function nf to resolve such equalities either. In the presence of higher-order types with inconsistent bounds, this would likely require some form of higher-order unification. In systems where types are identified up to β or $\beta\eta$-equality, judgmentally equal types have syntactically equal normal forms, so that the rule CST-NE can be replaced by a simple reflexivity rule (see e.g. [2]); in systems with singleton kinds (but no type intervals), declarations of the form $X : S(U : *)$ can be resolved during normalization by reducing X to U (see e.g. [49]). Neither of these techniques is applicable in $F^{\omega}_{\because}$, so we adopt the more flexible rule CST-NE instead. As we will see in Section 5.2, the presence of this rule considerably complicates the metatheory of the canonical system.

The checked subtyping judgment $\Gamma \vdash U \leq V \leftleftarrows K$ is kind-directed, i.e. the input kind K determines whether the types U and V being compared are proper types, in which case the rule CST-INTV applies, or whether they are of dependent arrow kind, in which case the rule CST-ABS applies. The rule CST-INTV checks the left- and right-hand types V_1, V_2 against the input kind $U \mathinner{..} W$, and compares them as proper types. Because normal types are η-long, the only normal types of dependent arrow kind are operator abstractions. The rule CST-ABS checks whether two such abstractions are subtypes in the common input kind $(X{:}J) \to K$ by checking them against that kind and comparing their bodies in the kind K under the additional assumption $X{:}J$. As in the declarative subtyping rule for operator abstractions, the domain annotations J_1 and J_2 are only used to kind check the respective abstractions, but not to compare their bodies.

The rules for canonical subkinding are identical to their declarative counterparts, except that CSK-INTV uses canonical proper subkinding to compare the bounds of the corresponding intervals.

[1] For comparison with subtyping in $F_{\leq}$, see e.g. Figs. 26-1 and 26-2 in [43].

Like their declarative counterparts, canonical equality judgments for kinds $\Gamma \vdash J = K$ and types $\Gamma \vdash U = V \Leftarrow K$ can only be derived through their respective antisymmetry rules CSK-AntiSym and CST-AntiSym. However, unlike the declarative versions, the canonical antisymmetry rules have explicit validity conditions. The additional premises $\Gamma \vdash J\ \mathsf{kd}$ and $\Gamma \vdash K\ \mathsf{kd}$ of CSK-AntiSym ensure that the left- and right-hand sides of the resulting kind equation are well-formed. The premise $\Gamma \vdash K\ \mathsf{kd}$ of CST-AntiSym, ensures that the input kind K of the judgment is well-formed. No explicit validity conditions are included for the left- and right-hand sides U, V of CST-AntiSym since suitable sub-premises are already present in the two checked subtyping premises $\Gamma \vdash U \leq V \Leftarrow K$ and $\Gamma \vdash V \leq U \Leftarrow K$ of the rule.

The inference rules of the canonical *spine equality* judgment $\Gamma \vdash J \rightrightarrows \boldsymbol{U} = \boldsymbol{V} \rightrightarrows K$ are similar to those of spine kinding. In the rule SpEq-Cons, we must again hereditarily substitute one of the heads of the spines $U_1, \boldsymbol{V}_1$ and $U_2, \boldsymbol{V}_2$ for X in the common codomain K to obtain a suitable input kind for comparing the tails $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$. In accordance with the declarative application rule ST-App, we pick U_1 as the substitute, knowing that we could just as well have chosen U_2.

Despite its name, the canonical system is not as canonical as one might expect; there are still quite a few sources of redundancy. Firstly, the variable and neutral kinding judgments are ambiguous. As discussed earlier, this ambiguity could be avoided by removing the rule CV-Sub at the cost of complicating the metatheory.

Secondly, several of the canonical rules have extra premises that act as validity conditions, much like those appearing in the extended declarative system presented in Chapter 3. These premises are redundant, and thus irrelevant for deriving canonical judgments, but they are useful when proving metatheoretic properties about the canonical system. It is likely that these validity conditions could be eliminated using the same approach as for the declarative system in Chapter 3.

Thirdly, the canonical proper subtyping judgment still features a transitivity rule CST-Trans. This rule is necessary because we cannot, in general, eliminate chains of subtyping judgments of the form $\Gamma \vdash U \leq X\,\boldsymbol{V} \leq W$. The culprit, as discussed earlier, are inconsistent bounds. If the declared kind of X has inconsistent bounds, we cannot eliminate the intermediate type $X\,\boldsymbol{V}$ and conclude $\Gamma \vdash U \leq W$ directly. We can, however, eliminate *top-level* uses of the transitivity rule CST-Trans, i.e. those where $\Gamma = \varnothing$. We will prove this claim in Section 5.4 where we also establish inversion of canonical subtyping for closed types.

5.1 Soundness and basic properties

Before we establish any other metatheoretic properties of the canonical system, let us prove its soundness with respect to the declarative presentation. To avoid confusion, we mark canonical judgments with the subscript "c" and declarative ones with "d" in the following lemma.

Lemma 5.2 (soundness of the canonical rules)**.**

1. *If* $\Gamma\ _{\mathsf{c}}\mathsf{ctx}$, *then* $\underline{\Gamma}\ _{\mathsf{d}}\mathsf{ctx}$.
2. *If* $\Gamma \vdash_{\mathsf{c}} K\ \mathsf{kd}$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{K}\ \mathsf{kd}$.
3. *If* $\Gamma \vdash_{\mathsf{c}} J \leq K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{J} \leq \underline{K}$.
4. *If* $\Gamma \vdash_{\mathsf{c}} J = K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{J} = \underline{K}$.
5. *If* $\Gamma \vdash_{\mathsf{var}} X : K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} X : \underline{K}$.
6. *If* $\Gamma \vdash_{\mathsf{ne}} N : K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{N} : \underline{K}$.
7. *If* $\Gamma \vdash_{\mathsf{c}} V \rightrightarrows K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{V} : \underline{K}$.
8. *If* $\Gamma \vdash_{\mathsf{c}} V \leftleftarrows K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{V} : \underline{K}$.
9. *If* $\Gamma \vdash_{\mathsf{c}} U \leq V$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{U} \leq \underline{V} : *$.
10. *If* $\Gamma \vdash_{\mathsf{c}} U \leq V : K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{U} \leq \underline{V} : \underline{K}$.
11. *If* $\Gamma \vdash_{\mathsf{c}} U = V : K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \underline{U} \leq \underline{V} : \underline{K}$.
12. *If* $\underline{\Gamma} \vdash_{\mathsf{d}} A : \underline{J}$ *and* $\Gamma \vdash_{\mathsf{c}} J \rightrightarrows \boldsymbol{V} \rightrightarrows K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \overline{A\,\underline{\boldsymbol{V}}} : \underline{K}$.
13. *If* $\underline{\Gamma} \vdash_{\mathsf{d}} A \leq B : \underline{J}$ *and* $\Gamma \vdash_{\mathsf{c}} J \rightrightarrows \boldsymbol{U} = \boldsymbol{V} \rightrightarrows K$, *then* $\underline{\Gamma} \vdash_{\mathsf{d}} \overline{A\,\underline{\boldsymbol{U}}} \leq \overline{B\,\underline{\boldsymbol{V}}} : \underline{K}$.

Proof. By induction on derivations of the various canonical judgments; for parts 1–11, on the derivation of the first premise, for parts 12 and 13, on that of the second premise. Most cases are straightforward. In cases involving synthesized kinding of proper types, we use K-SING and Lemmas 3.4, 3.20 and 5.1 to adjust kinds where necessary. In cases involving hereditary substitutions, i.e. those for CK-CONS and SPEQ-CONS, we use soundness of hereditary substitution in kinds (Corollary 4.7.1). □

As for the declarative system, the contexts of most canonical judgments are well-formed. There are two exceptions: kinding and equality of spines. The rules CK-EMPTY and SPEQ-EMPTY for empty spines offer no guarantee that the enclosing context is well-formed. This is not a problem in practice, since well-kinded spines only appear in judgments about neutral types, the heads of which must be kinded in a well-formed context.

Lemma 5.3 (context validity)**.** *Assume* $\Gamma \vdash \mathscr{J}$ *for any canonical judgment except spine kinding or equality. Then* $\Gamma\ \mathsf{ctx}$.

Proof. By simultaneous induction on the derivations of the various judgments. □

We can prove a few more validity properties at this point. Firstly, since synthesized kinds are singletons, validity of synthesized kinding judgments follows by CWF-INTV for proper types and by an easy induction for type operators. Secondly, validity of kind equality as well as the checked subtyping and type equality judgments follows immediately from the validity conditions included in the rules CSK-ANTISYM, CST-INTV, CST-ABS and CST-ANTISYM.

Lemma 5.4 (canonical validity – part 1)**.**

(synthesized kinding validity) *If* $\Gamma \vdash V \rightrightarrows K$, *then* $\Gamma \vdash K\ \mathsf{kd}$.

(checked subtyping validity) *If* $\Gamma \vdash U \leq V \leftleftarrows K$, *then* $\Gamma \vdash U \leftleftarrows K$ *and* $\Gamma \vdash V \leftleftarrows K$.

(kind equation validity) *If* $\Gamma \vdash J = K$, *then* $\Gamma \vdash J\ \mathsf{kd}$ *and* $\Gamma \vdash K\ \mathsf{kd}$.

(type equation validity) *If* $\Gamma \vdash U = V \leftleftarrows K$, *then* $\Gamma \vdash U \leftleftarrows K$, $\Gamma \vdash V \leftleftarrows K$ *and* $\Gamma \vdash K\ \mathsf{kd}$.

We prove the remaining validity properties of the canonical system in Section 5.2.1, once we have shown that hereditary substitutions preserve well-formedness of kinds.

Before we can do so, we need to establish the usual weakening and context narrowing lemmas for canonical typing.

Lemma 5.5 (weakening). *Assume* $\Gamma, \Delta \vdash \mathscr{J}$ *for any of the canonical judgments.*

1. *If* $\Gamma \vdash A \rightrightarrows A \mathrel{..} A$, *then* $\Gamma, x{:}A, \Delta \vdash \mathscr{J}$.
2. *If* $\Gamma \vdash K\ \mathsf{kd}$, *then* $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$.

Corollary 5.6 (iterated weakening). *If* $\Gamma, \Delta\ \mathsf{ctx}$ *and* $\Gamma \vdash \mathscr{J}$, *then* $\Gamma, \Delta \vdash \mathscr{J}$.

Lemma 5.7 (context narrowing – weak version).

1. *If* $\Gamma \vdash A \rightrightarrows A \mathrel{..} A$, $\Gamma \vdash A \leq B : *$ *and* $\Gamma, x{:}B, \Delta \vdash \mathscr{J}$, *then* $\Gamma, x{:}A, \Delta \vdash \mathscr{J}$.
2. *If* $\Gamma \vdash J\ \mathsf{kd}$, $\Gamma \vdash J \leq K$ *and* $\Gamma, X{:}K, \Delta \vdash \mathscr{J}$, *then* $\Gamma, X{:}J, \Delta \vdash \mathscr{J}$.

The proofs of both lemmas are routine inductions on the derivations of the respective judgments. The proof of context narrowing is only easy thanks to the rule CV-SUB. Without this rule, the canonical kinding judgments for variables and neutral types would become synthesis judgments and context narrowing would no longer hold in its present form for these two judgments. To see this, consider the variable kinding judgment $\Gamma, X{:}K, \Delta \vdash_{\mathsf{var}} X : K$, which, after eliminating CV-SUB, could only be derived using CV-VAR. If we were to narrow the context by changing the declared kind K of X to some J such that $\Gamma \vdash J \leq K$, then the synthesized kind of X would necessarily change to J too.

For neutral kinding, we would be in a similar situation as the new synthesized kind of the head would have to be propagated through the kinding derivation of the spine. Along the way, the new kind J would have to be unraveled by repeatedly separating $J = (X{:}J_1) \to J_2$ into J_1, J_2 and hereditarily substituting the next element of the spine into the codomain J_2. To prove context narrowing for the remainder of the canonical judgments, we would have to maintain the invariant that the new synthesized kind of a neutral type is a subtype of the original one, i.e. if $\Gamma, X{:}K_1, \Delta \vdash_{\mathsf{var}} N \rightrightarrows K_2$ and $\Gamma \vdash J_1 \leq K_1$ then $\Gamma, X{:}J_1, \Delta \vdash_{\mathsf{var}} N \rightrightarrows J_2$ and $\Gamma \vdash J_2 \leq K_2$. Because spine kinding involves hereditary substitution, a proof involving this invariant would require a hereditary substitution lemma for subtyping, i.e. a proof that hereditary substitutions preserve subtyping. We give such a proof in Section 5.2 and, as we will see there, it makes crucial use of context narrowing itself. By allowing us to establish Lemma 5.7 independently, the rule CV-SUB thus simplifies the proof of an otherwise rather complicated lemma (see Lemma 5.16 for details).

5.1.1 Order-theoretic properties

Having established context narrowing, we can prove the usual order-theoretic properties of canonical subkinding, subtyping, as well as kind and type equality. We start by stating an

proving the various reflexivity properties which, unlike those for the declarative relations, have to be proven simultaneously for the canonical variants.

Lemma 5.8. *The following reflexivity rules are all admissible.*

$$\frac{\Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash K \leq K}\ (\text{CSK-Refl}) \qquad \frac{\Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash K = K}\ (\text{CKEq-Refl})$$

$$\frac{\Gamma \vdash U \rightrightarrows V\,..\,W}{\Gamma \vdash U \leq U}\ (\text{CST-ReflSyn}) \qquad \frac{\Gamma \vdash V \leftleftarrows K \qquad \Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash V \leq V \leftleftarrows K}\ (\text{CST-ReflCk})$$

$$\frac{\Gamma \vdash V \rightrightarrows J \qquad \Gamma \vdash J \leq K \qquad \Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash V \leq V \leftleftarrows K}\ (\text{CST-ReflSub})$$

$$\frac{\Gamma \vdash J \rightrightarrows \boldsymbol{V} \rightrightarrows K}{\Gamma \vdash J \rightrightarrows \boldsymbol{V} = \boldsymbol{V} \rightrightarrows K}\ (\text{SpEq-Refl}) \qquad \frac{\Gamma \vdash V \leftleftarrows K \qquad \Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash V = V \leftleftarrows K}\ (\text{CTEq-Refl})$$

Proof. The proof is by mutual induction in the structure of the kinds and types being related to themselves, and then by case-analysis on the final rules of the corresponding kind formation and kinding derivations. The proof for CST-ReflSub proceeds by an inner induction on the derivation of $\Gamma \vdash J \leq K$. In the case for CSK-DArr where $V = \lambda X{:}J_1.U$ and we have $\Gamma \vdash K_1 \leq J_1$, $\Gamma, X{:}K_1 \vdash J_2 \leq K_2$ and $\Gamma, X{:}J_1 \vdash U \rightrightarrows J_2$, we use context narrowing to adjust the declared kind of X from J_1 to K_1 in $\Gamma, X{:}J_1 \vdash U \rightrightarrows J_2$ before applying the IH to obtain $\Gamma, X{:}K_1 \vdash U \leq U \leftleftarrows K_2$. □

Transitivity of the various relations and symmetry of the equalities are more easily established, thanks to the rule CST-Trans on the one hand, and to the structure of equality on the other.

Lemma 5.9. *Canonical subkinding, subtyping, kind and type equality are transitive.*

$$\frac{\Gamma \vdash J \leq K \qquad \Gamma \vdash K \leq L}{\Gamma \vdash J \leq L}\ (\text{CSK-Trans}) \qquad \frac{\Gamma \vdash J = K \qquad \Gamma \vdash K = L}{\Gamma \vdash J = L}\ (\text{CKEq-Trans})$$

$$\frac{\Gamma \vdash U \leq V \leftleftarrows K \qquad \Gamma \vdash V \leq W \leftleftarrows K}{\Gamma \vdash U \leq W \leftleftarrows K}\ (\text{CST-TransCk}) \qquad \frac{\Gamma \vdash U = V \leftleftarrows K \qquad \Gamma \vdash V = W \leftleftarrows K}{\Gamma \vdash U = W \leftleftarrows K}\ (\text{CTEq-Trans})$$

Proof. The proof of CSK-Trans is by induction on the structure of K and case analysis on the final rules used to derive the premises. In the case for $K = (X{:}K_1) \to K_2$ we use context narrowing. The proof of CST-Trans is by induction on the derivation of the first premise. The proofs of CKEq-Trans and CTEq-Trans are by inspection of the equality rules and use CSK-Trans and CST-Trans, respectively. □

Lemma 5.10. *Canonical kind and type equality are symmetric.*

$$\frac{\Gamma \vdash J = K}{\Gamma \vdash K = J} \;(\text{CKEQ-SYM}) \qquad \frac{\Gamma \vdash U = V \leftleftarrows K}{\Gamma \vdash V = U \leftleftarrows K} \;(\text{CTEQ-SYM})$$

Proof. By inspection of the equality rules, CSK-ANTISYM and CST-ANTISYM. □

Thanks to context narrowing and subkinding transitivity, we can prove admissibility of the following subsumption rules for the three checked judgments.

Lemma 5.11. *Kind subsumption is admissible in the checked judgments.*

$$\frac{\Gamma \vdash U \leftleftarrows J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash U \leftleftarrows K} \;(\text{CK-SUBCK}) \qquad \frac{\Gamma \vdash U \leq V \leftleftarrows J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash U \leq V \leftleftarrows K} \;(\text{CST-SUBCK})$$

$$\frac{\Gamma \vdash U = V \leftleftarrows J \qquad \Gamma \vdash J \leq K}{\Gamma \vdash U = V \leftleftarrows K} \;(\text{CTEQ-SUBCK})$$

Proof. Admissibility is proven separately for the three rules, in the order they are listed. The proof of CK-SUBCK is by inspection of the kind checking rules and uses CSK-TRANS. The proof of CST-SUBCK is by induction on the derivation of the second premise $\Gamma \vdash J \leq K$ and uses CK-SUBCK as well as context narrowing for the inductive step. The proof of CTEQ-SUBCK is by inspection of the equality rules and uses CST-SUBCK. □

Kind subsumption subsumes kind conversion thanks to the first of the following three rules, all of which follow immediately by inspection of the equality and checked subtyping rules.

$$\frac{\Gamma \vdash J = K}{\Gamma \vdash J \leq K} \;(\text{CSK-REFL-KEQ}) \qquad \frac{\Gamma \vdash U = V \leftleftarrows K}{\Gamma \vdash U \leq V \leftleftarrows K} \;(\text{CST-REFL-TEQ})$$

$$\frac{\Gamma \vdash U = V \leftleftarrows W\,..\,W'}{\Gamma \vdash U \leq V} \;(\text{CST-REFL-TEQ'})$$

5.1.2 Canonical replacements for declarative rules

Our completeness proof for the canonical system relies on the fact that the normal form $\mathsf{nf}(A)$ of any declaratively well-kinded type $\Gamma \vdash A : K$ kind checks against the normal form $\mathsf{nf}(K)$ of the corresponding kind K, i.e. that we have $\mathsf{nf}(\Gamma) \vdash \mathsf{nf}(A) \leftleftarrows \mathsf{nf}(K)$. To simplify the proof of this fact, we establish a set of admissible kind checking rules below that mirror the corresponding declarative rules.

We begin by proving that normal forms with synthesized or checked interval kinds also check against $*$.

Lemma 5.12. *Types inhabiting interval kinds are proper types. If* $\Gamma \vdash U \rightrightarrows V\,..\,W$ *or* $\Gamma \vdash U \leftleftarrows V\,..\,W$*, then also* $\Gamma \vdash U \leftleftarrows *$.

Proof. If $\Gamma \vdash U \rightrightarrows V\,..\,W$, then by Lemma 5.1, $\Gamma \vdash U \rightrightarrows U\,..\,U$. From this we derive the result by CST-BOT, CST-TOP, CSK-INTV and CK-SUB. If $\Gamma \vdash U \leftleftarrows V\,..\,W$, then this must have been derived using CK-SUB and hence $\Gamma \vdash U \rightrightarrows K$ and $\Gamma \vdash K \leq V\,..\,W$. By inspection of the subkinding rules, $K = V'\,..\,W'$ for some V' and W'. The result then follows by the first part of the lemma. □

The second part of the proof follows a pattern that is is quite typical for proofs in the remainder of this chapter. Thanks to the division of kinding into kind synthesis and checking, and thanks to the simplicity of both kind checking and subkinding derivations, we can often prove properties of kind checking judgments $\Gamma \vdash U \leftleftarrows K$ by appealing to similar properties of kind synthesis judgments $\Gamma \vdash U \rightrightarrows K'$ where K and K' have the same shape, i.e. where K and K' are either both intervals or both arrows. Two more examples of this pattern appear in the following lemma, where the admissibility proofs of the rules CST-CKBND$_1$ and CST-CKBND$_2$, which have kind checking judgments as their premises, appeal to instances of CST-SYNBND$_1$ and CST-SYNBND$_2$, respectively, which have similar kind synthesis judgments as their premises.

Lemma 5.13. *All of the following are admissible.*

$$\frac{\Gamma \vdash U \leftleftarrows V\,..\,W}{\Gamma \vdash U \rightrightarrows U\,..\,U}\ (\text{CK-SING'}) \qquad \frac{\Gamma \vdash V \rightrightarrows K}{\Gamma \vdash V \leftleftarrows K}\ (\text{CK-SYNCK}) \qquad \frac{\Gamma \vdash_{\mathsf{ne}} N : U\,..\,V}{\Gamma \vdash N \leftleftarrows U\,..\,V}\ (\text{CK-NECK})$$

$$\frac{\Gamma \vdash U \leftleftarrows * \qquad \Gamma \vdash V \leftleftarrows *}{\Gamma \vdash U \to V \leftleftarrows *}\ (\text{CK-ARR'}) \qquad \frac{\Gamma \vdash K\ \mathsf{kd} \qquad \Gamma, X{:}K \vdash V \leftleftarrows *}{\Gamma \vdash \forall X{:}K.\, V \leftleftarrows *}\ (\text{CK-ALL'})$$

$$\frac{\Gamma \vdash J\ \mathsf{kd} \qquad \Gamma, X{:}J \vdash V \leftleftarrows K}{\Gamma \vdash \lambda X{:}J.\, V \leftleftarrows (X{:}J) \to K}\ (\text{CK-ABS'}) \qquad \frac{\Gamma \vdash V_1 \leq V_2 \leftleftarrows U\,..\,W}{\Gamma \vdash V_1 \leq V_2 \leftleftarrows V_1\,..\,V_2}\ (\text{CST-INTV'})$$

$$\frac{\Gamma \vdash U \rightrightarrows V_1\,..\,V_2}{\Gamma \vdash V_1 \leq U}\ (\text{CST-SYNBND}_1) \qquad \frac{\Gamma \vdash U \rightrightarrows V_1\,..\,V_2}{\Gamma \vdash U \leq V_2}\ (\text{CST-SYNBND}_2)$$

$$\frac{\Gamma \vdash U \leftleftarrows V_1\,..\,V_2}{\Gamma \vdash V_1 \leq U}\ (\text{CST-CKBND}_1) \qquad \frac{\Gamma \vdash U \leftleftarrows V_1\,..\,V_2}{\Gamma \vdash U \leq V_2}\ (\text{CST-CKBND}_2)$$

$$\frac{\Gamma \vdash J \rightrightarrows \boldsymbol{U} \rightrightarrows (X{:}K) \to L \qquad \Gamma \vdash V \leftleftarrows K \qquad \Gamma \vdash K\ \mathsf{kd}}{\Gamma \vdash J \rightrightarrows \boldsymbol{U}, V \rightrightarrows L[X^{|K|} := V]}\ (\text{CK-SNOC})$$

$$\frac{\Gamma \vdash J \rightrightarrows \boldsymbol{U}_1 = \boldsymbol{U}_2 \rightrightarrows (X{:}K) \to L \qquad \Gamma \vdash V_1 = V_2 \leftleftarrows K}{\Gamma \vdash J \rightrightarrows \boldsymbol{U}_1, V_1 = \boldsymbol{U}_2, V_2 \rightrightarrows L[X^{|K|} := V_1]}\ (\text{SPEQ-SNOC})$$

Proof. Some of the rules are derivable others are admissible; most of the proofs are straightforward, so we omit the details. The proofs of the alternate type formation rules use CK-SING' and Lemma 5.12. The proofs of the alternate projection rules CST-SYNBND$_1$ and CST-SYNBND$_2$ use Lemma 5.1 and reflexivity; those of CST-CKBND$_1$ and CST-CKBND$_2$ are by inspection of kind checking and subkinding and use CST-SYNBND$_1$ and CST-SYNBND$_2$, respectively. The proofs of the last two rules are by induction on the derivations of the respective first premises. □

As in the declarative system, we define *canonical context equality* $\Gamma = \Delta \; \mathsf{ctx}$ as the pointwise lifting of canonical type and kind equality to context bindings:

$$\frac{}{\varnothing = \varnothing \; \mathsf{ctx}} \qquad \frac{\Gamma = \Delta \; \mathsf{ctx} \qquad \Gamma \vdash J = K}{\Gamma, X{:}J = \Delta, X{:}K \; \mathsf{ctx}} \qquad \frac{\Gamma = \Delta \; \mathsf{ctx} \qquad \Gamma \vdash U = V \leftleftarrows W \,..\, W'}{\Gamma, x{:}U = \Delta, x{:}V \; \mathsf{ctx}}$$

5.1.3 Simplification of canonical kinding

In Section 4.2.2 of the previous chapter, we showed that every well-formed kind and well-kinded type has a simply well-formed or simply well-kinded normal form, respectively (see Lemma 4.20). Canonically well-formed kinds and canonically well-kinded types are already in normal form, but we can still simplify their kind formation and kinding derivations, as the following pair of lemma shows. In the statement of the second lemma, we use the subscript "nes" to mark simple kinding judgments for neutral types and "ne" to mark their canonical counterparts.

Lemma 5.14. *Canonical subkinds and equal kinds simplify equally. If $\Gamma \vdash J \leq K$ or $\Gamma \vdash J = K$ then $|J| \equiv |K|$.*

Proof. Separately, by induction on subkinding and kind equality derivations, respectively. □

Lemma 5.15 (simplification)**.** *Well-formed kinds and well-kinded normal forms, neutrals and spines are also simply well-formed and well-kinded, respectively.*

1. *If $\Gamma \vdash_{\mathsf{var}} X : K$, then $|\Gamma| \vdash_{\mathsf{nes}} X : |K|$.*
2. *If $\Gamma \vdash K \; \mathsf{kd}$, then $|\Gamma| \vdash K \; \mathsf{kds}$.*
3. *If $\Gamma \vdash_{\mathsf{ne}} N : K$, then $|\Gamma| \vdash_{\mathsf{nes}} N : |K|$.*
4. *If $\Gamma \vdash V \rightrightarrows K$, then $|\Gamma| \vdash V : |K|$.*
5. *If $\Gamma \vdash V \leftleftarrows K$, then $|\Gamma| \vdash V : |K|$.*
6. *If $\Gamma \vdash J \rightrightarrows \boldsymbol{V} \rightrightarrows K$, then $|\Gamma| \vdash |J| : \boldsymbol{V} : |K|$.*

Proof. Part 1 is proven separately, parts 2–6 are proven simultaneously, all by induction on the derivations of the respective premises. The cases of CV-SUB and CK-SUB use of Lemma 5.14. □

Thanks to Lemma 5.15, properties of simply kinded normal forms still hold for canonically kinded normal forms. For example, by Lemmas 5.15 and 4.17, hereditary substitutions in

canonically kinded types commute.

5.2 The hereditary substitution lemma

We have arrived at the core of the technical development of this chapter: the proof of the *hereditary substitution lemma*. The hereditary substitution lemma states, roughly, that canonical judgments are preserved by hereditary substitutions of canonically well-kinded types. Just as the ordinary substitution lemma for the declarative system (Lemma 3.9) played a key role in the proofs of several metatheoretic properties in Chapter 3, the hereditary substitution lemma is key to proving important metatheoretic properties of the canonical system. But unlike that of its ordinary counterpart, the proof of the hereditary substitution lemma is rather challenging. This is reflected already in the statement of the lemma, which features 24 separate parts, all of which have to be proven simultaneously (see Lemma 5.16 below).

One reason for the large number of parts is simply that there are more judgment forms in the canonical system than there are in the declarative system. But the foremost reason is that the proof of the hereditary substitution lemma circularly depends on *functionality of the canonical judgments*, i.e. on the fact that hereditarily substituting canonically equal types in normal forms yields canonically equal normal forms. This also renders the proof more challenging since both properties have to be established at the same time.

The main source of complexity is the subtyping rule CST-NE. It is because of this rule that we have to prove the hereditary substitution and functionality lemmas simultaneously.

To illustrate this, consider the neutral types $N = X\,\boldsymbol{V}$ and $M = X\,\boldsymbol{W}$ with $\boldsymbol{V} = V_1, V_2$, $\boldsymbol{W} = W_1, W_2$ such that $X \notin \mathsf{fv}(\boldsymbol{V}) \cup \mathsf{fv}(\boldsymbol{W})$, and assume some Γ, Δ and U such that

$$\Gamma, \Delta \vdash \lambda Y_1{:}J_1.\, \lambda Y_2{:}J_2.\, U \leftleftarrows J \qquad \text{and} \qquad \Gamma, X{:}J, \Delta \vdash J \rightrightarrows \boldsymbol{V} = \boldsymbol{W} \rightrightarrows *$$

for $J = (Y_1{:}J_1) \to (Y_2{:}J_2) \to J_3$. Then, by CST-NE, we have $\Gamma, X{:}J, \Delta \vdash N \leq M$.

We would like to hereditarily substitute $U' = \lambda Y_1{:}J_1.\, \lambda Y_2{:}J_2.\, U$ for X in N and M and show that the resulting types remain subtypes, i.e. that

$$\Gamma, \Delta[X^{|J|} := U'] \vdash N[X^{|J|} := U'] \leq M[X^{|J|} := U'].$$

By the definition of hereditary substitution, we have

$$\begin{aligned}
N[X^{|J|} := U'] &\equiv (X\,\boldsymbol{V})[X^{|J|} := U'] \equiv U' \cdot^{|J|} (\boldsymbol{V}[X^{|J|} := U']) \\
&\equiv U' \cdot^{|J|} V_1 \cdot^{|J_2| \to |J_3|} V_2 \\
&\equiv ((\lambda Y_2{:}J_2.\, U)[Y_1{}^{|J_1|} := V_1]) \cdot^{|J_2| \to |J_3|} V_2 \\
&\equiv (\lambda Y_2{:}J_2[Y_1{}^{|J_1|} := V_1].\, U[Y_1{}^{|J_1|} := V_1]) \cdot^{|J_2| \to |J_3|} V_2 \\
&\equiv V'[Y_2{}^{|J_2|} := V_2]
\end{aligned}$$

where $V' = U[Y_1{}^{|J_1|} := V_1]$. Similarly, $M[X^{|J|} := U'] \equiv W'[Y_2{}^{|J_2|} := W_2]$ for $W' = U[Y_1{}^{|J_1|} := W_1]$. Hence, we need to show that

$$\Gamma, \Delta[X^{|J|} := U'] \vdash U[Y_1{}^{|J_1|} := V_1] \leq U[Y_1{}^{|J_1|} := W_1] \quad \text{and}$$
$$\Gamma, \Delta[X^{|J|} := U'] \vdash V'[Y_2{}^{|J_2|} := V_2] \leq W'[Y_2{}^{|J_2|} := W_2].$$

Since they belong to equal spines, V_1 and W_1 are judgmentally equal as types, and so are V_2 and W_2. But in general, neither of these pairs of types are syntactically equal, i.e. $V_1 \not\equiv W_1$, $V_2 \not\equiv W_2$ and $V' \not\equiv W'$. To establish the above inequations, we therefore need to show that simultaneous hereditary substitutions of judgmentally equal types preserve inequations.

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.

Lemma 5.16 (hereditary substitution)**.** *Hereditary substitutions of canonically kind-checked types preserve the canonical judgments; substitutions of canonically equal types in canonically well-formed and well-kinded expressions result in canonical equations; substitutions of canonically equal types preserve canonical (in)equations; kinding and subtyping of reducing applications is admissible. Assume that the following equations hold*

$$\Gamma \vdash U_1 = U_2 \leftleftarrows K \qquad \Gamma, \Sigma = \Gamma, \Delta[X^{|K|} := U_1]\,\mathsf{ctx} \qquad \Gamma, \Sigma = \Gamma, \Delta[X^{|K|} := U_2]\,\mathsf{ctx}$$

for given Γ, Δ, Σ, U_1, U_2, K *and* $X \notin \mathsf{dom}(\Gamma, \Delta, \Sigma)$.

1. *If* $\Gamma, X{:}K, \Delta \vdash J\,\mathsf{kd}$, *then* $\Gamma, \Sigma \vdash J[X^{|K|} := U_1]\,\mathsf{kd}$.
2. *If* $\Gamma, X{:}K, \Delta \vdash_{\mathsf{var}} X : J$, *then* $\Gamma, \Sigma \vdash U_1 \leftleftarrows J[X^{|K|} := U_1]$.
3. *If* $\Gamma, X{:}K, \Delta \vdash_{\mathsf{var}} Y : J$ *and* $Y \neq X$, *then* $\Gamma, \Sigma \vdash_{\mathsf{var}} Y : J[X^{|K|} := U_1]$.
4. *If* $\Gamma, X{:}K, \Delta \vdash_{\mathsf{ne}} N : V \mathinner{..} W$, *then* $\Gamma, \Sigma \vdash N[X^{|K|} := U_1] \leftleftarrows (V \mathinner{..} W)[X^{|K|} := U_1]$.
5. *If* $|\Gamma| \vdash J\,\mathsf{kds}$ *and* $\Gamma, X{:}K, \Delta \vdash J \rightrightarrows \boldsymbol{V} \rightrightarrows L$, *then*

$$\Gamma, \Sigma \vdash J[X^{|K|} := U_1] \rightrightarrows \boldsymbol{V}[X^{|K|} := U_1] \rightrightarrows L[X^{|K|} := U_1].$$

6. *If* $\Gamma, X{:}K, \Delta \vdash V \rightrightarrows J$, *then* $\Gamma, \Sigma \vdash V[X^{|K|} := U_1] \rightrightarrows J[X^{|K|} := U_1]$.
7. *If* $\Gamma, X{:}K, \Delta \vdash V \leftleftarrows J$, *then* $\Gamma, \Sigma \vdash V[X^{|K|} := U_1] \leftleftarrows J[X^{|K|} := U_1]$.
8. *If* $\Gamma, X{:}K, \Delta \vdash J\,\mathsf{kd}$, *then* $\Gamma, \Sigma \vdash J[X^{|K|} := U_1] \leq J[X^{|K|} := U_2]$.
9. *If* $\Gamma, X{:}K, \Delta \vdash J\,\mathsf{kd}$, *then* $\Gamma, \Sigma \vdash J[X^{|K|} := U_1] = J[X^{|K|} := U_2]$.
10. *If* $\Gamma, X{:}K, \Delta \vdash_{\mathsf{var}} X : J$, *then* $\Gamma, \Sigma \vdash U_1 = U_2 \leftleftarrows J[X^{|K|} := U_1]$.
11. *If* $\Gamma, X{:}K, \Delta \vdash_{\mathsf{ne}} N : V \mathinner{..} W$, *then* $\Gamma, \Sigma \vdash N[X^{|K|} := U_1] \leq N[X^{|K|} := U_2]$.
12. *If* $|\Gamma| \vdash J\,\mathsf{kds}$ *and* $\Gamma, X{:}K, \Delta \vdash J \rightrightarrows \boldsymbol{V} \rightrightarrows L$, *then*

$$\Gamma, \Sigma \vdash J[X^{|K|} := U_1] \rightrightarrows \boldsymbol{V}[X^{|K|} := U_1] = \boldsymbol{V}[X^{|K|} := U_2] \rightrightarrows L[X^{|K|} := U_1].$$

13. *If* $\Gamma, X{:}K, \Delta \vdash V \rightrightarrows W \mathinner{..} W'$, *then* $\Gamma, \Sigma \vdash V[X^{|K|} := U_1] \leq V[X^{|K|} := U_2]$.

14. *If* $\Gamma, X{:}K, \Delta \vdash V \rightrightarrows J$ *and* $\Gamma, X{:}K, \Delta \vdash J$ kd, *then*

$$\Gamma, \Sigma \vdash V[X^{|K|} := U_1] \leq V[X^{|K|} := U_2] \leftleftarrows J[X^{|K|} := U_1].$$

15. *If* $\Gamma, X{:}K, \Delta \vdash V \leftleftarrows J$ *and* $\Gamma, X{:}K, \Delta \vdash J$ kd, *then*

$$\Gamma, \Sigma \vdash V[X^{|K|} := U_1] \leq V[X^{|K|} := U_2] \leftleftarrows J[X^{|K|} := U_1].$$

16. *If* $\Gamma, X{:}K, \Delta \vdash J_1 \leq J_2$, *then* $\Gamma, \Sigma \vdash J_1[X^{|K|} := U_1] \leq J_2[X^{|K|} := U_2]$.
17. *If* $|\Gamma| \vdash J$ kds *and* $\Gamma, X{:}K, \Delta \vdash J \rightrightarrows \boldsymbol{V}_1 = \boldsymbol{V}_2 \rightrightarrows L$, *then*

$$\Gamma, \Sigma \vdash J[X^{|K|} := U_1] \rightrightarrows \boldsymbol{V}_1[X^{|K|} := U_1] = \boldsymbol{V}_2[X^{|K|} := U_2] \rightrightarrows L[X^{|K|} := U_1].$$

18. *If* $\Gamma, X{:}K, \Delta \vdash V_1 \leq V_2$, *then* $\Gamma, \Sigma \vdash V_1[X^{|K|} := U_1] \leq V_2[X^{|K|} := U_2]$.
19. *If* $\Gamma, X{:}K, \Delta \vdash V_1 \leq V_2 \leftleftarrows J$ *and* $\Gamma, X{:}K, \Delta \vdash J$ kd, *then*

$$\Gamma, \Sigma \vdash V_1[X^{|K|} := U_1] \leq V_2[X^{|K|} := U_2] \leftleftarrows J[X^{|K|} := U_1].$$

20. *If* $\Gamma, X{:}K, \Delta \vdash V_1 = V_2 \leftleftarrows J$, *then* $\Gamma, \Sigma \vdash V_1[X^{|K|} := U_1] = V_2[X^{|K|} := U_2] \leftleftarrows J[X^{|K|} := U_1]$.
21. *If* $\Gamma \vdash K \rightrightarrows \boldsymbol{V} \rightrightarrows J$, *then* $\Gamma \vdash U_1 \cdot^{|K|} \boldsymbol{V} \leftleftarrows J$.
22. *If* $K = (X{:}K_1) \to K_2$, $\Gamma \vdash V \leftleftarrows K_1$ *and* $\Gamma \vdash K_1$ kd, *then* $\Gamma \vdash U_1 \cdot^{|K|} V \leftleftarrows K_2[X^{|K_1|} := V]$.
23. *If* $\Gamma \vdash K \rightrightarrows \boldsymbol{V}_1 = \boldsymbol{V}_2 \rightrightarrows J$, *then* $\Gamma \vdash U_1 \cdot^{|K|} \boldsymbol{V}_1 = U_2 \cdot^{|K|} \boldsymbol{V}_2 \leftleftarrows J$.
24. *If* $K = (X{:}K_1) \to K_2$ *and* $\Gamma \vdash V_1 = V_2 \leftleftarrows K_1$, *then*

$$\Gamma \vdash U_1 \cdot^{|K|} V_1 = U_1 \cdot^{|K|} V_2 \leftleftarrows K_2[X^{|K_1|} := V_1].$$

Proof. As for the proof of Lemma 4.16, 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 simple kind $|K|$. Parts 1–20 proceed by an inner induction on the respective formation, kinding, subkinding, subtyping or equality derivations of the expressions in which U_1 and U_2 are being substituted for X. Parts 21–24 proceed by a case analysis on the final rule used to derive $\Gamma \vdash K \rightrightarrows \boldsymbol{V} \rightrightarrows J$, $\Gamma \vdash U_1 \leftleftarrows K$, $\Gamma \vdash K \rightrightarrows \boldsymbol{V}_1 = \boldsymbol{V}_2 \rightrightarrows J$ and $\Gamma \vdash U_1 = U_2 \leftleftarrows K$, respectively.

The proofs of parts 1–7 are similar to that of the declarative substitution lemma (Lemma 3.9), while those of parts 8–20 resemble the proof of the extended functionality lemma (Lemma 3.19). In cases like CWF-DARR or CK-ALL, where the context is extended by an additional binding, we use the IH together with context narrowing (Lemma 5.7) to maintain the invariants $\Gamma, \Sigma = \Gamma, \Delta[X^{|K|} := U_1]$ ctx and $\Gamma, \Sigma = \Gamma, \Delta[X^{|K|} := U_2]$ ctx.

The cases where the proofs of parts 1–20 differ most substantially from those of Lemma 3.9 and Lemma 3.19 are parts 4, 11 and the case for CST-NE of part 18, which deal with neutral types. There, we proceed by case distinction on $X = Y$, where Y is the head of the corresponding neutral types. If $X = Y$, then we proceed using either part 21, or part 23 followed by CST-REFL-TEQ'. If $X \neq Y$, then we use part 3 and proceed with either part 5 followed by CK-NECK, or

with parts 12 or 17 followed by CST-NE. In the cases for CST-BND$_1$ and CST-BND$_2$, we use part 4 followed by CST-CKBND$_1$ or CST-CKBND$_1$.

In the cases for CK-CONS and SPEQ-CONS of parts 5, 12 and 17, respectively, where $J = (Y{:}J_1) \to J_2$, we use Lemma 4.17.1 to show that hereditary substitutions in kinds commute, i.e. that $J_2[Y^{|J_1|} := V_1][X^{|K|} := U_1] \equiv J_2[X^{|K|} := U_1][Y^{|J_1|} := V_1[X^{|K|} := U_1]]$. The necessary simple kinding derivations are provided by case analysis of the final rule used to derive the premise $|\Gamma| \vdash J$ kds and Lemma 5.15.5.

The proofs of parts 22 and 24 resemble that of Lemma 4.16.5 but are complicated slightly by the presence of subkinding. We show the proof of part 22, that of part 24 is similar. We have $K = (X{:}K_1) \to K_2$, $\Gamma \vdash U_1 \leftleftarrows K$, $\Gamma \vdash V \leftleftarrows K_1$ and $\Gamma \vdash K_1$ kd, and we want to show that $\Gamma \vdash U_1 \cdot^{|K|} V \leftleftarrows K_2[X^{|K_1|} := V]$. By inspection of the kind checking and subkinding rules, we must have $U_1 = \lambda X{:}J_1.U$ such that $\Gamma, X{:}J_1 \vdash U \rightrightarrows J_2$, $\Gamma \vdash K_1 \leq J_1$ and $\Gamma, X{:}K_1 \vdash J_2 \leq K_2$, and by the definition of reducing application, $U_1 \cdot^{|K|} V \equiv U[X^{|K_1|} := V]$. Using context narrowing and the IH for part 7, we obtain $\Gamma \vdash U[X^{|K_1|} := V] \rightrightarrows J_2[X^{|K_1|} := V]$. By TEQ-REFL and the IH for part 16, we have $\Gamma \vdash J_2[X^{|K_1|} := V] \leq K_2[X^{|K_1|} := V]$. We conclude by CK-SUB. □

5.2.1 Validity

With the hereditary substitution lemma in place, we can now prove the remaining validity properties of the canonical judgments. The most intricate cases are those for spine kinding and equality, which is where we use Lemma 5.16.

Lemma 5.17 (canonical validity – part 2)**.**

(spine kinding validity)	*If* $\Gamma \vdash J$ kd *and* $\Gamma \vdash J \rightrightarrows \boldsymbol{U} \rightrightarrows K$, *then* $\Gamma \vdash K$ kd.
(spine equation validity)	*If* $\Gamma \vdash J_1 \leq J_2$ *and* $\Gamma \vdash J_2 \rightrightarrows \boldsymbol{U} = \boldsymbol{V} \rightrightarrows K_2$, *then* $\Gamma \vdash J_2 \rightrightarrows \boldsymbol{U} \rightrightarrows K_2$, $\Gamma \vdash J_1 \rightrightarrows \boldsymbol{V} \rightrightarrows K_1$ *and* $\Gamma \vdash K_1 \leq K_2$ *for some* K_1.
(neutral kinding validity)	*If* $\Gamma \vdash N : K$, *then* $\Gamma \vdash K$ kd.
(subkinding validity)	*If* $\Gamma \vdash J \leq K$, *then* $\Gamma \vdash J$ kd *and* $\Gamma \vdash K$ kd.
(proper subtyping validity)	*If* $\Gamma \vdash U \leq V$, *then* $\Gamma \vdash U \rightrightarrows U\,..\,U$ *and* $\Gamma \vdash V \rightrightarrows V\,..\,V$.
(checked kinding validity)	*If* $\Gamma \vdash V \leftleftarrows K$, *then* $\Gamma \vdash K$ kd.

Proof. Subkinding and proper subtyping validity are proven simultaneously, the remaining parts are proven separately, in the order they are listed. All parts are proven by induction on derivations of the judgments they are named after: spine kinding and equation validity are proven by induction on their respective second premises, the remaining parts on their respective first premises. In inductive steps of the proofs of spine kinding and equation validity, we use the hereditary substitution lemmas to derive suitable first premises for applying the IH. The proof of spine equation validity relies on checked equation validity from Lemma 5.17. The proof of neutral kinding validity relies on spine kinding validity. In the proof of proper subtyping validity, we use neutral kinding validity in the cases for the bound projection rules CST-BND$_{1,2}$. The proof of checked kinding validity relies on proper subtyping validity. □

5.2.2 Lifting of weak equality to canonical equality

In Section 4.2.2 of the previous chapter, we established a number of weak commutativity properties (see Lemmas 4.22 and 4.26). Among others, we showed that normalization weakly commutes with hereditary substitution. But up until now, we do not have any effective means to put these properties to use – we have yet to establish a relationship between weak equality and the equality judgments of the declarative and canonical systems.

To remedy this situation, we prove that a weak equation $U \approx V$ can be lifted to canonical equation $\Gamma \vdash U = V \leftleftarrows K$, provided the left- and right-hand sides U, V are well-kinded, i.e. $\Gamma \vdash U \leftleftarrows K$ and $\Gamma \vdash V \leftleftarrows K$. Similarly, we show that weakly equal kinds are canonically equal if they are well-formed.

Lemma 5.18. *Weakly equal canonically well-formed kinds and well-kinded types are canonically equal.*

1. *If* $\Gamma \vdash J\ \mathsf{kd}$, $\Gamma \vdash K\ \mathsf{kd}$ *and* $J \approx K$, *then* $\Gamma \vdash J \leq K$.
2. *If* $\Gamma \vdash U \rightrightarrows U\,..\,U$, $\Gamma \vdash V \rightrightarrows V\,..\,V$ *and* $U \approx V$, *then* $\Gamma \vdash U \leq V$.
3. *If* $\Gamma \vdash K\ \mathsf{kd}$, $\Gamma \vdash U \leftleftarrows K$, $\Gamma \vdash V \leftleftarrows K$ *and* $U \approx V$, *then* $\Gamma \vdash U \leq V \leftleftarrows K$.
4. *If* $\Gamma \vdash J\ \mathsf{kd}$, $\Gamma \vdash J \leq K_1$, $\Gamma \vdash J \leq K_2$, $\Gamma \vdash K_1 \rightrightarrows \boldsymbol{U}_1 \rightrightarrows V_1\,..\,W_1$, $\Gamma \vdash K_2 \rightrightarrows \boldsymbol{U}_2 \rightrightarrows V_2\,..\,W_2$, *and* $\boldsymbol{U}_1 \approx \boldsymbol{U}_2$, *then* $\Gamma \vdash J \rightrightarrows \boldsymbol{U}_1 = \boldsymbol{U}_2 \rightrightarrows V\,..\,W$ *for some* V *and* W.
5. *If* $\Gamma \vdash J\ \mathsf{kd}$, $\Gamma \vdash K\ \mathsf{kd}$ *and* $J \approx K$, *then* $\Gamma \vdash J = K$.
6. *If* $\Gamma \vdash K\ \mathsf{kd}$, $\Gamma \vdash U \leftleftarrows K$, $\Gamma \vdash V \leftleftarrows K$ *and* $U \approx V$, *then* $\Gamma \vdash U = V \leftleftarrows K$.

Proof. Simultaneously for all 6 parts by simultaneous induction on the corresponding pairs of kinds, types or spines being related, then by case analysis on the final rules used to derive the corresponding formation, kinding and weak equality judgments. In the proof of part 5, we apply the IH for part 1 directly to the equations $J \approx K$ and $K \approx J$, where the latter is derived using symmetry of weak equality (Lemma 4.1). Neither J nor K decrease in this step, but the proof of part 5 does not make any further use of the IH and could therefore be inlined in the proofs of the other parts. The proof of part 5 is similar.

The proofs of the remaining parts are largely routine. The most interesting case is that of WEQ-CONS in part 4, where we have $\boldsymbol{U}_1 = (U_1, \boldsymbol{U}'_1)$, $\boldsymbol{U}_2 = (U_2, \boldsymbol{U}'_2)$, $K_1 = (X{:}K_{11}) \to K_{12}$, $K_2 = (X{:}K_{21}) \to K_{22}$, $U_1 \approx U_2$ and $\boldsymbol{U}'_1 \approx \boldsymbol{U}'_2$. Analyzing the derivations of the remaining premises, we have

$$\Gamma \vdash K_{11} \leq J_1 \qquad \Gamma, X{:}K_{11} \vdash J_2 \leq K_{12} \qquad \Gamma \vdash K_{21} \leq J_1 \qquad \Gamma, X{:}K_{21} \vdash J_2 \leq K_{22}$$

such that $J = (X{:}J_1) \to J_2$, as well as

$$\Gamma \vdash U_1 \leftleftarrows K_{11} \qquad \Gamma \vdash K_{12}[X^{|K_{11}|} := U_1] \rightrightarrows \boldsymbol{U}'_1 \rightrightarrows V_1\,..\,W_1$$
$$\Gamma \vdash U_2 \leftleftarrows K_{21} \qquad \Gamma \vdash K_{22}[X^{|K_{21}|} := U_2] \rightrightarrows \boldsymbol{U}'_2 \rightrightarrows V_2\,..\,W_2$$

We use CK-SUBCK and the IH for part 6 to derive $\Gamma \vdash U_1 = U_2 \leftleftarrows J_1$, then we use hereditary

substitution (Lemmas 5.16.16 and 5.16.8) and Lemma 5.14 to derive

$$\Gamma \vdash J_2[X^{|J_1|} := U_1] \leq K_{12}[X^{|K_{11}|} := U_1]$$
$$\Gamma \vdash J_2[X^{|J_1|} := U_1] \leq J_2[X^{|J_1|} := U_2] \leq K_{22}[X^{|K_{21}|} := U_2]$$

We conclude the case by the IH for part 4 and SpEq-Cons. □

5.3 Completeness of canonical kinding

In the previous chapter, we saw that every declaratively well-formed kind or well-kinded type has a judgmentally equal $\beta\eta$-normal form (Lemmas 4.12 and 4.20). In this section, we prove that every declarative judgment has a canonical counterpart where the expressions related by the original judgment have been normalized. Roughly, whenever $\Gamma \vdash_{\mathsf{d}} \mathscr{J}$ holds, we also have $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}_{\mathsf{nf}(\Gamma)}(\mathscr{J})$. Since normalization does not change the meaning of an expression, this result establishes completeness of the canonical system w.r.t. to the declarative one.

There are several judgments for kinding types in the canonical system, but only one in the declarative system. To establish completeness, we show that, if a type A is of kind K according to declarative kinding, then the normal form $\mathsf{nf}(A)$ kind checks against the normal form $\mathsf{nf}(K)$, i.e. if $\Gamma \vdash_{\mathsf{d}} A : K$, then $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) \leftleftarrows \mathsf{nf}(K)$.

When A is a variable $A = X$, the normal form $\mathsf{nf}(A)$ is its η-expansion $\mathsf{nf}(A) = \eta_{\mathsf{nf}(K)}(X)$ and we use the following lemma to prove that it kind checks against $\mathsf{nf}(K)$.

Lemma 5.19 (η-expansion)**.** *η-expansion preserves the canonical kinds of neutral types. If $\Gamma \vdash_{\mathsf{ne}} N : K$, then $\Gamma \vdash \eta_K(N) \leftleftarrows K$.*

Instead of proving the lemma directly, we first prove the following helper lemma.

Lemma 5.20.

1. *If $\Gamma, X{:}J \vdash K$ kd and $\Gamma, X{:}J \vdash \eta_J(X) \leftleftarrows J$, then $\Gamma, X{:}J \vdash K[X^{|J|} := \eta_J(X)] = K$.*
2. *If $\Gamma \vdash_{\mathsf{ne}} N : J$ and $\Gamma \vdash J \leq K$, then $\Gamma \vdash \eta_K(N) \leftleftarrows K$.*

The first part says that hereditary substitutions of η-expanded variables in kinds vanish, while the second part is a strengthened version of Lemma 5.19.

Proof. The two parts are proven separately. For the first part, we use simplification of canonical kinding (Lemma 5.15.2) and Lemma 4.22.2 to derive $K[X^{|J|} := \eta_J(X)] \approx K$. By weakening and the hereditary substitution lemma (Lemma 5.16.1), we have $\Gamma, X{:}J \vdash K[X^{|J|} := \eta_J(X)]$ kd. The conclusion of the first part then follows by Lemma 5.18.5.

The proof of the second part is by induction on the structure of K and case analysis on the final rule used to derive $\Gamma \vdash J \leq K$. In the base case, we use CK-NeCk and CK-SubCk. In the

inductive case, we have $K = (X{:}K_1) \to K_2$, $J = (X{:}J_1) \to J_2$ such that $\Gamma \vdash K_1 \leq J_1$ and $\Gamma, X{:}K_1 \vdash J_2 \leq K_2$. By subkinding validity, we further have $\Gamma \vdash K_1$ kd and $\Gamma, X{:}K_1 \vdash K_2$ kd. By weakening, the IH and CSK-REFL, we obtain first $\Gamma, X{:}K_1 \vdash \eta_{K_1}(X) \leftleftarrows K_1$, then $\Gamma, X{:}K_1 \vdash \eta_{K_1}(X) \leftleftarrows J_1$ by weakening and CK-SUBCK. By inspection of neutral kinding, we know that $N = Y\,\boldsymbol{V}$ for some Y and $\boldsymbol{V}$ and that $\Gamma \vdash_{\mathsf{var}} Y : L$ and $\Gamma \vdash L \rightrightarrows \boldsymbol{V} \rightrightarrows (X{:}J_1) \to J_2$. We use weakening, CK-SNOC and CK-NE to derive $\Gamma, X{:}K_1 \vdash_{\mathsf{ne}} Y\,\boldsymbol{V}\,(\eta_{K_1}(X)) : J_2[X^{|J_1|} := \eta_{K_1}(X)]$.

Now we see why it was necessary to strengthen the IH: the body of the η-expansion of N has kind $J_2[X^{|J_1|} := \eta_{K_1}(X)]$ rather than K_2 as required by Lemma 5.19. In order to apply the IH, we show that

$$
\begin{aligned}
\Gamma, X{:}K_1 \vdash J_2[X^{|J_1|} := \eta_{K_1}(X)] &\equiv J_2[X^{|K_1|} := \eta_{K_1}(X)] && \text{(by Lemma 5.14)}\\
&\leq K_2[X^{|K_1|} := \eta_{K_1}(X)] && \text{(by Lemma 5.16.16)}\\
&= K_2. && \text{(by part 1)}
\end{aligned}
$$

We conclude the case by the IH and CK-ABS'. □

Lemma 5.19 as well as a strengthened version of Lemma 5.20.1 now follow as corollaries.

Corollary 5.21. *If* $\Gamma, X{:}J \vdash K$ kd, *then* $\Gamma, X{:}J \vdash K[X^{|J|} := \eta_J(X)] = K$.

To establish completeness of the canonical system w.r.t. the declarative system, we show that every declarative judgment derived using the *extended* declarative rules, rather than the original ones, has a canonical counterpart. The proof makes crucial use of the validity conditions present in the extended rules. To avoid confusion, we again mark canonical judgments with the subscript “c” and extended declarative ones with “e”. To enhance readability, we omit the subscript $\mathsf{nf}(\Gamma)$, writing e.g. $\mathsf{nf}(A)$ instead of $\mathsf{nf}_{\mathsf{nf}(\Gamma)}(A)$.

Lemma 5.22 (completeness of the canonical rules – extended version)**.**

1. *If* $\Gamma\ {}_{\mathsf{e}}\mathsf{ctx}$, *then* $\mathsf{nf}(\Gamma)\ {}_{\mathsf{c}}\mathsf{ctx}$.
2. *If* $\Gamma \vdash_{\mathsf{e}} K$ kd, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(K)$ kd.
3. *If* $\Gamma \vdash_{\mathsf{e}} A : K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) \leftleftarrows \mathsf{nf}(K)$.
4. *If* $\Gamma \vdash_{\mathsf{e}} J \leq K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(J) \leq \mathsf{nf}(K)$.
5. *If* $\Gamma \vdash_{\mathsf{e}} A \leq B : C\,..\,D$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) \leq \mathsf{nf}(B)$.
6. *If* $\Gamma \vdash_{\mathsf{e}} A \leq B : K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) \leq \mathsf{nf}(B) \leftleftarrows \mathsf{nf}(K)$.
7. *If* $\Gamma \vdash_{\mathsf{e}} J = K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(J) = \mathsf{nf}(K)$.
8. *If* $\Gamma \vdash_{\mathsf{e}} A = B : K$, *then* $\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(A) = \mathsf{nf}(B) \leftleftarrows \mathsf{nf}(K)$.
9. *If* $\Gamma \vdash_{\mathsf{e}} A : (X{:}J) \to K$ *and* $X \notin \mathsf{fv}(A)$, *then*

$$\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(\lambda X{:}J.\,A\,X) = \mathsf{nf}(A) \leftleftarrows \mathsf{nf}((X{:}J) \to K).$$

10. *If* $\Gamma, X{:}J \vdash_{\mathsf{e}} K$ kd, $\Gamma \vdash_{\mathsf{e}} A : J$ *and* $\Gamma \vdash_{\mathsf{e}} K[X := A]$ kd, *then*

$$\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(K)[X^{|\mathsf{nf}(J)|} := \mathsf{nf}(A)] = \mathsf{nf}(K[X := A]).$$

11. *If* $\Gamma, X{:}J \vdash_{\mathsf{e}} A : K$*,* $\Gamma \vdash_{\mathsf{e}} B : J$*,* $\Gamma, X{:}J \vdash_{\mathsf{e}} K\ \mathsf{kd}$*,* $\Gamma \vdash_{\mathsf{e}} A[X := B] : K[X := B]$ *and* $\Gamma \vdash_{\mathsf{e}} K[X := B]\ \mathsf{kd}$*, then*

$$\mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}((\lambda X{:}J.\, A)\, B) = \mathsf{nf}(A[X := B]) \Leftarrow \mathsf{nf}(K[X := B]).$$

Equivalent statements w.r.t. the original declarative rules follow by equivalence of the original and extended declarative systems.

Proof. All parts are proven simultaneously, by induction on the derivations of the respective premises, except for parts 9–11, which are helper lemmas that apply the IH directly to all of their premises but could be inlined in the proofs of the other parts.

Thanks to the admissible rules introduced in Lemma 5.13, the proofs of parts 1–3 are straightforward, except for the cases of K-VAR, where we use Lemma 5.19, and K-EXTAPP, where we use the hereditary substitution lemma to normalize $A = A_1\, A_2$ if $\mathsf{nf}(A_1)$ is an abstraction, and the IH for part 10 together with CK-SUBCK to adjust the kind of the result. The validity conditions of K-EXTAPP are crucial in this last step.

Parts 5 and 8 follow almost immediately from part 6, part 7 from part 4.

The proofs of parts 9–11 all follow the same pattern. First, we use the IH to normalize the premises and establish validity of the left- and right-hand sides of the respective equations. Then we use Lemma 4.27, Lemma 4.26.1 and Lemma 4.26.2, respectively, to derive weak versions of these equations, and Lemma 5.18 to turn them into canonical equations.

The remaining parts 4 and 6 are the most difficult to prove. The cases of the extended β and η-conversion rules are covered by parts 9 and 11 thanks to the validity conditions in the extended rules. The case of ST-EXTAPP is similar to that of K-EXTAPP – again the validity conditions are crucial. Some of the remaining cases are covered by the admissible rules introduced in Sections 5.1.1 and 5.1.2. The challenging cases are those where one of the premises of the corresponding rule extends the contexts, i.e. those of CSK-DARR, CST-ALL and CST-ABS. We show the case for CSK-DARR here, the other two are similar.

We are given $\Gamma \vdash_{\mathsf{e}} K_1 \le J_1$, $\Gamma, X{:}K_1 \vdash_{\mathsf{e}} J_2 \le K_2$ and $\Gamma \vdash_{\mathsf{e}} (X{:}J_1) \to J_2\ \mathsf{kd}$ with $J = (X{:}J_1) \to J_2$ and $K = (X{:}K_1) \to K_2$. We start by applying the IH to all the premises and analyze the last of the resulting derivations to obtain

$$\begin{array}{ll} \mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(K_1) \le \mathsf{nf}(J_1) & \mathsf{nf}(\Gamma), X{:}\mathsf{nf}(K_1) \vdash_{\mathsf{c}} \mathsf{nf}_{\mathsf{nf}(\Gamma, X:K_1)}(J_2) \le \mathsf{nf}_{\mathsf{nf}(\Gamma, X:K_1)}(K_2) \\ \mathsf{nf}(\Gamma) \vdash_{\mathsf{c}} \mathsf{nf}(J_1)\ \mathsf{kd} & \mathsf{nf}(\Gamma), X{:}\mathsf{nf}(J_1) \vdash_{\mathsf{c}} \mathsf{nf}_{\mathsf{nf}(\Gamma, X:J_1)}(J_2)\ \mathsf{kd}. \end{array}$$

Note the different contexts $\mathsf{nf}(\Gamma, X{:}K_1)$ and $\mathsf{nf}(\Gamma, X{:}J_1)$ used to normalize J_2 in the second and fourth of these judgments, respectively. This leads to a syntactic difference in the resulting normal forms, i.e. we have $\mathsf{nf}_{\mathsf{nf}(\Gamma, X:K_1)}(J_2) \not\equiv \mathsf{nf}_{\mathsf{nf}(\Gamma, X:J_1)}(J_2)$. In order to apply CSK-DARR, we need to resolve this difference.

Transitivity-free subtyping of closed proper types $\boxed{\vdash_{\mathsf{tf}} U \leq V}$

$$\frac{\varnothing \vdash V \rightrightarrows V\,..\,V}{\vdash_{\mathsf{tf}} V \leq \top} \quad (\text{TFST-TOP}) \qquad \frac{\varnothing \vdash V \rightrightarrows V\,..\,V}{\vdash_{\mathsf{tf}} \bot \leq V} \quad (\text{TFST-BOT})$$

$$\frac{\varnothing \vdash U_2 \leq U_1 \qquad \varnothing \vdash V_1 \leq V_2}{\vdash_{\mathsf{tf}} U_1 \to V_1 \leq U_2 \to V_2} \quad (\text{TFST-ARR})$$

$$\frac{\varnothing \vdash K_2 \leq K_1 \qquad X{:}K_2 \vdash V_1 \leq V_2 \qquad \varnothing \vdash \forall X{:}K_1.\,V_1 \rightrightarrows \forall X{:}K_1.\,V_1\,..\,\forall X{:}K_1.\,V_1}{\vdash_{\mathsf{tf}} \forall X{:}K_1.\,V_1 \leq \forall X{:}K_2.\,V_2} \quad (\text{TFST-ALL})$$

Figure 5.3 – Top-level transitivity-free canonical subtyping

We notice that $|\mathsf{nf}(\Gamma, X{:}J_1)| \equiv |\mathsf{nf}(\Gamma, X{:}K_1)|$ by Lemma 5.14. Hence, by Lemma 4.10, we have $\mathsf{nf}_{\mathsf{nf}(\Gamma,X:J_1)}(J_2) \approx \mathsf{nf}_{\mathsf{nf}(\Gamma,X:K_1)}(J_2)$. Using context narrowing and subkinding validity, we derive

$$\mathsf{nf}(\Gamma), X{:}\mathsf{nf}(K_1) \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma,X:J_1)}(J_2)\ \mathsf{kd} \qquad \text{and} \qquad \mathsf{nf}(\Gamma), X{:}\mathsf{nf}(K_1) \vdash \mathsf{nf}_{\mathsf{nf}(\Gamma,X:K_1)}(J_2)\ \mathsf{kd},$$

from which we obtain, by Lemma 5.18.1,

$$\mathsf{nf}(\Gamma), X{:}\mathsf{nf}(K_1) \vdash_{\mathsf{c}} \mathsf{nf}_{\mathsf{nf}(\Gamma,X:J_1)}(J_2) = \mathsf{nf}_{\mathsf{nf}(\Gamma,X:K_1)}(J_2) \leq \mathsf{nf}_{\mathsf{nf}(\Gamma,X:K_1)}(K_2)$$

We conclude the case by CWF-DARR and CSK-DARR. □

5.4 Inversion of canonical subtyping

As we saw in Chapter 3, preservation of types under CBV reduction does not hold in arbitrary contexts. The culprit are type variable bindings with inconsistent bounds. Such bindings can inject arbitrary inequations into the subtyping relation and hence break putative properties that hold for subtyping of closed types. For example, the absurd assumption $X : \top\,..\,\bot$ trivializes the subtyping relation under any context in which it appears. To see this, consider the following derivation, where $\Gamma = X : \top\,..\,\bot$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash U \to V \leq \top} \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash_{\mathsf{ne}} X : \top\,..\,\bot}}{\Gamma \vdash \top \leq X}\ (\text{CST-BND}_1) \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash_{\mathsf{ne}} X : \top\,..\,\bot}}{\Gamma \vdash X \leq \bot}\ (\text{CST-BND}_2)}{\Gamma \vdash \top \leq \bot}\ (\text{CST-TRANS})}{\Gamma \vdash U \to V \leq \bot}\ (\text{CST-TRANS}) \quad \dfrac{\vdots}{\Gamma \vdash \bot \leq \forall X{:}K.\,W}}{\Gamma \vdash U \to V \leq \forall X{:}K.\,W}\ (\text{CST-TRANS})$$

Under such conditions, subtyping cannot be inverted in any meaningful way. We therefore consider inversion of canonical subtyping only in the empty context, following the approach taken by Rompf and Amin in their type safety proof for DOT [45]

As a first step we show that any top-level uses of the transitivity rule CST-TRANS can be eliminated. To do so, we introduce a helper judgment $\vdash_{\mathsf{tf}} U \leq V$, which states that U is a proper subtype of V in the empty context (see Fig. 5.3). It is easy to see that this judgment is sound w.r.t. canonical subtyping in the empty context (the proof is by routine induction on subtyping derivations).

Lemma 5.23 (soundness of top-level subtyping)**.** *If* $\vdash_{\mathsf{tf}} U \leq V$*, then* $\varnothing \vdash U \leq V$.

Crucially, the inference rules for the judgment $\vdash_{\mathsf{tf}} U \leq V$ do not include a transitivity rule, but the following variant of that rule is admissible.

Lemma 5.24 (top-level transitivity elimination)**.** *The following is admissible.*

$$\frac{\varnothing \vdash U \leq V \qquad \vdash_{\mathsf{tf}} V \leq W}{\vdash_{\mathsf{tf}} U \leq W}\ (\text{TFST-TRANS})$$

Proof. The proof is by induction on the derivation of the first premise and case analysis of the final rule used to derive the second. In the case of CST-TRANS, we use the IH twice. In the case of CST-BOT where $\varnothing \vdash \bot \leq U$ and $\vdash_{\mathsf{tf}} U \leq V$, we use Lemma 5.23 and validity of canonical typing to derive $\varnothing \vdash V \rightrightarrows V \mathinner{..} V$ and conclude with TFST-BOT. Similarly, in cases where the second premise was derived using TFST-TOP, we use validity of canonical subtyping and TFST-TOP. □

Thanks to TFST-TRANS, it is straightforward to establish completeness, and thus equivalence of the judgments $\varnothing \vdash U \leq V$ and $\vdash_{\mathsf{tf}} U \leq V$.

Lemma 5.25 (equivalence of top-level subtyping)**.** *The two versions of canonical subtyping are equivalent in the empty context:* $\varnothing \vdash U \leq V$ *iff* $\vdash_{\mathsf{tf}} U \leq V$.

Proof. We have already proven soundness ($\Leftarrow$). Completeness ($\Rightarrow$) is by induction on the derivations of $\varnothing \vdash U \leq V$ and uses TFST-TRANS in the case of CST-TRANS. □

Inversion of the canonical subtyping relation in the empty context now follows immediately by inspection of the transitivity-free subtyping rules and Lemma 5.25. We only state the relevant cases.

Corollary 5.26 (inversion of canonical subtyping – embedding)**.** *Let* $\varnothing \vdash U_1 \leq U_2$.

1. *If* $U_1 = V_1 \to W_1$ *and* $U_2 = V_2 \to W_2$*, then* $\varnothing \vdash V_2 \leq V_1$ *and* $\varnothing \vdash W_1 \leq W_2$.
2. *If* $U_1 = \forall X{:}K_1.\, V_1$ *and* $U_2 = \forall X{:}K_2.\, V_2$*, then* $\varnothing \vdash K_2 \leq K_1$ *and* $X{:}K_2 \vdash V_1 \leq V_2$.

Corollary 5.27 (inversion of canonical subtyping – contradiction)**.** *Closed arrows are not canonical subtypes of closed universals and vice-versa. For any* U, V, W *and* K*, we have*

1. $\varnothing \vdash U \to V \not\leq \forall X{:}K.\, W$*, and*
2. $\varnothing \vdash \forall X{:}K.\, U \not\leq V \to W$.

5.5 Type safety revisited

We are finally ready to prove type safety of $F^{\omega}_{\cdot\cdot}$. We start by proving the declarative version of subtyping inversion. Again, we only state the relevant cases.

Lemma 5.28 (inversion of declarative subtyping – embedding). *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* $X{:}K_2 \vdash B_1 \leq B_2 : *$.

The proof makes use of the following generation lemma for well-kinded arrow and universal types, which is proven by induction on kinding derivations.

Lemma 5.29 (generation of kinding for arrows and universals). *The following are admissible.*

$$\frac{\Gamma \vdash A \to B : *}{\Gamma \vdash A : * \qquad \Gamma \vdash B : *} \qquad\qquad \frac{\Gamma \vdash \forall X{:}K.\, A : *}{\Gamma \vdash K \;\mathsf{kd} \qquad \Gamma, X{:}K \vdash A : *}$$

Proof of Lemma 5.28. By completeness of canonical subtyping and soundness of normalization. We show only the first part, the second is analogous. Assume $\varnothing \vdash B_1 \to C_1 \leq B_2 \to C_2 : *$. Then by validity of declarative subtyping (Lemma 3.17), generation of kinding for arrow types, soundness of normalization (Lemma 4.12) and completeness of canonical subtyping, we have

$$\begin{array}{ll} \varnothing \vdash_{\mathsf{d}} B_1 = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(B_1) : * & \varnothing \vdash_{\mathsf{d}} C_1 = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(C_1) : * \\ \varnothing \vdash_{\mathsf{d}} B_2 = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(B_2) : * & \varnothing \vdash_{\mathsf{d}} C_2 = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(C_2) : * \\ \multicolumn{2}{l}{\varnothing \vdash_{\mathsf{c}} \mathsf{nf}_{\mathsf{nf}(\Gamma)}(B_1) \to \mathsf{nf}_{\mathsf{nf}(\Gamma)}(C_1) \leq \mathsf{nf}_{\mathsf{nf}(\Gamma)}(B_2) \to \mathsf{nf}_{\mathsf{nf}(\Gamma)}(C_2)} \end{array}$$

By inversion and soundness of canonical subtyping, it follows that

$$\begin{array}{l} \varnothing \vdash_{\mathsf{d}} B_2 = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(B_2) \leq \mathsf{nf}_{\mathsf{nf}(\Gamma)}(B_1) = B_1 : * \quad \text{and} \\ \varnothing \vdash_{\mathsf{d}} C_1 = \mathsf{nf}_{\mathsf{nf}(\Gamma)}(C_1) \leq \mathsf{nf}_{\mathsf{nf}(\Gamma)}(C_2) = C_2 : *. \end{array}$$

□

We also prove a declarative counterpart of Corollary 5.27, which is used in the proof of the progress theorem below.

Lemma 5.30 (inversion of declarative subtyping – contradiction). *Closed arrows are not subtypes of closed universals and vice-versa. For any* A, B, C *and* K, *we have*

1. $\varnothing \vdash A \to B \not\leq \forall X{:}K.\, C$, *and*
2. $\varnothing \vdash \forall X{:}K.\, A \not\leq B \to C$.

Proof. By completeness of canonical subtyping, then by contradiction using Corollary 5.27. □

This is all we need to prove weak preservation. Recall that weak preservation (Theorem 3.43) states that CBV reduction preserves the types of closed terms, i.e. if $\vdash t : A$ and $t \longrightarrow_{\mathsf{v}} t'$, then $\vdash t' : A$.

Proof of Theorem 3.43. The proof is by induction on typing derivations and case analysis on CBV reduction rules. The interesting cases are those where β-contractions occur. We revisit the case of T-APP when the reduction step is an instance of R-APPABS. The case for T-TAPP with R-TAPPTABS is similar. We have $t = (\lambda x{:}B.\, s)\ v$ with $\vdash \lambda x{:}B.\, s : C \to A$ and $\vdash v : C$ for some B and C. By generation of term abstractions (Lemma 3.44.1), $x{:}B \vdash s : D$ and $\vdash B \to D \leq C \to A : *$ for some D. By inversion of subtyping (Lemma 5.28.2), we have $\vdash C \leq B : *$ and $\vdash D \leq A : *$ and hence $\vdash v : B$ and $x{:}B \vdash s : A$ by subsumption. To conclude the proof we need to show that $\vdash s[x := v] : A$, which follows from the substitution lemma (Lemma 3.9). □

This establishes the first half of type safety. For the second half, progress, we first need to prove a standard canonical forms lemma.

Lemma 5.31 (canonical forms)**.** *Let v be a closed, well-typed value.*

1. *If $\varnothing \vdash v \in A \to B$, then $v = \lambda x{:}C.\, t$ for some C and t.*
2. *If $\varnothing \vdash v \in \forall X{:}K.\, A$, then $v = \lambda X{:}J.\, t$ for some J and t.*

Proof. Separately for the two parts; each by case analysis, first on v, then on the final typing rule used to derive the respective premise. Since the only values are abstractions, the relevant sub-cases are T-ABS, T-TABS and T-SUB. The sub-cases for T-ABS and T-TABS are immediate. In the sub-cases for T-SUB, we first use the generation lemma for abstractions (Lemma 3.44), then dismiss impossible sub-cases using Lemma 5.30. □

Thanks to the canonical forms lemma, the proof of the progress theorem is now entirely standard.

Theorem 5.32 (progress)**.** *If $\vdash t : A$, then either $t = v$ for some value v, or $t \longrightarrow_{\mathsf{v}} t'$ for some term t'.*

Proof. By routine induction on typing derivations. The cases for T-APP and T-TAPP use the canonical forms lemma (Lemma 5.31). □

6 Extending the theory

One of the goals of $F^{\omega}_{\cdot\cdot}$ is to provide a stepping stone to the development of a higher-order version of the calculus of *Dependent Object Types (DOT)*, Scala's core calculus [4]. While DOT admits encodings of some type operators, it is not expressive enough to cover the full spectrum of type computations supported by Scala's type system because it lacks intrinsics for higher-order computations, such as type operator abstractions and applications [39].

Meanwhile, $F^{\omega}_{\cdot\cdot}$ lacks other important type system constructs found in Scala, such as *type members* and *path-dependent types* which feature prominently in DOT. Ideally, we would like to combine the two calculi to obtain a full theory of higher-order dependent object types. In this chapter, we briefly sketch a possible extension of $F^{\omega}_{\cdot\cdot}$ with type members, and discuss some of the challenges involved in adapting the existing metatheory to that extension.

6.1 Type members in Scala and DOT

Central to all variants of DOT is the notion of *type members*, which corresponds closely to the eponymous construct in Scala. We have already seen a few examples of type members in the introduction. Recall the definition of the module `boundedUniversal` from Chapter 1:

```
object boundedUniversal {
  trait Bounded[B, F[_ <: B]] { def apply[X <: B]: F[X] }
  type All[F[_]] = Bounded[Any, F]
}
```

In Scala, modules (or objects) such as `boundedUniversal` are first-class values, i.e. they can be abstracted over and passed to methods just like any other term. Objects may contain type members such as `Bounded` or `All`, which can be selected using dot-notation, as in `boundedUniversal.All`. Type expressions of the form `p.M` are called *type selections*, with `M` a type label and $\mathtt{p} = \mathtt{x}.\mathtt{f}_1.\mathtt{f}_2.\cdots.\mathtt{f}_n$ a *path*, i.e. a (stable) identifier `x` followed by a sequence of field selections $\mathtt{f}_i$. Since the concrete definition of the type `p.M` depends on the concrete

definition of the path `p`, such types are *path-dependent*.

To see how type members and path-dependent types might be useful, consider the following Scala example:

```
trait Functor {
  type F[_]
  def map[A, B](g: A => B)(fa: F[A]): F[B]
}

def mapTwice[A](f: Functor, g: A => A)(fa: f.F[A]): f.F[A] =
  f.map(g)(f.map(g)(fa))
```

It defines a trait `Functor`, which may be thought of as a module signature or type class for functors. The trait has two members: an abstract type operator `F[_]` (the action of the functor on types), and an abstract method `map` (the action of the functor on maps). The polymorphic method `mapTwice` takes an instance `f` of `Functor` as its first parameter, and applies `f.map` twice to a given function `g: A => A` and instance `fa` of `F[A]`. Note that the signature of the method `mapTwice` is path-dependent: the type `f.F[A]`, which is the type of the argument `fa` as well as the return type of the method, depends on the `Functor` instance `f`.

To call `mapTwice`, we need a concrete instance of `Functor`. Here are two candidates:

```
object listFunctor extends Functor {
  type F[X] = List[X]
  def map[A, B](g: A => B)(fa: F[A]): F[B] = fa.map(g)
}

object idFunctor extends Functor {
  type F[X] = X
  def map[A, B](g: A => B)(fa: F[A]): F[B] = g(fa)
}

mapTwice(listFunctor, (x: Int) => x + 1)(List(1, 2, 3))  // List(3, 4, 5)
mapTwice(idFunctor,   (x: Int) => x + 1)(1)              // 3
```

In DOT, as in Scala, a type member definition $\{M = A\}$ is a term-level construct that associates a type label M with a proper type A. Type member definitions inhabit type member declarations of the form $\{M : A \,..\, B\}$ which are themselves proper types and which consist of a type label M and a pair of types A and B bounding M from below and above, respectively. In other words, a type member declaration associates a label M with a type interval $A \,..\, B$. The canonical type of a member definition $\{M = A\}$ is the singleton declaration $\{M : A \,..\, A\}$.

Since type member definitions in DOT are restricted to proper types, Scala traits with higher-kinded type members like the one in our `Functor` example cannot be faithfully modeled in DOT. In the remainder of this chapter, we take a first step towards a higher-order version of DOT by sketching a minimal extension of $F^{\omega}_{::}$ with type members.[1]

6.2 Higher-order type members

While type intervals are baked into type declarations in DOT, we have separated the concept of type intervals from that of type member declarations in $F^{\omega}_{::}$ through our notion of interval kinds. This allows arbitrary types – not just type members – to inhabit intervals, and suggests a natural generalization of type member declarations to the higher-order setting: a higher-order type member declaration $\{ M : K \}$ associates the type label M with an arbitrary kind K, which may or may not be a (higher-order) type interval.

Type member definitions are the introduction forms of type member declarations: they pack types A into terms $\{ M = A \}$. The corresponding elimination forms are type selections $p.M$ consisting of a path p and a type label M. As in DOT, we restrict paths p to be either variables $p = x$ or values $p = v$, in order to avoid the complexities of fully dependent types and issues arising from non-normalizing terms in versions of DOT that feature recursion. This restriction is not unique to DOT, its use, along with that of the dot-notation for type selection, can be traced back at least to Harper and Lillibridge's work on translucent sums [29, 33].

To support type members and type selections, we extend the grammar of $F^{\omega}_{::}$ as follows.

$M, N, \ \dots$	Type label
$p, q \quad ::= x \mid v$	Path
$s, t \quad ::= \dots \mid \{ M = A \}$	Type member definition
$u, v, w ::= \dots \mid \{ M = A \}$	Type member definition (value)
$A, B, C ::= \dots \mid \{ M : K \} \mid p.M$	Type member declaration and selection

On the static side, we add the following inference rules for kinding, typing and subtyping.

$$\frac{\Gamma \vdash K \ \mathsf{kd}}{\Gamma \vdash \{ M : K \} : *} \ (\text{K-MEM}) \qquad \frac{\Gamma \vdash p : \{ M : K \}}{\Gamma \vdash p.M : K} \ (\text{K-SEL})$$

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash \{ M = A \} : \{ M : K \}} \ (\text{T-MEM}) \qquad \frac{\Gamma \vdash K_1 \leq K_2 \ \mathsf{kd}}{\Gamma \vdash \{ M : K_1 \} \leq \{ M : K_2 \} : *} \ (\text{ST-MEM})$$

The path typing rules correspond to those for term typing but with the obvious restriction to

[1] To be more precise, the extension sketched in this chapter corresponds to a higher-order version of the $D_{<:}$ sub-language of DOT [4, 3]

path expressions as subjects.

It is less clear if and how one should extend the dynamics of types and terms, i.e. the reduction relations $\longrightarrow_{\beta}$ and $\longrightarrow_{\mathsf{v}}$, the normalization function nf, and the computational rules of subtyping, in the presence of type members. Recall that in DOT, there is no explicit notion of computation in types. The only non-canonical types in DOT are path selections $p.M$ on type member definitions $p = \{ M = A \}$. Rather than introducing separate reduction relations on types and paths, such selections are resolved in DOT through a pair of subtyping rules for path selections, which allow one to conclude that $\Gamma \vdash A_1 \leq p.M$ and $\Gamma \vdash p.M \leq A_2$, provided that $\Gamma \vdash p : \{ M : A_1 \,..\, A_2 \}$. These rules closely resemble (and are in fact derivable from) the bound projection rules ST-BND$_{1,2}$.

But things are more complicated in $F^{\omega}_{..}$ due to the presence of type operators and type computations in general. For example, using the current subtyping rules (including the rule ST-MEM introduced above), we cannot, in general, derive that $\Gamma \vdash (\{ M = A \}.M)\, B \not\leq A\, B$. (We will see a counterexample shortly.) Another concern is that closed, well-kinded type applications such as $\vdash (\{ M = \lambda X{:}{*}.\, X \}.M)\, \top$ must be considered normal forms, unless we extend the definitions of $\longrightarrow_{\beta}$ and nf. This severely complicates the proof of (top-level) inversion of canonical subtyping.

On the other hand, extending the dynamics of paths and types comes with its own set of problems. Most importantly – and perhaps surprisingly – a naive extension of $\longrightarrow_{\beta}$ with the following contraction rule for type selections breaks the normalization and subject reduction properties for well-kinded types.

$$\frac{}{\{ M = A \}.M \longrightarrow_{\beta} A} \ (\text{R-MEMSEL})$$

Let use briefly illustrate the problems associated with each of these alternatives – imposing DOT-style restrictions on the dynamics of paths and types, or extending $\longrightarrow_{\beta}$ with reduction rules for paths, type selections and type declarations.

6.2.1 Avoiding additional reductions in types

We start by showing why the contraction rule R-MEMSEL is not necessary in DOT, i.e. how equalities of the form $\{ M = A \}.M = A$ can be established via the bound projection rules ST-BND$_{1,2}$, and where this approach falls short in the presence of type operators.

Consider a type member definition $\{ M = A \}$ for some well-kinded proper type $\Gamma \vdash A : *$. By K-SING, T-MEM, and K-SEL, we have $\Gamma \vdash \{ M = A \}.M : A \,..\, A$, and hence by ST-BND$_{1,2}$ and antisymmetry,

$$\Gamma \vdash \{ M = A \}.M = A : *,$$

i.e. we have successfully resolved $\{ M = A \}.M$ to A.

The example relies on A, and thus $\{ M = A \}.M$, being a proper type, so that K-SING and the bound projection rules can be applied directly to $\{ M = A \}.M$. But similar derivations are possible even if A is a type operator. Consider $\{ M = \lambda X{:}*. A \}$ and assume that $\Gamma, X{:}* \vdash A : *$ and $\Gamma \vdash B : *$ for some B. Then we have $\Gamma \vdash \lambda X{:}*. A : (X{:}*) \to A\,..\,A$ by K-SING and K-ABS, and by T-MEM, K-SEL and K-APP, it follows that

$$\Gamma \vdash (\{ M = \lambda X{:}*. A \}.M)\, B \;:\; (A\,..\,A)[X := B].$$

Hence by ST-BND$_{1,2}$ and antisymmetry,

$$\Gamma \vdash (\{ M = \lambda X{:}*. A \}.M)\, B \;=\; A[X := B] \;:\; *.$$

Letting $B = X$ and using η-expansion, we can further derive

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

However, this principle breaks down if we try to reduce a type selection $\{ M = A \}.M$ where A is an abstract type operator. To see this, let $A = X$ and assume some Γ and B such that $\Gamma(X) = * \to *$ and $\Gamma \vdash B : *$. We can at best derive

$$\Gamma \vdash (\{ M = X \}.M)\, B \;:\; *$$

but this is not precise enough to relate $\{ M = X \}.M$ and X via the bound projection rules.

We conclude that the kinding, typing and subtyping rules are not – in their current form – powerful enough to resolve type member selections in general. We could of course add typed contraction rules for type selections directly to the subtyping judgment without extending the $\longrightarrow_\beta$ relation or the normalization function nf, but this would still leave us with the problem of having to invert canonical inequations involving closed "normal forms" such as $(\{ M = \lambda X{:}K. U \}.M)\, V$ at the top level.

6.2.2 Permitting additional reductions in types

Let us instead consider what happens if we extend the $\longrightarrow_\beta$ relation with the contraction rule R-MEMSEL mentioned above, as well as three additional rules to make $\longrightarrow_\beta$ compatible with the new path and type constructs.

$$\frac{}{\{ M = A \}.M \longrightarrow_\beta A} \;(\text{R-MemSel}) \qquad \frac{p \longrightarrow_\beta p'}{p.M \longrightarrow_\beta p'.M} \;(\text{R-Sel})$$

$$\frac{K \longrightarrow_\beta K'}{\{ M : K \} \longrightarrow_\beta \{ M : K' \}} \;(\text{R-MemDecl}) \qquad \frac{A \longrightarrow_\beta A'}{\{ M = A \} \longrightarrow_\beta \{ M = A' \}} \;(\text{R-MemDef})$$

These reduction rules seem very natural and allow one to resolve type selections directly in raw types, e.g.

$$(\{ M = \lambda X{:}K.\, A \}.M)\, B \quad \longrightarrow_\beta \quad (\lambda X{:}K.\, A)\, B \quad \longrightarrow_\beta \quad A[X := B].$$

However, the rule R-MEMSEL, together with the new kinding and subtyping rules introduced above, breaks many of the metatheoretic properties we established in Chapters 4 and 5, including subject reduction in types, normalization and completeness of the canonical system. It is only partly to blame though: the main culprit is, once again, the possible presence of assumptions with inconsistent bounds in open terms. In $F^\omega_{..}$, inconsistent bounds are a problem at the type level in that they prohibit subtyping inversion in non-empty contexts. With the new rules, in particular K-SEL, type-level assumptions can now be reflected into the kind level, in a way that was not possible before.

To see this, assume a context $\Gamma = \Gamma_1, X{:}\{ M : * \} .. \{ M : * \to * \}, \Gamma_2$. In this context, we have $\Gamma \vdash \{ M = \top \} : \{ M : * \}$ but also $\Gamma \vdash \{ M = \top \} : \{ M : * \to * \}$, by bound projection (ST-BND$_{1,2}$), transitivity (ST-TRANS) and subsumption (T-SUB). By K-SEL, we get $\Gamma \vdash \{ M = \top \}.M : * \to *$ and by K-APP, $\Gamma \vdash (\{ M = \top \}.M)\, \top : *$. If we now apply R-MEMSEL, we step from a well-kinded type to the ill-kinded, stuck type $\top\,\top$:

$$(\{ M = \top \}.M)\, \top \quad \longrightarrow_\beta \quad \top\,\top.$$

And thus we have invalidated subject reduction for open types. Similar counter examples can be constructed to show that open types are no longer normalizing, invalidating Lemma 4.20 (well-kinded types have simply-kinded normal forms) and Lemma 5.22 (completeness of the canonical system).

This does of course not mean that type safety no longer holds. It may well be the case that restricted versions of subject reduction, normalization and subtyping inversion hold for closed terms. However our proof strategies from Chapter 4 and Chapter 5 rely in essential ways on types being normalized bottom-up, including the bodies of possibly inconsistent abstractions. A safety proof for the extended calculus will therefore most likely require a different, more robust approach to establish normalization. A promising candidate in the form of a logical-relations-based strong normalization proof for $D_{<:}$ – a subset of DOT – was recently proposed by Wang and Rompf [51].

6.2.3 Non-termination

Interestingly, the problem of non-terminating types can be recreated in the new Dotty compiler for Scala [40]. The following code is taken from a recent bug report.[2]

[2]Filed on July 18, 2017 by the author. See `https://github.com/lampepfl/dotty/issues/2887`

```
trait A { type S[X[_] <: [_] => Any, Y[_]] <: [_] => Any; type I[_] }
trait B { type S[X[_],Y[_]]; type I[_] <: [_] => Any }
trait C { type M <: B }
trait D { type M >: A }

object Test {
  def test(x: C & D): Unit = {
    def foo(a: A, b: B)(z: a.S[b.I,a.I][b.S[a.I,a.I]]) = z
    def bar(a: A, y: x.M) = foo(a,y)
    def baz(a: A) = bar(a, a)
    baz(new A {
      type S[X[_] <: [_] => Any, Y[_]] = [Z] => X[Z][Y[Z]];
      type I[X] = X
    })(1)
  }
}
```

This snippet uses an encoding of the well-known non-terminating untyped SKI-combinator term $SII(SII)$, first into the untyped lambda calculus, then into the Scala type system, in order to trap the type checker in a non-terminating type reduction. To get the offending type to kind check, it uses a variant of the above example of inconsistent bounds being reflected into the kind level.

The code uses an intersection type `C & D` in order to introduce a pair of type members `S` and `I` with inconsistent kind annotations. The use of the intersection `C & D` is necessary because the compiler detects and rejects member declarations with blatantly inconsistent bounds. So instead, two pairs of conflicting declarations are prepared in the traits `A` and `B` and combined via the shared member `M` of the intersection `C & D`.

The example is deceptively simple. It may be surprising that the compiler is not able to detect the inconsistent bounds and reject them. But in general, such bounds can arise through much more subtle combinations of traits with seemingly consistent member declarations. Finding a more systematic approach to detecting such instances remains an open problem.

7 Conclusion

We set out to show that higher-order bounded quantification, bounded operator abstractions and translucent type definitions can be uniformly expressed through the concept of type intervals. To this end, we developed $F^{\omega}_{..}$, a formal theory of higher-order subtyping with type intervals.

In $F^{\omega}_{..}$, type intervals are represented through interval kinds. We showed how interval kinds can be used to encode bounded universal quantification, bounded type operators as well as singleton kinds in a semantics-preserving manner. We also gave examples of how interval kinds can be used to abstract over and use first-class higher-order type inequalities. We discussed and illustrated problems that arise when such abstractions are inconsistent, i.e. when the corresponding interval kinds have inconsistent bounds.

We established basic metatheoretic properties of $F^{\omega}_{..}$. 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. Our normalization proof is based on hereditary substitution and thus fully syntactic.

We gave an alternative, canonical presentation of the kind and type level of $F^{\omega}_{..}$, defined directly on $\beta\eta$-normal forms. We proved 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, our metatheoretic development culminated in a type safety proof of $F^{\omega}_{..}$.

Our entire metatheoretic developments is syntactic, i.e. does not involve any model constructions, and has been fully mechanized in Agda [50].

Finally, we briefly sketched a possible extension of $F^{\omega}_{..}$ toward a higher-order version of the calculus of Dependent Object Types (DOT) and discussed some of the challenges involved in adapting the existing metatheory to that extension. Though we leave the full development of

higher-order DOT for future work, we believe that the development of $F^{\omega}_{\cdot\cdot}$ constitutes in itself a non-trivial step toward the goal of developing solid theoretical foundations for the Scala programming language.

Bibliography

[1] A. Abel. Polarized subtyping for sized types. *Mathematical Structures in Computer Science*, 18:797–822, 10 2008.

[2] A. Abel and D. Rodriguez. Syntactic metatheory of higher-order subtyping. In M. Kaminski and S. Martini, editors, *Proceedings of the 22nd International Workshop on Computer Science Logic (CSL 2008), 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16–19, 2008*, volume 5213 of *LNCS*, pages 446–460, Berlin, Heidelberg, 2008. Springer.

[3] N. Amin. *Dependent Object Types*. PhD thesis, School of Computer and Communication Sciences, École polytechnique fédérale de Lausanne, Lausanne, Switzerland, 2016. EPFL thesis no. 7156.

[4] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. The essence of dependent object types. In S. Lindley, C. McBride, P. Trinder, and D. Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, volume 9600 of *LNCS*, pages 249–272. Springer International Publishing, 2016.

[5] N. Amin, A. Moors, and M. Odersky. Dependent object types. In *Proceedings of the 19th International Workshop on Foundations of Object-Oriented Languages (FOOL 2012), Tucson, AZ, USA, October 22, 2012*, 2012.

[6] N. Amin and T. Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017), Paris, France*, pages 666–679. ACM, 2017.

[7] N. Amin, T. Rompf, and M. Odersky. 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*, pages 233–249, New York, NY, USA, 2014. ACM.

[8] D. Aspinall. Subtyping with singleton types. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th Workshop (CSL 1994), Kazimierz, Poland, September 25–30, 1994 Selected Papers*, volume 933 of *LNCS*, pages 1–15, Berlin, Heidelberg, 1995. Springer.

[9] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2):273–309, 2001.

[10] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, Oxford, UK, 1992.

[11] L. Cardelli. 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*, pages 70–79, New York, NY, USA, 1988. ACM.

[12] L. Cardelli. Notes about $F^{\omega}_{<:}$. Unpublished manuscript, October 1990.

[13] L. Cardelli and G. Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[14] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system f with subtyping. In T. Ito and A. R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS 1991), Sendai, Japan, September 24–27, 1991*, pages 750–770, Berlin, Heidelberg, 1991. Springer.

[15] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, Dec. 1985.

[16] A. Compagnoni. Higher-order subtyping and its decidability. *Information and Computation*, 191(1):41–103, 2004.

[17] A. Compagnoni and H. Goguen. Anti-symmetry of higher-order subtyping. In J. Flum and M. Rodriguez-Artalejo, editors, *Proceedings of the 13th International Workshop on Computer Science Logic (CSL 1999), 8th Annual Conference of the EACSL Madrid, Spain, September 20–25, 1999*, volume 1683 of *LNCS*, pages 420–438, Berlin, Heidelberg, 1999. Springer.

[18] A. Compagnoni and H. Goguen. Typed operational semantics for higher-order subtyping. *Information and Computation*, 184(2):242–297, 2003.

[19] A. B. Compagnoni. Decidability of higher-order subtyping with intersection types. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic, 8th International Workshop, (CSL 1994), Kazimierz, Poland, September 25–30, 1994, Selected Papers*, volume 933 of *LNCS*, pages 46–60, Berlin, Heidelberg, 1995. Springer.

[20] K. Crary. 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*, pages 125–135, New York, NY, USA, 1997. ACM.

[21] K. Crary. 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*, pages 21–29, New York, NY, USA, 2009. ACM.

[22] K. F. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[23] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In R. Královič and P. Urzyczyn, editors, *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science (MFCS 2006), Stará Lesná, Slovakia, August 28–September 1, 2006*, volume 4162 of *LNCS*, pages 1–23, Berlin, Heidelberg, 2006. Springer.

[24] J. Cretin and D. Rémy. System f with coercion constraints. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), Vienna, Austria, July 2014*, pages 34:1–34:10, New York, NY, USA, 2014. ACM.

[25] P.-L. Curien and G. Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2(1):55–91, March 1992.

[26] D. Duggan and A. Compagnoni. Subtyping for object type constructors. In *Proceedings of 6th International Workshop on Foundations of Object-Oriented Languages (FOOL 6), San Antonio, TX, USA, January 23, 1999*, page 4, 1999.

[27] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[28] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, July 2007.

[29] R. Harper and M. Lillibridge. 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*, pages 123–137, New York, NY, USA, 1994. ACM.

[30] R. Harper and F. Pfenning. On equivalence and canonical forms in the lf type theory. *ACM Transactions on Computational Logic*, 6(1):61–101, Jan. 2005.

[31] C. Keller and T. Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming (MSFP 2010), Baltimore, Maryland, USA*, pages 3–10, New York, NY, USA, 2010. ACM.

[32] D. K. Lee, K. Crary, and R. Harper. 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*, pages 173–184, New York, NY, USA, 2007. ACM.

[33] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 1997. Available as CMU Technical Report CMU-CS-97-122.

[34] A. Moors, F. Piessens, and M. Odersky. 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*, pages 423–438, New York, NY, USA, 2008. ACM.

[35] A. Moors, F. Piessens, and M. Odersky. Safe type-level abstraction in Scala. In *Proceedings of the International Workshop on Foundations of Object-Oriented Languages (FOOL 2008), San Francisco, CA, USA, January 13, 2007*, pages 1–13, 2008.

[36] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory*, volume 200. Oxford University Press, Oxford, UK, July 1990.

[37] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, September 2007.

[38] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, July 21–25, 2003*, volume 2743 of *LNCS*, pages 201–224, Berlin, Heidelberg, 2003. Springer.

[39] M. Odersky, G. Martres, and D. Petrashko. Implementing higher-kinded types in Dotty. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala (SCALA@SPLASH 2016), Amsterdam, Netherlands, October 30–November 4, 2016*, pages 51–60, New York, NY, USA, 2016. ACM.

[40] M. Odersky and the Dotty Team. Dotty – a research platform for new language concepts and compiler technologies for Scala – `http://dotty.epfl.ch`. Source code available from `https://github.com/lampepfl/dotty`, 2017.

[41] F. Pfenning and C. Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Trento, Italy, July 7–10, 1999*, volume 1632 of *LNCS*, pages 202–206, Berlin, Heidelberg, 1999. Springer.

[42] B. Pierce and M. Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1–2):235–282, 1997.

[43] B. C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.

[44] T. Rompf and N. Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University, 2015. `http://arxiv.org/abs/1510.05216`.

[45] T. Rompf and N. Amin. 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,* pages 624–641, New York, NY, USA, 2016. ACM.

[46] G. Scherer and D. Rémy. Full reduction in the face of absurdity. In J. Vitek, editor, *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, April 11–18, 2015*, volume 9032 of *LNCS*, pages 685–709, Berlin, Heidelberg, 2015. Springer.

[47] M. Steffen. *Polarized Higher-Order Subtyping.* PhD thesis, Technische Fakultät, Universität Erlangen, 1998.

[48] C. A. Stone. Type definitions. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 9, pages 347–400. MIT Press, Cambridge, MA, USA, 2004.

[49] C. A. Stone and R. Harper. 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*, pages 214–227, New York, NY, USA, 2000. ACM.

[50] S. Stucki. f-omega-int-agda – F_ω with interval kinds mechanized in Agda. Source code available from `https://github.com/sstucki/f-omega-int-agda`, 2017.

[51] F. Wang and T. Rompf. Towards strong normalization for dependent object types (DOT). In P. Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017), Barcelona, Spain, June 18–23, 2017*, volume 74 of *LIPIcs*, pages 27:1–27:25, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[52] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *International Workshop on Types for Proofs and Programs (TYPES 2003), Torino, Italy, April 30–May 4, 2003, Revised Selected Papers*, volume 3085 of *LNCS*, pages 355–377, Berlin, Heidelberg, 2004. Springer.

[53] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.

[54] J. Zwanenburg. Pure type systems with subtyping. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA 1999), L'Aquila, Italy, April 7–9, 1999*, volume 1581 of *LNCS*, pages 381–396. Springer, Berlin, Heidelberg, 1999.

CURRICULUM VITAE

Sandro Stucki

E-MAIL `sandro.stucki@epfl.ch`

WEB `https://sstucki.github.io/`

PHONE +41 77 429 5407

ADDRESS Avenue de Beaumont 1
1012 Lausanne
Switzerland

Research interests

- Programming languages.
- Type systems and theory.
- Semantics and implementation of domain-specific languages.
- Formal methods for scientific modeling.

Education

SEPT 2012 – TODAY **PhD Computer Science at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland** (ongoing)

PhD thesis at the Programming Methods Laboratory (LAMP) under the supervision of Prof. Martin Odersky.

My main research revolves around the formalization of Scala's core type system. Other research topics include the design of domain-specific languages for modeling probabilistic and stochastic systems, especially biochemical systems. I also supervise MSc student projects and TA both undergraduate and graduate classes.

SEPT 2011 – SEPT 2012 **MSc Computer Science at Université Paris Diderot, France**

Parisian Master of Research in Computer Science (MPRI)

Master's thesis in collaboration with the Center for Synthetic and Systems Biology (SynthSys) at the University of Edinburgh, UK and the Proofs, Programs and Systems (PPS) laboratory at Université Paris Diderot.

The curriculum consisted mainly of courses in theoretical computer science, including advanced algorithms and complexity, functional programming and type systems, as well as computational biology.

OCT 2002 – FEB 2008 **MSc Computer Science at EPFL, Switzerland**

Master of Science in Computer Science, specialized in Computer Engineering

Master's thesis in collaboration with Mitrionics AB, Lund, Sweden and the Processor Architecture Laboratory (LAP) at EPFL.

The curriculum included advanced courses in computer architecture, digital design, embedded systems and compiler construction as well as courses in computational modeling of biological systems.

AUG 2004 – JUN 2005 **Exchange year at Linköpings University (LiU), Sweden**

Courses from the computer science curriculum.

Teaching experience

CS-452 – FALL 2016 As an assistant TA of the graduate course on Foundations of Software (CS-452 – FOS), I supervised weakly exercise sessions, lead tutorial sessions and helped in the design and grading of exams.

CS-171 – 2013–2016 As the head TA of the undergraduate course on Logic Systems (CS-171), I supervised weakly exercise sessions and helped in the grading of exams and term projects.

STUDENT SUPERVISION I supervised 3 MSc thesis projects on topics in the areas of type systems and domain-specific languages.

Selected publications

WADLERFEST'16 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and *Sandro Stucki*. 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*, 2016.

SCALA'15 Manohar Jonnalagedda and *Sandro Stucki*. Fold-Based Fusion as a Library: a Generative Programming Pearl. In *Proc. ACM SIGPLAN Symposium on Scala*, 2015.

ICFEM'14 Vincent Danos, Tobias Heindel, Ricardo Honorato-Zimmer, and *Sandro Stucki*. Approximations for Stochastic Graph Rewriting. In *Proc. Formal Methods and Software Engineering*, 2014.

OOPSLA'14 Manohar Jonnalagedda, Thierry Coppey, *Sandro Stucki*, Tiark Rompf, and Martin Odersky. Staged Parser Combinators for Efficient Data Processing. In *Proc. Object Oriented Programming Systems Languages & Applications*, 2014.

GPCE'14 Vojin Jovanovic, Amir Shaikhha, *Sandro Stucki*, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. Yin-yang: Concealing the Deep Embedding of DSLs. In *Proc. International Conference on Generative Programming: Concepts and Experiences*, 2014.

SCALA'13 *Sandro Stucki*, Nada Amin, Manohar Jonnalagedda, and Tiark Rompf. What are the Odds? Probabilistic programming in Scala. In *Proc. Workshop on Scala*, 2013.

Industry experience

JUL 2014 – SEP 2014 **Research intern at Oracle Labs, Geneva, Switzerland**

Development of graph database algorithms based on graph rewriting.

MAR 2010 – JULY 2011 **Systems developer at ResQU AB, Lund, Sweden**

- *Software development* of low-level software components of a GSM-based search and rescue system (Hepkie) in C/C++

MAR 2008 – FEB 2010 **Applications developer at Mitrionics AB, Lund, Sweden**

- *Application development* of FPGA-accelerated applications (mostly for bioinformatics and text processing) in C/C++ and Mitrion-C
- *Product development* of the Mitrion SDK (Mitrion-C compiler, hardware abstraction layer "Mithal") in Java and C/C++

APR 2006 – SEPT 2006 **Internship at Thomson's Corporate Research Lab, Hanover, Germany**

Development of tools for GPU-accelerated image processing of high-definition video frames in C/C++, OpenGL, GLSL.

Language skills

GERMAN Mother tongue.

ENGLISH Fluent speaker, very good writing skills (CAE certificate)

FRENCH Fluent speaker, good writing skills.

SWEDISH Fluent speaker, good writing skills (TISUS certificate).