



# Tracking how dependently-typed functions use their arguments

Stephanie Weirich  
University of Pennsylvania  
Philadelphia, USA



# Collaborators

## Yiyun Liu

Jonathan Chan  
Jessica Shi  
Pritam Choudhury



Yiyun: DCOI: *Dependent Calculus of Indistinguishability*

- POPL 24 – DCOI the language
- POPL 25 – DCOI the logic
- POPL 26 – Equality w/ surjective pairing

Forthcoming dissertation *Dependency tracking and Dependent types*



Let's talk about constant functions

---

$\text{id} : \forall (A : \text{Type}) \rightarrow A \rightarrow A$

$\text{id} = \lambda A x. x$

$\text{id} = \lambda \_ x. x$

Erasure semantics for  
type polymorphism

# Erasure semantics for type polymorphism

---

```
data List (A : Type) : Type where
```

```
  Nil  : List A
```

```
  Cons : A → List A → List A
```

```
map : ∀ (A B : Type) → (A → B) → List A → List B
```

```
map = λ A B f xs.
```

```
  case xs of
```

```
    Nil ⇒ Nil
```

```
    Cons y ys ⇒ Cons (f y) (map A B f ys)
```

# Erasure semantics for type polymorphism

---

**data**

Nil

Cons

```
map = λ _ _ f xs.  
      case xs of  
        Nil ⇒ Nil  
        Cons y ys ⇒ Cons (f y) (map _ _ f ys)
```

# Erasure in **dependently-typed** languages

---

```
data Vec (n:Nat) (A:Type) : Type where
  Nil    : Vec Zero A
  Cons   :  $\Pi(m:Nat)$   $\rightarrow$  A  $\rightarrow$  (Vec m A)  $\rightarrow$  Vec (Succ m) A
```

```
map :  $\forall(A B : Type)$   $\rightarrow$   $\Pi(n:Nat)$   $\rightarrow$  (A  $\rightarrow$  B)
      $\rightarrow$  Vec n A  $\rightarrow$  Vec n B
```

```
map =  $\lambda$  A B n f xs.
      case xs of
        Nil  $\Rightarrow$  Nil
        Cons m y ys  $\Rightarrow$ 
          Cons m (f y) (map A B m f ys)
```

# Erasure in **dependently-typed** languages

---

```
data Vec (n:Nat) (A:Type) : Type where
  Nil    : Vec Zero A
  Cons   :  $\forall(m:Nat)$   $\rightarrow$  A  $\rightarrow$  (Vec m A)  $\rightarrow$  Vec (Succ m) A
```

```
map :  $\forall(A B : Type)$   $\rightarrow$   $\forall(n:Nat)$   $\rightarrow$  (A  $\rightarrow$  B)
      $\rightarrow$  Vec n A  $\rightarrow$  Vec n B
```

```
map =  $\lambda$  A B n f xs.
      case xs of
        Nil  $\Rightarrow$  Nil
        Cons m y ys  $\Rightarrow$ 
          Cons m (f y) (map A B m f ys)
```

# Erasure in **dependently-typed** languages

---

**data**

Nil

Cons

```
map = λ _ _ _ f xs.  
      case xs of  
        Nil ⇒ Nil  
        Cons _ y ys ⇒  
            Cons _ (f y) (map _ _ _ f ys)
```

# Refinement/Subset types

---

```
type EvenNat = { n : Nat | isEven n } Eraseable proof
```

```
plusIsEven : Π(m n : Nat) → isEven m → isEven n  
            → isEven (m + n)
```

```
plusIsEven = λ m n p1 p2. ...
```

```
plus : EvenNat → EvenNat → EvenNat
```

```
plus = λ en em. case en, em of
```

```
    (n, np), (m, mp) ⇒
```

```
    (n + m, plusIsEven n m np mp)
```

# Refinement/Subset types

---

```
plus = λ en em. case en, em of  
      (n, _ ), (m, _ ) ⇒  
        (n + m, _ )
```

# Erased code is irrelevant

---

- Not all terms are needed for computation: some function arguments and data structure components are there only for type checking
- Especially common in dependently-typed programming and proving
- Can call such code *irrelevant*

# Why care about irrelevance?

---

1. The compiler can produce faster code
  - No need to compute erased arguments      Run-time irrelevance
2. The type checker can run more quickly
  - Comparing types for equality requires reduction, which can be sped up by erasure
3. Verification is less work for programmers
  - Proving that terms are equal is easier when you can ignore the irrelevant parts
4. More programs type check
  - May not be able to prove the irrelevant parts equal

Compile-time irrelevance

## Less work for verification: proof irrelevance

---

```
type EvenNat = { n : Nat | isEven n }

-- prove equality of two EvenNats
congEvenNat : (n m : Nat)
  → (np : isEven n)
  → (mp : isEven m)
  → (n = m)
  -- no need for proof of np = mp
  → ((n, np) = (m, mp) : EvenNat)
congEvenNat = λ n m en em p. ...
```

# More programs type check

---

...when more terms are equal, *by definition*

type tells us that  $f$  is a constant function

example :  $\forall (f : \forall (x : \text{Bool}) \rightarrow \text{Bool})$   
 $\rightarrow (f \text{ True} = f \text{ False})$

example =  $\lambda f . \text{Refl}$

Sound for the type checker to decide that these terms are equal

Proof of equality comes directly from type checker

Type checkers identify irrelevant code

---

But how?

How should type checkers identify and take advantage of irrelevant code?

---

- 1. Erasure**
- 2. Modes**
- 3. Dependency**
- 4. Many other ways (grades, sorts, truncation, etc)**

# Core dependent type system

---

$\Gamma \vdash a : A$

$$\text{VAR} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\text{PI} \quad \frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi x : A. B : \star}$$

$$\text{ABS} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \Pi x : A. B : \star}{\Gamma \vdash \lambda x : A. b : \Pi x : A. B}$$

$$\text{APP} \quad \frac{\Gamma \vdash b : \Pi x : A. B \quad \Gamma \vdash a : A}{\Gamma \vdash b a : B[a/x]}$$

$$\text{CONV} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash B : \star \quad \vdash A \equiv B}{\Gamma \vdash a : B}$$

# Erasure

---

You can't use something  
that is not there

Miquel, TLCA 01  
**Barras and Bernardo, FoSSaCS 2008**

Zombie/Trellys [Kimmel et al. MSFP 2012]  
Dependent Haskell [Weirich et al. ICFP 2018]

# ICC: Implicit Calculus of Constructions

---

- Extend core language with irrelevant (implicit) abstractions

E-ABS

$$\frac{\begin{array}{l} \Gamma, x : A \vdash b : B \\ \Gamma \vdash \forall x : A. B : \star \\ x \notin \text{fv}|b| \end{array}}{\Gamma \vdash \lambda x : \mathbf{I} A. b : \forall x : A. B}$$

E-APP

$$\frac{\begin{array}{l} \Gamma \vdash b : \forall x : A. B \\ \Gamma \vdash a : A \end{array}}{\Gamma \vdash b a^{\mathbf{I}} : B[a/x]}$$

# ICC: Implicit Calculus of Constructions

- Extend core language with irrelevant (implicit) abstractions
- Annotations enable decidable type checking

E-ABS

$$\frac{\begin{array}{l} \Gamma, x : A \vdash b : B \\ \Gamma \vdash \forall x : A. B : \star \\ x \notin \text{fv}|b| \end{array}}{\Gamma \vdash \lambda x : \mathbf{I} A. b : \forall x : A. B}$$

E-APP

$$\frac{\begin{array}{l} \Gamma \vdash b : \forall x : A. B \\ \Gamma \vdash a : A \end{array}}{\Gamma \vdash b \mathbf{I} a : B[a/x]}$$

# ICC: Implicit Calculus of Constructions

- Extend core language with irrelevant (implicit) abstractions
- Annotations enable decidable type checking
- Irrelevant parameters must not appear *relevantly*
- Erasure operation  $|a|$  removes irrelevant terms

E-ABS

$$\begin{array}{l} \Gamma, x : A \vdash b : B \\ \Gamma \vdash \forall x : A. B : \star \end{array}$$

$$x \notin \text{fv}|b|$$

$$\frac{}{\Gamma \vdash \lambda x : \mathbf{I} A. b : \forall x : A. B}$$

E-APP

$$\Gamma \vdash b : \forall x : A. B$$

$$\Gamma \vdash a : A$$

$$\frac{}{\Gamma \vdash b a^{\mathbf{I}} : B[a/x]}$$

# Erasure during conversion

---

- Conversion between *erased* types
- Compile-time irrelevance: erased parts ignored when comparing types for equality

E-CONV-ANN

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : \star \quad \vdash |A| \equiv |B|}{\Gamma \vdash a : B}$$

# Drawback: no irrelevant projections

```
filter :  $\forall(A:\text{Type}) \rightarrow \forall(n:\text{Nat})$   
         $\rightarrow (A \rightarrow \text{Bool}) \rightarrow (\text{Vec } n \ A) \rightarrow \exists(m:\text{Nat}) \times (\text{Vec } m \ A)$ 
```

```
filter =  $\lambda A \ n \ f \ v.$ 
```

```
  case v of
```

```
    Nil  $\Rightarrow$ 
```

```
      (0, Nil)
```

```
    Cons m x xs  $\Rightarrow$ 
```

```
      let p = filter A m f
```

```
      if f x
```

```
        then (Succ p.1, Cons p.1 x p.2)
```

```
        else p
```

Length is not statically known and irrelevant.

UNSOUND: Irrelevant first projection means that it is erased during conversion  
 $|\text{Vec } p.1 \ A| = \text{Vec } ?? \ A$

# Modes

---

Distinguish relevant and irrelevant abstractions through  
*modes*

Pfenning, LICS 01

**Mishra-Linger and Sheard, FoSSaCS 08**

Abel and Scherer, LMCS 12

DDC, Choudhury and Weirich, ESOP 22

DE, Liu and Weirich, ICFP 23

# Modes

---

- Type system controlled by modes:  $m ::= \mathbf{R} \mid \mathbf{I}$ 
  - Variables have modes, must be  $\mathbf{R}$  when used
  - $\Gamma ::= \varepsilon \mid \Gamma, x :^m A$
  - Resurrection ( $\Gamma^m$ ): replaces all  $m$  modes with  $\mathbf{R}$
  - Mode-annotated quantification:  
 $\Pi x :^m A.B$  unifies  $\Pi x : A.B$  and  $\forall x : A.B$
- Modal type marks irrelevant code:  $\Box A$

# Mode-annotated functions

Only relevant variables can be used

$$\frac{\text{M-VAR} \quad x :^{\mathbf{R}} A \in \Gamma}{\Gamma \vdash x : A}$$

$\Pi$ -bound variables always relevant in the type

$$\frac{\text{M-PI} \quad \begin{array}{l} \Gamma \vdash A : \star \\ \Gamma, x :^{\mathbf{R}} A \vdash B : \star \end{array}}{\Gamma \vdash \Pi x :^m A. B : \star}$$

Types checked with "resurrected" context

$$\frac{\text{M-ABS} \quad \begin{array}{l} \Gamma^{\mathbf{I}} \vdash \Pi x :^m A. B : \star \\ \Gamma, x :^m A \vdash b : B \end{array}}{\Gamma \vdash \lambda x :^m A. a : \Pi x :^m A. B}$$

$$\frac{\text{M-APP} \quad \begin{array}{l} \Gamma \vdash b : \Pi x :^m A. B \\ \Gamma^m \vdash a : A \end{array}}{\Gamma \vdash b a^m : B[a/x]}$$

Irrelevant arguments checked with resurrected context

Mode on  $\Pi$ -type determines mode in the context

$$\frac{\text{M-CONV} \quad \begin{array}{l} \Gamma \vdash a : A \\ \vdash A \equiv B \end{array}}{\Gamma \vdash a : B}$$

Conversion ignores irrelevant arguments

# Conversion ignores irrelevant arguments

---

- Usual rules, plus
  - compare arguments marked **R**
  - ignore arguments marked **I**

EQ-REL

$$\frac{\begin{array}{l} \vdash b_1 \equiv b_2 \\ \vdash a_1 \equiv a_2 \end{array}}{\vdash b_1 a_1^{\mathbf{R}} \equiv b_2 a_2^{\mathbf{R}}}$$

EQ-IRR

$$\frac{\vdash b_1 \equiv b_2}{\vdash b_1 a_1^{\mathbf{I}} \equiv b_2 a_2^{\mathbf{I}}}$$

# Modes for irrelevance

---

- Benefits
  - Modes generalize  $\Pi$  and  $\forall$
  - Easy implementation: mark variables when introduced in the context, mark the context for resurrection
- Drawbacks
  - No irrelevant projections
  - Formation rule for  $\Pi$ -types looks a bit strange

## Alternative rule for $\Pi$ -types

---

- From [Pfenning 99] [Abel and Scherer 2012]
- Irrelevant arguments must be irrelevant *everywhere* including in types. No parametric polymorphism!

M-PI

$$\frac{\begin{array}{c} \Gamma \vdash A : \star \\ \Gamma, x :^{\mathbf{R}} A \vdash B : \star \end{array}}{\Gamma \vdash \Pi x :^m A. B : \star}$$

M-PI-ALT

$$\frac{\begin{array}{c} \Gamma \vdash A : \star \\ \Gamma, x :^m A \vdash B : \star \end{array}}{\Gamma \vdash \Pi x :^m A. B : \star}$$

# Dependency

---

Track when outputs depend on inputs

DCC, Abadi et al., POPL 99

DDC, Choudhury and Weirich, ESOP 22  
**DCOI, Liu, Chan, Shi, Weirich, POPL 24, 25**

# Dependency tracking

---

- Typing judgment ensures that low-level outputs do not depend on high-level inputs

$$x :^H \text{Bool} \vdash a :^L \text{Int}$$

Input level

x can only be used when  
observer level is  $\geq H$

Observer level

a can only use variables whose  
levels are  $\leq L$

- Type system parameterized by ordered set of levels
  - Relevance ( $R < I$ )
  - *Other examples:* Security levels ( $\text{Low} < \text{Med} < \text{High}$ )  
Staged computation ( $0 < 1 < 2\dots$ )

# Typing rules with dependency levels

$$\Gamma \vdash a :^\ell A$$

D-VAR

$$\frac{x :^m A \in \Gamma}{\Gamma \vdash x :^\ell A}$$

Variable usage restricted by observer level

$$m \leq \ell$$

D-PI

$$\frac{\Gamma \vdash A :^\ell \star \quad \Gamma, x :^m A \vdash B :^\ell \star}{\Gamma \vdash \Pi x :^m A. B :^\ell \star}$$

Vars have same level in terms and types

D-ABS

$$\frac{\Gamma, x :^m A \vdash b :^\ell B \quad \Gamma \vdash \Pi x :^m A. B :^{\ell_1} \star}{\Gamma \vdash \lambda x :^m A. b :^\ell \Pi x :^m A. B}$$

Terms do not observe types, so level unimportant

D-APP

$$\frac{\Gamma \vdash b :^\ell \Pi x :^m A. B \quad \Gamma \vdash a :^m A}{\Gamma \vdash b a^m :^\ell B[a/x]}$$

$\Pi$ -types record the dependency levels of their arguments

Application requires compatible dependency levels

# Indistinguishability: indexed definitional equality

$$\vdash a \equiv^{\ell} b$$

Observer at level  $\ell$  cannot distinguish between terms

If observer has a higher level than the argument, arguments must agree

If observer does not have a higher level, arguments are ignored

EQ-D-APP-DIST

$$\frac{\vdash b_0 \equiv^{\ell} b_1 \quad \vdash a_0 \equiv^{\ell} a_1 \quad \ell_0 \leq \ell}{\vdash b_0 a_0^{\ell_0} \equiv^{\ell} b_1 a_1^{\ell_0}}$$

EQ-D-APP-INDIST

$$\frac{\vdash b_0 \equiv^{\ell} b_1 \quad \ell_0 \not\leq \ell}{\vdash b_0 a_0^{\ell_0} \equiv^{\ell} b_1 a_1^{\ell_0}}$$

Conversion can be used at **any** observer level

---

$$\frac{\text{D-CONV} \quad \Gamma \vdash a :^{\ell} A \quad \Gamma \vdash B :^{\ell_0} \star \quad \vdash A \equiv^{\ell_0} B}{\Gamma \vdash a :^{\ell} B}$$

Type system is **sound** because we cannot equate types with different head forms at *any* dependency level

# DCOI: irrelevant projections

`filter` :  $(A : \mathbb{I} \text{ Type}) \rightarrow (n : \mathbb{I} \text{ Nat})$       Type is checked with I-observer  
           $\rightarrow (A \rightarrow \text{Bool}) \rightarrow (\text{Vec } n \text{ A}) \rightarrow (m : \mathbb{I} \text{ Nat}) \times (\text{Vec } m \text{ A})$

`filter` =  $\lambda A \ n \ f \ v.$

**case** `v` **of**      Definition checks with R observer, but  
    `Nil`  $\Rightarrow (0^{\mathbb{I}}, \text{Nil})$       contains I-marked subterms

`Cons`  $m^{\mathbb{I}} \ x \ xs \Rightarrow$

**let** `p` = `filter`  $A^{\mathbb{I}} \ m^{\mathbb{I}} \ f \ xs$  **in**

**if** `f` `x`

**then**  $((\text{Succ } p.1)^{\mathbb{I}}, \text{Cons } p.1^{\mathbb{I}} \ x \ p.2)$

**else** `p`      First projection allowed in  
                            I-marked subterms only

# DCOI: Dependent Calculus of Indistinguishability

---

- **Yiu, Chan, Shi and Weirich. *Internalizing Indistinguishability with Dependent Types*. POPL 2024**
  - Based on Pure Type System (PTS)
  - Key results: Syntactic type soundness, noninterference
- **Yiu, Chan and Weirich. *Consistency of a Dependent Calculus of Indistinguishability*. POPL 2025**
  - Predicative universe hierarchy
  - Observer-indexed propositional equality, J-eliminator
  - Key results: Consistency, normalization, and decidable (observer-indexed) equality
- All results mechanized using Rocq

# DCOI: Proof-specific features

False-elimination

$$\frac{\text{D-ABSURD} \quad \Gamma \vdash a :^m \perp \quad \Gamma \vdash A :^{\ell_1} \star}{\Gamma \vdash \mathbf{absurd} \ a :^\ell A}$$

Proof of false can be eliminated at *any* observer level.

Propositional indistinguishability

D-EQ

$$\frac{\Gamma \vdash a :^m A \quad \Gamma \vdash b :^m A \quad m \leq \ell}{\Gamma \vdash a =^m b :^\ell \star}$$

Equality must be observable to the type

D-REFL

$$\frac{\Gamma \vdash a :^m A}{\Gamma \vdash \mathbf{refl} :^\ell a =^m a}$$

Can create reflexivity proofs about any level

D-J

$$\frac{\Gamma \vdash a :^{\ell_0} a_1 =^m a_2 \quad \Gamma, x :^m A, y :^{\ell_0} a_1 =^m x \vdash B :^m \star \quad \Gamma \vdash b :^\ell B[a_1/x][\mathbf{refl}/y] \quad \ell_0 \leq \ell}{\Gamma \vdash \mathbf{J} \ a \ b :^\ell B[a_2/x][a/y]}$$

Equality between terms at level  $m$  witnessed via substitution at level  $m$   
 Equality between proofs at level  $\ell_0$  witnessed via substitution at level  $\ell_0$

Level of proof must be less than current level

# DCOI: Dependent Calculus of Indistinguishability

---

- Benefits
  - Irrelevant projection is sound!
  - Irrelevant absurdity is consistent!
  - Can reason about *indistinguishability* as a proposition
- Open questions
  - Compatibility with type-directed equality?
  - Relationship with sProp? Ghost Type Theory?
  - Irrelevance for single-constructor eliminators?
  - Level quantification?
  - Applications besides irrelevance? (Metaprogramming, axiom usage, security)