A Theory of Higher-Order Subtyping with Type Intervals

Sandro Stucki   Paolo G. Giarrusso

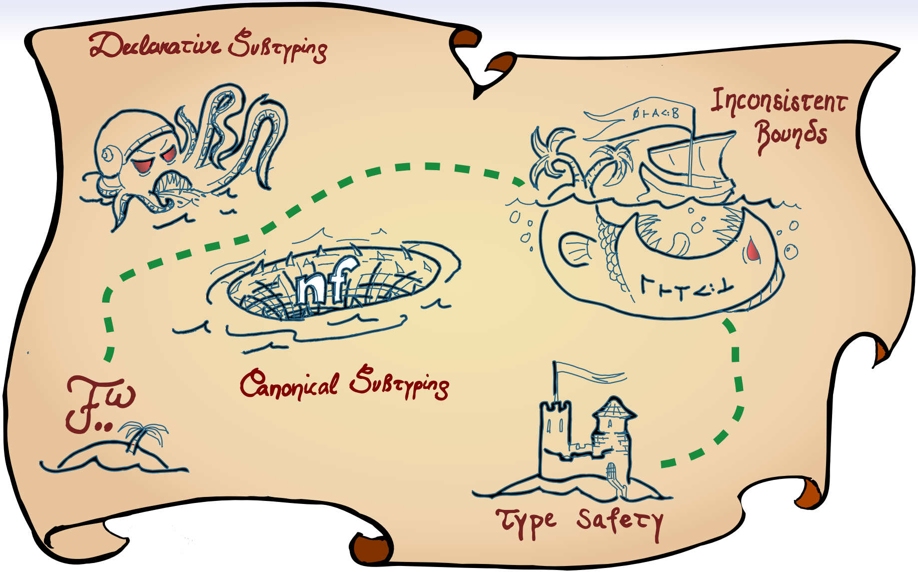
ICPF 2021 – 22–27 Aug 2021

sandros@chalmers.se   @stuckintheory

CHALMERS UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

BEDROCK Systems Inc.


Declarative Subtyping
Inconsistent Bounds
$\emptyset \vdash A <: B$
$\Gamma \vdash \top <: \bot$
nf
$F^{\omega}_{..}$
Canonical Subtyping
Type Safety

DOT and Dotty

DOT

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1]( ), Tiark Rompf[2],
and Sandro Stucki[1]

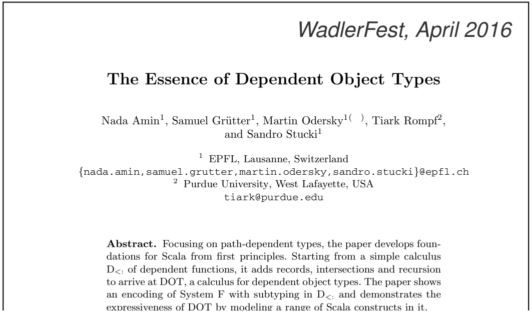
[1] EPFL, Lausanne, Switzerland
{nada.amin,samuel.grutter,martin.odersky,sandro.stucki}@epfl.ch
[2] Purdue University, West Lafayette, USA
tiark@purdue.edu

Abstract. Focusing on path-dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_{<:}$ 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 $D_{<:}$ 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



WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1]( ), Tiark Rompf[2], and Sandro Stucki[1]

[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 $D_{<:}$ 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 $D_{<:}$ 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)

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1], Tiark Rompf[2], and Sandro Stucki[1]

[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 $D_{<:}$ 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 $D_{<:}$ 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 Amin[1], Samuel Grütter[1], Martin Odersky[1]( ), Tiark Rompf[2], and Sandro Stucki[1]

[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 $D_{<:}$ 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 $D_{<:}$ 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 Amin[1], Samuel Grütter[1], Martin Odersky[1]( ), Tiark Rompf[2], and Sandro Stucki[1]

[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 $\mathrm{D}_{<:}$ 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 $\mathrm{D}_{<:}$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Dotty/Scala 3

Scala Symposium, Oct 2016

Implementing Higher-Kinded Types in Dotty

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

Abstract

dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kinded 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-kinded applications, without re-using 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

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1], Tiark Rompf[2], and Sandro Stucki[1]

[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 $D_{<:}$ 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 $D_{<:}$ 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, Switerland: {first.last}@epfl.ch

Abstract

dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kinded 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-kinded applications, without reusing much of the existing concepts of the calculus and the compiler.

DOT and Dotty

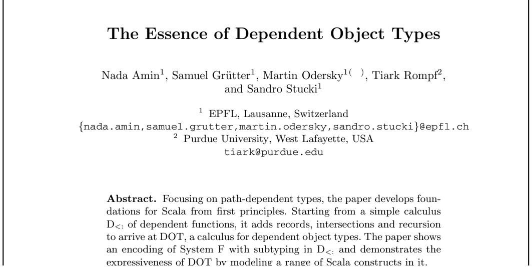
DOT

- 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

WadlerFest, April 2016



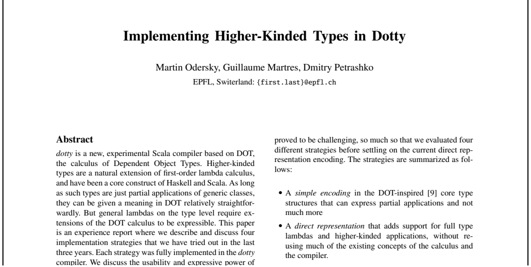
The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1]( ), Tiark Rompf[2], and Sandro Stucki[1]

[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 $D_{<:}$ 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 $D_{<:}$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Scala Symposium, Oct 2016



Implementing Higher-Kinded Types in Dotty

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

Abstract

dotty is a new, experimental Scala compiler based on DOT, the calculus of Dependent Object Types. Higher-kinded 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-kinded 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

WadlerFest, April 2016

The Essence of Dependent Object Types

Nada Amin[1], Samuel Grütter[1], Martin Odersky[1], Tiark Rompf[2], and Sandro Stucki[1]

[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 $D_{<:}$ 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 $D_{<:}$ 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-kinded 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-kinded applications, without re-using much of the existing concepts of the calculus and the compiler.

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

```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 >: 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

```
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).

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 >: 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

```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 >: 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.

The Anatomy of a Type Interval

```
X >: A <: B
```

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 $\{\, A <: \cdots <: B \,\} = A \mathrel{..} B$

The Anatomy of a Type Interval

`X >: A <: B` $\qquad\qquad X : A \mathinner{..} B$

Intuition: X is an element of the set of types $\{\, A <: \cdots <: B \,\} = A \mathinner{..} B$

The Anatomy of a Type Interval

`X >: A <: B` $\qquad X : A \mathinner{..} B$

Intuition: X is an element of the set of types $\{ A <: \cdots <: B \} = A \mathinner{..} B$

Special cases

Upper bound `X <: B` $\qquad X : \bot \mathinner{..} B$

- $\bot =$ `Nothing` $=$ minimal/bottom type;

The Anatomy of a Type Interval

`X >: A <: B` $\qquad$ $X : A \mathrel{..} B$

Intuition: X is an element of the set of types $\{\, A <: \cdots <: B \,\} = A \mathrel{..} B$

Special cases

Upper bound	`X <: B`	$X : \bot \mathrel{..} B$
Lower bound	`X >: A`	$X : A \mathrel{..} \top$

- $\bot =$ `Nothing` $=$ minimal/bottom type;
- $\top =$ `Any` $=$ maximal/top type;

The Anatomy of a Type Interval

`X >: A <: B` $\qquad X : A \mathinner{..} B$

Intuition: X is an element of the set of types $\{ A <: \cdots <: B \} = A \mathinner{..} B$

Special cases

Upper bound	`X <: B`	$X : \bot \mathinner{..} B$
Lower bound	`X >: A`	$X : A \mathinner{..} \top$
Abstract	`X`	$X : \bot \mathinner{..} \top$

- $\bot$ = `Nothing` = minimal/bottom type;
- $\top$ = `Any` = maximal/top type;
- $\bot \mathinner{..} \top = * =$ kind of all types.

The Anatomy of a Type Interval

`X >: A <: B` $\qquad\qquad X : A \mathrel{..} B$

Intuition: X is an element of the set of types $\{\, A <: \cdots <: B \,\} = A \mathrel{..} B$

Special cases

Upper bound	`X <: B`	$X : \bot \mathrel{..} B$
Lower bound	`X >: A`	$X : A \mathrel{..} \top$
Abstract	`X`	$X : \bot \mathrel{..} \top$
Alias	`X = A`	$X : A \mathrel{..} A$

- $\bot =$ `Nothing` $=$ minimal/bottom type;
- $\top =$ `Any` $=$ maximal/top type;
- $\bot \mathrel{..} \top = * =$ kind of all types.
- $A \mathrel{..} A =$ 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` $\qquad F : (X{:}A\,..\,B) \to G\,..\,H$

We can also represent bounded operators

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	`F1[X] = List[X]`	$F_1 : (X{:}*) \to \mathsf{List}\,X \,..\, \mathsf{List}\,X$

The Anatomy of a Type Interval (cont.)

`F[X >: A <: B] >: G <: H` $F : (X{:}A \mathinner{..} B) \to G \mathinner{..} H$

We can also represent bounded operators

Examples

Alias	`F1[X] = List[X]`	$F_1 : (X{:}*) \to \mathsf{List}\, X \mathinner{..} \mathsf{List}\, X$
Upper bound	`F2[X] <: List[X]`	$F_2 : (X{:}*) \to \bot \mathinner{..} \mathsf{List}\, X$

The Anatomy of a Type Interval (cont.)

`F[X >: A <: B] >: G <: H` $\qquad F : (X{:}A\,..\,B) \to G\,..\,H$

We can also represent bounded operators

Examples

Alias	`F1[X] = List[X]`	$F_1 : (X{:}*) \to \mathsf{List}\,X\,..\,\mathsf{List}\,X$
Upper bound	`F2[X] <: List[X]`	$F_2 : (X{:}*) \to \bot\,..\,\mathsf{List}\,X$
HO bounded op.	`F3[X, Y[_ <: X]]`	$F_3 : (X{:}*) \to (Y{:}(\_{:}\bot\,..\,X) \to *) \to *$

The Anatomy of a Type Interval (cont.)

`F[X >: A <: B] >: G <: H` $\qquad F : (X{:}A\,..\,B) \to G\,..\,H$

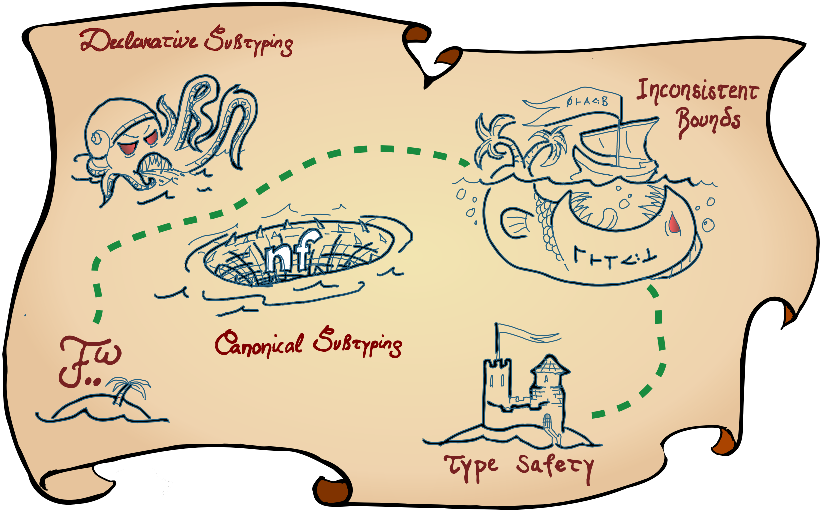
We can also represent bounded operators

Examples

Alias	`F1[X] = List[X]`	$F_1 : (X{:}*) \to \mathsf{List}\,X\,..\,\mathsf{List}\,X$
Upper bound	`F2[X] <: List[X]`	$F_2 : (X{:}*) \to \bot\,..\,\mathsf{List}\,X$
HO bounded op.	`F3[X, Y[_ <: X]]`	$F_3 : (X{:}*) \to (Y{:}(\_{:}\bot\,..\,X) \to *) \to *$

NB. The operators $F_1 - F_3$ all have dependent kinds.

Proving Type Safety of $F^{\omega}_{\cdot\cdot}$



Proving Type Safety of $F^{\omega}_{..}$

The big challenge is to prove subtyping inversion.

Proving Type Safety of $F^{\omega}_{\cdot\cdot}$

The big challenge is to prove subtyping inversion.

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

Proving Type Safety of $F^{\omega}_{\cdot\cdot}$

The big challenge is to prove subtyping inversion.

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

Main sub-challenges:

1. Subtyping derivations may involve computation ($\beta\eta$-conversions).

Proving Type Safety of $F^{\omega}_{\cdot\cdot}$

The big challenge is to prove subtyping inversion.

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

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.

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

Main sub-challenges:

1. Subtyping derivations may involve computation ($\beta\eta$-conversions).
2. Subtyping derivations may involve subsumption (via subkinding).
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.

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

Main sub-challenges:

1. Subtyping derivations may involve computation ($\beta\eta$-conversions).
2. Subtyping derivations may involve subsumption (via subkinding).
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.

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

Challenge 1: Getting Rid of $\beta\eta$-Conversions

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

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

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.

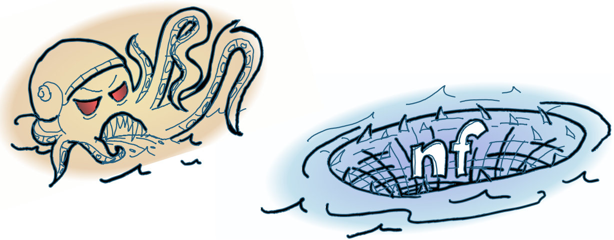
$$\frac{\Gamma \vdash Z : (X{:}J) \to K \qquad \Gamma \vdash V : J}{\Gamma \vdash Z\, V : K[V/X]}$$

Challenge 1: Getting Rid of $\beta\eta$-Conversions

New problem: dependent kinding of applications involves substitutions.

$$\frac{\Gamma \vdash Z : (X{:}J) \to K \qquad \Gamma \vdash V : J}{\Gamma \vdash Z\,V : K[V/X]}$$

Challenge 1: Getting Rid of $\beta\eta$-Conversions

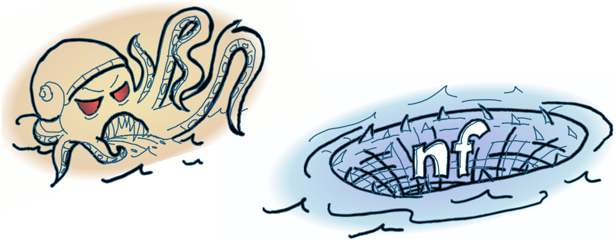


New problem: dependent kinding of applications involves substitutions.

$$\frac{\Gamma \vdash Z : (X{:}J) \to K \qquad \Gamma \vdash V : J}{\Gamma \vdash Z\,V : K[V/X^{|J|}]}$$

New solution: use hereditary substitution

Challenge 1: Getting Rid of $\beta\eta$-Conversions



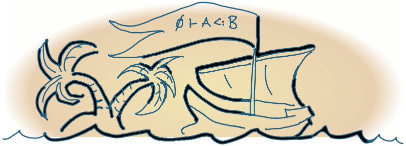
New problem: dependent kinding of applications involves substitutions.

$$\frac{\Gamma \vdash Z : (X{:}J) \to K \qquad \Gamma \vdash V : J}{\Gamma \vdash Z\,V : K[V/X^{|J|}]}$$

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

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce arbitrary subtyping relationships.



Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

$$X{:}\ \top\ ..\ \bot\ \ \vdash\qquad\qquad X\qquad\qquad\qquad :\ *$$

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

$$X\colon \top \,..\, \bot \;\vdash\; \top <: X \qquad : *$$

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

$$X\colon \top \,..\, \bot \;\vdash\; A \to B \;<:\; \top \;<:\; X \qquad\qquad\qquad : \;*$$

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

$$X\colon \top \,..\, \bot \;\vdash\; A \to B \;<:\; \top \;<:\; X \;<:\; \bot \qquad\qquad : \;*$$

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

$$X\colon \top \mathinner{..} \bot \;\; \vdash \;\; A \to B \;<:\; \top \;<:\; X \;<:\; \bot \;<:\; \forall Y{:}K.\,C \;\;:\;\; *$$

Challenge 3: Inconsistent Bounds

Problem: Type variables can introduce inconsistent subtyping relationships.

$$X\colon \top \mathrel{..} \bot \;\vdash\; A \to B \;<:\; \top \;<:\; X \;<:\; \bot \;<:\; \forall Y{:}K.\,C \;:\; *$$

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.

$$X\colon \top \,..\, \bot \;\vdash\; A \to B \;<:\; \top \;<:\; X \;<:\; \bot \;<:\; \forall Y\!:\!K.\,C \;:\; *$$

NB. This causes all sorts of problems:

- subject reduction (preservation) fails,
- subtyping becomes undecidable,
- …

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

Inversion – Step by Step

declarative

$$\varnothing \vdash_{\mathsf{d}} A \to B <: A' \to B'$$

Inversion – Step by Step

declarative canonical

$$\varnothing \vdash_{\mathsf{d}} A \to B <: A' \to B' \xrightarrow{\mathsf{nf}} \varnothing \vdash_{\mathsf{c}} U \to V <: U' \to V'$$

- $U = \mathsf{nf}(A)$, $V = \mathsf{nf}(B)$, ...

Inversion – Step by Step

declarative | canonical | transitivity-free

$$\varnothing \vdash_{\mathsf{d}} A \to B <: A' \to B' \xrightarrow{\mathsf{nf}} \varnothing \vdash_{\mathsf{c}} U \to V <: U' \to V' \xrightarrow{\simeq} \vdash_{\mathsf{tf}} U \to V <: U' \to V'$$

- $U = \mathsf{nf}(A)$, $V = \mathsf{nf}(B)$, $\ldots$

Inversion – Step by Step

declarative canonical transitivity-free

$$\varnothing \vdash_{\mathsf{d}} A \to B <: A' \to B' \xrightarrow{\text{nf}} \varnothing \vdash_{\mathsf{c}} U \to V <: U' \to V' \xrightarrow{\simeq} \vdash_{\mathsf{tf}} U \to V <: U' \to V'$$

$$\big\downarrow \text{invert}$$

$$\vdash_{\mathsf{tf}} U' <: U$$
$$\vdash_{\mathsf{tf}} V <: V'$$

- $U = \mathsf{nf}(A)$, $V = \mathsf{nf}(B)$, ...

Inversion – Step by Step

$$
\begin{tikzcd}
\text{declarative} & \text{canonical} & \text{transitivity-free} \\
\varnothing \vdash_{\mathsf{d}} A \to B <: A' \to B' \arrow[r, "\mathsf{nf}"] & \varnothing \vdash_{\mathsf{c}} U \to V <: U' \to V' \arrow[r, "\simeq"] & \vdash_{\mathsf{tf}} U \to V <: U' \to V' \arrow[d, "\text{invert}"] \\
& \begin{array}{l} \varnothing \vdash_{\mathsf{c}} U' <: U \\ \varnothing \vdash_{\mathsf{c}} V \ <: V' \end{array} & \begin{array}{l} \vdash_{\mathsf{tf}} U' <: U \\ \vdash_{\mathsf{tf}} V \ <: V' \end{array} \arrow[l, "\simeq"']
\end{tikzcd}
$$

- $U = \mathsf{nf}(A)$, $V = \mathsf{nf}(B)$, $\ldots$

Inversion – Step by Step

$$
\begin{tikzcd}[column sep=large]
\text{declarative} & \text{canonical} & \text{transitivity-free} \\
\varnothing \vdash_{\mathsf{d}} A \to B <: A' \to B' \arrow[r, "\mathsf{nf}"] \arrow[d, dashed] & \varnothing \vdash_{\mathsf{c}} U \to V <: U' \to V' \arrow[r, "\simeq"] & \vdash_{\mathsf{tf}} U \to V <: U' \to V' \arrow[d, "\text{invert}"] \\
\begin{array}{l} \varnothing \vdash_{\mathsf{d}} A' = U' <: U = A \\ \varnothing \vdash_{\mathsf{d}} B = V <: V' = B' \end{array} & \begin{array}{l} \varnothing \vdash_{\mathsf{c}} U' <: U \\ \varnothing \vdash_{\mathsf{c}} V <: V' \end{array} \arrow[l, "\text{nf sound}"'] & \begin{array}{l} \vdash_{\mathsf{tf}} U' <: U \\ \vdash_{\mathsf{tf}} V <: V' \end{array} \arrow[l, "\simeq"']
\end{tikzcd}
$$

- $U = \mathsf{nf}(A)$, $V = \mathsf{nf}(B)$, ...
- nf sound: $\Gamma \vdash A = \mathsf{nf}_\Gamma(A)$ for all Γ and A.

There's More in the Paper...

There's More in the Paper. . .

- Recap of the $F^{\omega}_{<:}$ family and high-level intro to $F^{\omega}_{\cdot\cdot}$ (with examples).

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, . . .).

There's More in the Paper…

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

There's More in the Paper...

- Recap of the $F^{\omega}_{<:}$ family and high-level intro to $F^{\omega}_{\cdot\cdot}$ (with examples).
- Full presentation of $F^{\omega}_{\cdot\cdot}$ (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 $F^{\omega}_{\cdot\cdot}$ (with examples).
- Full presentation of $F^{\omega}_{\cdot\cdot}$ (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`