A Theory of Higher-Order Subtyping with Type Intervals

Sandro Stucki    Paolo G. Giarrusso

ICPF 2021 – 22–27 Aug 2021
sandros@chalmers.se    @stuckintheory
Declarative Subtyping

Inconsistent Bounds

Canonical Subtyping

Type Safety

S. Stucki, P. G. Giarrusso A Theory of Higher-Order Subtyping with Type Intervals
The Essence of Dependent Object Types

Nada Amin¹, Samuel Grütter¹, Martin Odersky¹¹, Tiark Rompf², and Sandro Stucki³

¹ EPFL, Lausanne, Switzerland
{nada.amin,samuel.grutter,martin.odersky,sandro.stucki}@epfl.ch
² Purdue University, West Lafayette, USA
tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus Dₖ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in Dₖ and demonstrates the expressiveness of DOT by modelling a range of Scala constructs in it.
DOT and Dotty

DOT

• a minimal core calculus for Scala

The Essence of Dependent Object Types

WadlerFest, April 2016

Nada Amin¹, Samuel Grütter¹, Martin Odersky¹( ), Tiark Rompf², and Sandro Stucki³

¹ EPFL, Lausanne, Switzerland
{manda.amin, samuel.grutter, martin.odersky, sandro.stucki}@epfl.ch
² Purdue University, West Lafayette, USA
³ tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus D₀ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in D₀ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.
DOT and Dotty

DOT

• a minimal core calculus for Scala
• proven type-safe (in Coq)

The Essence of Dependent Object Types

Nada Amin1, Samuel Grütter1, Martin Odersky1( ), Tiark Rompf2, and Sandro Stucki3

1 EPFL, Lausanne, Switzerland
{nada.amin, samuel.grutter, martin.odersky, sandro.stucki}@epfl.ch
2 Purdue University, West Lafayette, USA
tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus Dc of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in Dc and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.
DOT and Dotty

DOT

- a minimal core calculus for Scala
- proven type-safe (in Coq)
- does not support HK types

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_c$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_c$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.
DOT and Dotty

DOT

• a minimal core calculus for Scala
• proven type-safe (in Coq)
• does not support HK types

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin1, Samuel Grütter1, Martin Odersky1( ), Tiark Rompf2, and Sandro Stucki3

1 EPFL, Lausanne, Switzerland
{nada.amin, samuel.grutter, martin.odersky, sandro.stucki}@epfl.ch
2 Purdue University, West Lafayette, USA
tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_0$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_0$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Dotty/Scala 3

Implementing Higher-Kinded Types in Dotty

Martin Odersky, Guillaume Martres, Dmitry Petrashko
EPFL, Switzerland: {first.last}@epfl.ch

Abstract
dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kind types are a natural extension of first-order lambda calculus, and have been a core construct of Haskell and Scala. As long as such types are just partial applications of generic classes, they can be given a meaning in DOT relatively straightforwardly. But general lambdas on the type level require extensions of the DOT calculus to be expressible. This paper is an experience report where we describe and discuss four implementation strategies that we have tried out in the last three years. Each strategy was fully implemented in the dotty compiler. We discuss the usability and expressive power of proved to be challenging, so much so that we evaluated four different strategies before settling on the current direct representation encoding. The strategies are summarized as follows:

• A simple encoding in the DOT-inspired [9] core type structures that can express partial applications and not much more
• A direct representation that adds support for full type lambdas and higher-kind applications, without reusing much of the existing concepts of the calculus and the compiler.

S. Stucki, P. G. Giarrusso

A Theory of Higher-Order Subtyping with Type Intervals
DOT and Dotty

DOT

• a minimal core calculus for Scala
• proven type-safe (in Coq)
• does not support HK types

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin1, Samuel Grütter1, Martin Odersky1, Tiark Rompf2, and Sandro Stucki3

1 EPFL, Lausanne, Switzerland
{nada.amin,samuel.grutter,martin.odersky,sandro.stucki}@epfl.ch
2 Purdue University, West Lafayette, USA
tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus \( D_0 \) of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in \( D_0 \) and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Dotty/Scala 3

• a Scala compiler based on DOT

Scala Symposium, Oct 2016

Implementing Higher-Kinded Types in Dotty

Martin Odersky, Guillaume Martres, Dmitry Petrashko
EPFL, Switzerland: (first.last)@epfl.ch

Abstract
dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kind types are a natural extension of first-order lambda calculus, and have been a core construct of Haskell and Scala. As long as such types are just partial applications of generic classes, they can be given a meaning in DOT relatively straightforwardly. But general lambdas on the type level require extensions of the DOT calculus to be expressible. This paper is an experience report where we describe and discuss four implementation strategies that we have tried out in the last three years. Each strategy was fully implemented in the dotty compiler. We discuss the usability and expressive power of

proved to be challenging, so much so that we evaluated four different strategies before settling on the current direct representation encoding. The strategies are summarized as follows:

• A simple encoding in the DOT-inspired [9] core type structures that can express partial applications and not much more
• A direct representation that adds support for full type lambdas and higher-kind applications, without reusing much of the existing concepts of the calculus and the compiler.
DOT and Dotty

**DOT**

- a minimal core calculus for Scala
- proven type-safe (in Coq)
- does not support HK types

**Dotty/Scala 3**

- a Scala compiler based on DOT
- type safety unclear

---

The Essence of Dependent Object Types

Nada Amin¹, Samuel Grütter¹, Martin Odersky¹, Tiark Rompf², and Sandro Stucki³

¹ EPFL, Lausanne, Switzerland
² Purdue University, West Lafayette, USA
³ EPFL, Switzerland: {first.last}@epfl.ch

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_1$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_1$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.
DOT and Dotty

DOT

- a minimal core calculus for Scala
- proven type-safe (in Coq)
- does not support HK types

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin¹, Samuel Grütter¹, Martin Odersky¹( ), Tiark Rompf², and Sandro Stucki¹

¹ EPFL, Lausanne, Switzerland
² Purdue University, West Lafayette, USA

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_0$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_0$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Dotty/Scala 3

- a Scala compiler based on DOT
- type safety unclear
- does support HK types

Scala Symposium, Oct 2016

Implementing Higher-Kinded Types in Dotty

Martin Odersky, Guillaume Martres, Dmitry Petrashko
EPFL, Switzerland: {first.last}@epfl.ch

Abstract
dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kindred types are a natural extension of first-order lambda calculus, and have been a core construct of Haskell and Scala. As long as such types are just partial applications of generic classes, they can be given a meaning in DOT relatively straightforwardly. But general lambdas on the type level require extensions of the DOT calculus to be expressible. This paper is an experience report where we describe and discuss four implementation strategies that we have tried out in the last three years. Each strategy was fully implemented in the dotty compiler. We discuss the usability and expressive power of prototyped, and shortcomings of all of them. But significant theoretical work remains to be done.
HK Types – An Example

type Ordering[A] = (A, A) => Boolean

abstract class SortedView[A, B <: A](xs: List[A], ord: Ordering[B]) {
  def foldLeft[C](z: C, op: (C, A) => C): C
  def concat[C <: A <=: B](ys: List[C]): SortedView[C, B]
  // declarations of further operations such as 'map', 'flatMap', etc.
}
HK Types – An Example

type Ordering[A] = (A, A) => Boolean

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

• Types can take parameters: i.e. we have type operators.
HK Types – An Example

```scala
type Ordering[A] = (A, A) => Boolean

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

- Types can take parameters: i.e. we have type operators.
- Type parameters of methods can have bounds (as usual).
HK Types – An Example

type Ordering[A] = (A, A) => Boolean

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

• Types can take parameters: i.e. we have type operators.
• Type parameters of methods can have bounds (as usual).
• Type parameters of operators can also have bounds!
HK Types – An Example

type Ordering[A] = (A, A) => Boolean

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

- Types can take parameters: i.e. we have type operators.
- Type parameters of methods can have bounds (as usual).
- Type parameters of operators can also have bounds!
- Type definitions can be used to introduce aliases.

S. Stucki, P. G. Giarrusso
The Anatomy of a Type Interval

\[ X >: A <: B \]

**Special cases**
- **Upper bound**
  \[ X <: B \]
  \[ X : \bot \ldots \top \]
- **Lower bound**
  \[ X >: A \]
  \[ X : \bot \ldots \top \]

**Abstract**
\[ X : \bot \ldots \top \]

**Alias**
- \( X = A \)
  \[ X : A \ldots A \]
  - \( \bot = \) Nothing
  - \( \top = \) Any
  - \( \bot \ldots \top = \ast = \) kind of all types.
- \( A \ldots A = \) singleton containing only \( A \).
The Anatomy of a Type Interval

\[ X >: A <: B \]

*Intuition:* \( X \) has bounds \( A <: X <: B \).
The Anatomy of a Type Interval

$X >: A <: B$

*Intuition:* $X$ is an element of the set of types $\{ A <: \cdots <: B \}$
The Anatomy of a Type Interval

$X >: A <: B$

*Intuition:* $X$ is an element of the set of types $\{ A <: \cdots <: B \} = A .. B$
The Anatomy of a Type Interval

\[
X >: A <: B
\]

\[
X : A .. B
\]

*Intuition:* \(X\) is an element of the set of types \(\{A <: \cdots <: B\} = A .. B\)
The Anatomy of a Type Interval

\[ X >: A <: B \]
\[ X : A .. B \]

**Intuition:** \( X \) is an element of the set of types \( \{ A <: \cdots <: B \} = A .. B \)

**Special cases**

- **Upper bound** \( X <: B \) \( X : \bot .. B \)

- \( \bot = \text{Nothing} = \text{minimal/bottom type} \)
The Anatomy of a Type Interval

\[ X >: A <: B \quad \text{and} \quad X : A .. B \]

*Intuition*: \( X \) is an element of the set of types \( \{ A <: \cdots <: B \} = A .. B \)

**Special cases**

- **Upper bound** \( X <: B \) \[ X : \bot .. B \]
- **Lower bound** \( X >: A \) \[ X : A .. \top \]

- \( \bot = \text{Nothing} = \text{minimal/bottom type} \)
- \( \top = \text{Any} = \text{maximal/top type} \)
The Anatomy of a Type Interval

\[ X >: A <: B \quad X : A .. B \]

*Intuition:* \(X\) is an element of the set of types \(\{ A <: \cdots <: B \} = A .. B \)

**Special cases**

- **Upper bound** \(X <: B\) \(X : \bot .. B\)
- **Lower bound** \(X >: A\) \(X : A .. \top\)
- **Abstract** \(X\) \(X : \bot .. \top\)

- \(\bot = \text{Nothing} = \text{minimal/bottom type};\)
- \(\top = \text{Any} = \text{maximal/top type};\)
- \(\bot .. \top = \ast = \text{kind of all types}.\)
The Anatomy of a Type Interval

\[ X >: A <: B \quad X : A .. B \]

*Intuition:* \( X \) is an element of the set of types \( \{ A <: \cdots <: B \} = A .. B \)

**Special cases**

- **Upper bound**
  \[ X <: B \quad X : \bot .. B \]
- **Lower bound**
  \[ X >: A \quad X : A .. \top \]
- **Abstract**
  \[ X \]
  \[ X : \bot .. \top \]
- **Alias**
  \[ X = A \]
  \[ X : A .. A \]

- \( \bot = \text{Nothing} = \text{minimal/bottom type} \)
- \( \top = \text{Any} = \text{maximal/top type} \)
- \( \bot .. \top = * = \text{kind of all types} \)
- \( A .. A = \text{singleton containing only } A \)
The Anatomy of a Type Interval (cont.)

\[ F[X >: A <: B] >: G <: H \]

We can also represent bounded operators
The Anatomy of a Type Interval (cont.)

\[ F[X >: A <: B] >: G <: H \quad \quad F : (X:A..B) \rightarrow G..H \]

We can also represent bounded operators

F_1[X] = List[X]  
F_1 : (X:* \rightarrow List X..List X

Upper bound

F_2[X] <: List[X]  
F_2 : (X:* \rightarrow \bot..List X

HO bounded op.

F_3[X, Y[<: X]]  
F_3 : (X:* \rightarrow (Y_:\bot..X)\rightarrow * \rightarrow *
The Anatomy of a Type Interval (cont.)

\[ F[X >: A <: B] >: G <: H \quad F : (X:A .. B) \rightarrow G .. H \]

We can also represent bounded operators

Examples

Alias \[ F_1[X] = \text{List}[X] \quad F_1 : (X:*) \rightarrow \text{List } X .. \text{List } X \]
The Anatomy of a Type Interval (cont.)

\[
F[X >: A <: B] >: G <: H \quad F : (X:A..B) \rightarrow G..H
\]

We can also represent bounded operators

Examples

Alias \hspace{1cm} F1[X] = \text{List}[X] \hspace{1cm} F_1 : (X:* \rightarrow \text{List} X .. \text{List} X

Upper bound \hspace{1cm} F2[X] <: \text{List}[X] \hspace{1cm} F_2 : (X:* \rightarrow \bot .. \text{List} X
The Anatomy of a Type Interval (cont.)

\[ F[X >: A <: B] >: G <: H \quad F : (X:A..B) \to G..H \]

We can also represent bounded operators

Examples

Alias \hspace{1em} F1[X] = List[X] \hspace{1em} F_1 : (X:*) \to List X .. List X

Upper bound \hspace{1em} F2[X] <: List[X] \hspace{1em} F_2 : (X:*) \to \bot .. List X

HO bounded op. \hspace{1em} F3[X, Y[_ <: X]] \hspace{1em} F_3 : (X:*) \to (Y:(_:\bot .. X) \to *) \to *

NB. The operators \( F_1 \) – \( F_3 \) all have dependent kinds.
The Anatomy of a Type Interval (cont.)

\[ F[X >: A <: B] >: G <: H \quad F : (X:A .. B) \rightarrow G .. H \]

We can also represent bounded operators

Examples

- **Alias**
  \[ F1[X] = \text{List}[X] \quad F_1 : (X:* \rightarrow \text{List}X \rightarrow \text{List}X \]

- **Upper bound**
  \[ F2[X] <: \text{List}[X] \quad F_2 : (X:* \rightarrow \bot \rightarrow \text{List}X \]

- **HO bounded op.**
  \[ F3[X, Y[\_ <: X]] \quad F_3 : (X:* \rightarrow (Y:(\_ : \bot .. X) \rightarrow *) \rightarrow * \]

**NB.** The operators \( F_1 \rightarrow F_3 \) all have dependent kinds.
Proving Type Safety of $F^\omega$.

Main sub-challenges:
1. Subtyping derivations may involve computation ($\beta\eta$-conversions).
2. Subtyping derivations may involve subsumption (via subkinding).
3. Type variables with inconsistent bounds can reflect arbitrary subtyping assumptions into subtyping derivations.
Proving Type Safety of $F^\omega$

The big challenge is to prove subtyping inversion.
The big challenge is to prove subtyping inversion.

\[
\begin{align*}
\Gamma \vdash A_1 \rightarrow B_1 <: A_2 \rightarrow B_2 : * \\
\Gamma \vdash A_2 <: A_1 : * \quad \Gamma \vdash B_1 <: B_2 : *
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \forall X : K_1 . A_1 <: \forall X : K_2 . A_2 : * \\
\Gamma \vdash K_2 <: K_1 \quad \Gamma, X : K_2 \vdash A_1 <: A_2 : *
\end{align*}
\]
The big challenge is to prove \textit{subtyping inversion}.

\[
\begin{align*}
\Gamma &
\vdash A_1 \to B_1 <: A_2 \to B_2 : * \\
\Gamma &
\vdash A_2 <: A_1 : * & \Gamma &
\vdash B_1 <: B_2 : *
\end{align*}
\]

\[
\begin{align*}
\Gamma &
\vdash \forall X : K_1. A_1 <: \forall X : K_2. A_2 : * \\
\Gamma &
\vdash K_2 <: K_1 & \Gamma, X : K_2 &
\vdash A_1 <: A_2 : *
\end{align*}
\]

\textbf{Main sub-challenges:}

1. Subtyping derivations may involve \textit{computation} ($\beta\eta$-conversions).
Proving Type Safety of $F^\omega$

The big challenge is to prove subtyping inversion.

\[
\Gamma \vdash A_1 \rightarrow B_1 <: A_2 \rightarrow B_2 : *
\]
\[
\Gamma \vdash A_2 <: A_1 : *
\]
\[
\Gamma \vdash B_1 <: B_2 : *
\]
\[
\Gamma \vdash \forall X:K_1.\, A_1 <: \forall X:K_2.\, A_2 : *
\]
\[
\Gamma \vdash K_2 <: K_1
\]
\[
\Gamma, X:K_2 \vdash A_1 <: A_2 : *
\]

Main sub-challenges:

1. Subtyping derivations may involve computation ($\beta\eta$-conversions).
2. Subtyping derivations may involve subsumption (via subkinding).
Proving Type Safety of $F^{\omega}$

The big challenge is to prove subtyping inversion.

$$
\frac{
\Gamma \vdash A_1 \rightarrow B_1 <: A_2 \rightarrow B_2 : * \\
\Gamma \vdash A_2 <: A_1 : * \\
\Gamma \vdash B_1 <: B_2 : *
}{
\Gamma \vdash A_2 <: A_1 : * \\
\Gamma \vdash B_1 <: B_2 : *
}
$$

Main sub-challenges:

1. Subtyping derivations may involve computation ($\beta\eta$-conversions).
2. Subtyping derivations may involve subsumption (via subkinding).
3. Type variables with inconsistent bounds can reflect arbitrary subtyping assumptions into subtyping derivations.
The big challenge is to prove subtyping inversion.

\[
\begin{align*}
\Gamma \vdash A_1 \rightarrow B_1 < : A_2 \rightarrow B_2 : * \\
\Gamma \vdash A_2 < : A_1 : * \quad \Gamma \vdash B_1 < : B_2 : *
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \forall X : K_1. A_1 < : \forall X : K_2. A_2 : * \\
\Gamma \vdash K_2 < : K_1 \quad \Gamma, X : K_2 \vdash A_1 < : A_2 : *
\end{align*}
\]

Main sub-challenges:

1. Subtyping derivations may involve computation (\(\beta\eta\)-conversions).
2. Subtyping derivations may involve subsumption (via subkinding).
3. Type variables with inconsistent bounds can reflect arbitrary subtyping assumptions into subtyping derivations.
Challenge 1: Getting Rid of $\beta\eta$-Conversions

Problem: $\beta\eta$-conversions get in the way of inversion.

$$
\Gamma \vdash A_1 \to A_2 <: (\lambda X:*. X \to A_2) A_1 <: \cdots <: (\lambda X:*. X \to B_2) B_1 <: B_1 \to B_2 : *$

S. Stucki, P. G. Giarrusso
A Theory of Higher-Order Subtyping with Type Intervals
Challenge 1: Getting Rid of $\beta\eta$-Conversions

Problem: $\beta\eta$-conversions get in the way of inversion.

$\Gamma \vdash A_1 \rightarrow A_2 <: (\lambda X:* . X \rightarrow A_2) A_1 <: \cdots <: (\lambda X:* . X \rightarrow B_2) B_1 <: B_1 \rightarrow B_2 : *$

Solution: normalize types and kinds – no redexes, no conversions!
Challenge 1: Getting Rid of $\beta\eta$-Conversions

New problem: dependent kinding of applications involves substitutions.

$$\Gamma \vdash Z : (X:J) \rightarrow K \quad \Gamma \vdash V : J$$

$$\Gamma \vdash Z \, V : K[V/X]$$
New problem: dependent kinding of applications involves substitutions.

\[
\Gamma \vdash Z : (X : J) \rightarrow K \quad \Gamma \vdash V : J
\]
\[
\Gamma \vdash Z \, V : K[V/X]
\]
Challenge 1: Getting Rid of $\beta\eta$-Conversions

New problem: dependent kinding of applications involves substitutions.

\[
\Gamma \vdash Z : (X : J) \rightarrow K \quad \Gamma \vdash V : J
\]

\[
\Gamma \vdash Z \, V : K[V/X|J|]
\]

New solution: use hereditary substitution
Challenge 1: Getting Rid of $\beta\eta$-Conversions

New problem: dependent kinding of applications involves substitutions.

$$
\Gamma \vdash Z : (X : J) \to K \quad \Gamma \vdash V : J
$$

$$
\Gamma \vdash Z V : K[V/X|J|]
$$

New solution: use hereditary substitution (introducing further problems...)

S. Stucki, P. G. Giarrusso
A Theory of Higher-Order Subtyping with Type Intervals
Challenge 3: Inconsistent Bounds

**Problem:** Type variables can introduce arbitrary subtyping relationships.

NB. This causes all sorts of problems:
- subject reduction (preservation) fails,
- subtyping becomes undecidable,
- . . .

**Solution:** invert $<: X$ only for closed types – no variables, no inconsistencies!
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

\[ X : \top .. \bot \vdash X : * \]
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

\[
\begin{align*}
X &: \top \ldots \bot \vdash \\
\top &: X \\
\end{align*}
\]

NB. This causes all sorts of problems:
- subject reduction (preservation) fails,
- subtyping becomes undecidable,
- ...

Solution: invert \(<\) only for closed types – no variables, no inconsistencies!
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

\[
X : \top .. \bot \vdash A \rightarrow B \leq \top \leq : X
\]

Note: This causes all sorts of problems:

• subject reduction (preservation) fails,
• subtyping becomes undecidable,
• ...

Solution: Invert \( \leq \) only for closed types – no variables, no inconsistencies!
Challenge 3: Inconsistent Bounds

**Problem:** Type variables can introduce *inconsistent* subtyping relationships.

\[
X : \top \ldots \bot \proves A \rightarrow B <: \top <: X <: \bot : * 
\]

NB. This causes all sorts of problems:
- subject reduction (preservation) fails,
- subtyping becomes undecidable,
- ... 

**Solution:** invert \(<:\) only for closed types – no variables, no inconsistencies!
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

\[
X: \top \ldots \bot \vdash A \rightarrow B <: \top <: X <: \bot <: \forall Y: K. C : \ast
\]

NB. This causes all sorts of problems:
• subject reduction (preservation) fails,
• subtyping becomes undecidable,
• ...

Solution: invert only for closed types – no variables, no inconsistencies!

S. Stucki, P. G. Giarrusso
A Theory of Higher-Order Subtyping with Type Intervals
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

\[ X : \top \ldots \bot \vdash A \to B <: \top <: X <: \bot <: \forall Y : K. C : * \]

NB. This causes all sorts of problems:
- subject reduction (preservation) fails,
- subtyping becomes undecidable,
- …
Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

\[ X: \top \ldots \bot \vdash A \to B <: \top <: X <: \bot <: \forall Y: K. C : * \]

NB. This causes all sorts of problems:

- subject reduction (preservation) fails,
- subtyping becomes undecidable,
- \ldots

Solution: invert <: only for closed types – no variables, no inconsistencies!
Inversion – Step by Step

declarative

\[ \emptyset \vdash_d A \rightarrow B \triangleleft A' \rightarrow B' \]
Inversion – Step by Step

declarative

\[ \emptyset \vdash_d A \rightarrow B <: A' \rightarrow B' \]

canonical

\[ \emptyset \vdash_c U \rightarrow V <: U' \rightarrow V' \]

\[ U = \text{nf}(A), \ V = \text{nf}(B), \ldots \]
Inversion – Step by Step

declarative

\[ \emptyset \vdash_d A \rightarrow B <: A' \rightarrow B' \quad \xrightarrow{\text{nf}} \quad \emptyset \vdash_c U \rightarrow V <: U' \rightarrow V' \quad \xrightarrow{\simeq} \quad \vdash_{\text{tf}} U \rightarrow V <: U' \rightarrow V' \]

\[ \bullet \quad U = \text{nf}(A), \ V = \text{nf}(B), \ldots \]
Inversion – Step by Step

\[\emptyset \vdash_d A \rightarrow B <: A' \rightarrow B' \quad \text{nf} \quad \emptyset \vdash_c U \rightarrow V <: U' \rightarrow V' \quad \sim \quad \vdash_{tf} U \rightarrow V <: U' \rightarrow V' \]

\[\vdash_{tf} U' <: U\]
\[\vdash_{tf} V <: V'\]

\[U = \text{nf}(A), \ V = \text{nf}(B), \ldots\]
Inversion – Step by Step

declarative          canonical          transitivity-free

\( \emptyset \vdash_d A \to B <: A' \to B' \) \stackrel{\text{nf}}{\longrightarrow} \( \emptyset \vdash_c U \to V <: U' \to V' \) \stackrel{\sim}{\longrightarrow} \( \vdash_{\text{tf}} U \to V <: U' \to V' \)

\( \emptyset \vdash_c U' <: U \)
\( \emptyset \vdash_c V <: V' \) \stackrel{\sim}{\longrightarrow} \( \vdash_{\text{tf}} V <: V' \)

\( \vdash_{\text{tf}} U' <: U \)

\( U = \text{nf}(A), \ V = \text{nf}(B), \ldots \)
Inversion – Step by Step

**declarative**

\[ \emptyset \vdash_d A \to B <: A' \to B' \quad \text{nf} \quad \emptyset \vdash_c U \to V <: U' \to V' \quad \sim \quad \vdash_{tf} U \to V <: U' \to V' \]

\[ \emptyset \vdash_d A' = U' <: U = A \]

\[ \emptyset \vdash_d B = V <: V' = B' \quad \text{nf sound} \]

**canonical**

\[ \emptyset \vdash_c U' <: U \]

\[ \emptyset \vdash_c V <: V' \quad \sim \quad \vdash_{tf} V <: V' \]

**transitivity-free**

\[ \vdash_{tf} U' <: U \]

\[ \vdash_{tf} V <: V' \]

• \( U = \text{nf}(A), \ V = \text{nf}(B), \ldots \)

• \textbf{nf sound:} \( \Gamma \vdash A = \text{nf}_\Gamma(A) \) for all \( \Gamma \) and \( A \).

---

S. Stucki, P. G. Giarrusso

A Theory of Higher-Order Subtyping with Type Intervals
There’s More in the Paper…
There’s More in the Paper…

- Recap of the $F^\omega_<$ family and high-level intro to $F^\omega$ (with examples).
- Full presentation of $F^\omega$ (syntax, typing, SOS, …).
- Undecidability of subtyping. … and in the extended version (https://arxiv.org/abs/2107.01883).
- Additional definitions and lemmas. … and in the artifact (https://zenodo.org/record/5060213).
- Mechanization of the full metatheory!
There’s More in the Paper…

- Recap of the $F^\omega_<$ family and high-level intro to $F^\omega$ (with examples).
- Full presentation of $F^\omega$ (syntax, typing, SOS, …).

... and in the extended version ([https://arxiv.org/abs/2107.01883](https://arxiv.org/abs/2107.01883)).

... and in the artifact ([https://zenodo.org/record/5060213](https://zenodo.org/record/5060213)).
There's More in the Paper…

- Recap of the $F^\omega_<$ family and high-level intro to $F^\omega$ (with examples).
- Full presentation of $F^\omega$ (syntax, typing, SOS, …).
- Undecidability of subtyping.
There’s More in the Paper…

- Recap of the $F^{<\omega}$ family and high-level intro to $F^{\omega}$ (with examples).
- Full presentation of $F^{\omega}$ (syntax, typing, SOS, …).
- Undecidability of subtyping.

…and in the extended version (https://arxiv.org/abs/2107.01883)…

- Additional definitions and lemmas.
There’s More in the Paper…

- Recap of the $F^\omega_{<.}$ family and high-level intro to $F^\omega$ (with examples).
- Full presentation of $F^\omega$ (syntax, typing, SOS, …).
- Undecidability of subtyping.

…and in the extended version (https://arxiv.org/abs/2107.01883)…
- Additional definitions and lemmas.
- Human-readable proofs for (most) results.
There’s More in the Paper…

- Recap of the $F^\omega_{<}$ family and high-level intro to $F^\omega$ (with examples).
- Full presentation of $F^\omega$ (syntax, typing, SOS, …).
- Undecidability of subtyping.

…and in the extended version (https://arxiv.org/abs/2107.01883)…

- Additional definitions and lemmas.
- Human-readable proofs for (most) results.

…and in the artifact (https://zenodo.org/record/5060213).

- Mechanization of the full metatheory!
Thank you!

Coauthor
Paolo Giarrusso

Collaborators
- Guillaume Martres
- Nada Amin
- Martin Odersky
- Andreas Abel
- Jesper Cockx

Check out the Agda mechanization!

https://github.com/sstucki/f-omega-int-agda
https://zenodo.org/record/5060213
Except where otherwise noted, this work is licensed under

https://creativecommons.org/licenses/by/4.0/