# Tracking how dependently-typed functions use their arguments

Stephanie Weirich

University of Pennsylvania
Philadelphia, USA

Penn Engineering
University of Pennsylvania

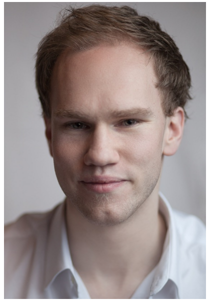
# Collaborators

Yiyun Liu
Jonathan Chan
Jessica Shi
Pritam Choudhury
Richard Eisenberg
Harley Eades III
Antoine Voizard
Pedro Henrique Avezedo de Amorim
Anastasiya Kravchuk-Kirilyuk









Penn Engineering
UNIVERSITY of PENNSYLVANIA

# Let's talk about constant functions

```
id : ∀ (A : Type) → A → A
id = λ A x. x
```

```
id = λ _ x. x
```

Erasure semantics for type polymorphism

# Erasure semantics for 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 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 : Π(m:Nat) → A → (Vec m A) → Vec (Succ m) A

map : ∀(A B : Type) → ∀(n : Nat) → (A → B)
    → Vec n A → Vec n B
map = λ A B n f v.
        case v of
          Nil ⇒ Nil
          Cons m x xs ⇒
              Cons m (f x) (map A B m f xs)
```

# Erasure in dependently-typed languages

```
data Vec (n:Nat) (A:Type) : Type where
  Nil  : Vec Zero A
  Cons : ∀(m:Nat) → A → (Vec m A) → Vec (Succ m) A
```

Erasable data

```
map : ∀(A B : Type) → ∀(n : Nat) → (A → B)
    → Vec n A → Vec n B
map = λ A B n f v.
        case v of
          Nil ⇒ Nil
          Cons m x xs ⇒
              Cons m (f x) (map A B m f xs)
```

# Erasure in dependently-typed languages

```
data
  Nil
  Cons




map = λ _ _ _ f v.
        case v of
          Nil ⇒ Nil
          Cons _ x xs ⇒
              Cons _ (f x) (map _ _ _ f xs)
```

# Refinement/Subset types

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

Erasable 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, _)
```

# Erasable code is irrelevant

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

# Why care about irrelevance?

1. The compiler can produce faster code
   - Erase arguments and their computation
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 may not require reasoning about irrelevant components
4. More programs type check
   - Sound to ignore irrelevant components when checking type equality

# 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. …
```

All proofs of equal properties are equal

# More programs type check

…when more terms are equal, *by definition*

type tells us that f is a constant function

```
example : ∀(f : ∀ (x : Bool) -> Bool)
            → (f True = f False)
example = λ f . Refl
```

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

Proof of equality comes directly from type checker

# Why care about irrelevance?

1. The compiler can produce faster code
   - Erasure / Run-time irrelevance
2. The type checker can run more quickly
   - Type checker optimizations
3. Verification is less work for programmers
   - Proof irrelevance, propositional irrelevance
4. More programs type check
   - Compile-time irrelevance, definitional proof irrelevance

# Type checkers for dependently-typed languages should identify irrelevant code

## But how?

## How should type checkers for dependently-typed languages identify irrelevant code?

1. **Erasure**
2. **Modes**
3. **Dependency**

# Core dependent type system

$$\boxed{\Gamma \vdash a : A}$$

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Pi}\ \frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi\, x : A.B : \star}$$

$$\textsc{Abs}\ \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash \Pi\, x : A.B : \star}{\Gamma \vdash \lambda x : A.\, b : \Pi\, x : A.B}$$

$$\textsc{App}\ \frac{\Gamma \vdash b : \Pi\, x : A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : B[a/x]}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : \star \qquad \vdash A \equiv B}{\Gamma \vdash a : B}$$

# Erasure

## You can't use something that is not there

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

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

$$\textsc{E-Abs}\ \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash \forall x : A.\, B : \star \qquad x \notin \mathsf{fv}|b|}{\Gamma \vdash \lambda x :^{\mathbf{I}} A.\, b : \boxed{\forall x : A.\, B}}$$

$$\textsc{E-App}\ \frac{\Gamma \vdash b : \forall x : A.\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a^{\mathbf{I}} : B[a/x]}$$

# ICC: Implicit Calculus of Constructions

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

$$\text{E-ABS}\;\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \forall x{:}A.\,B : \star \quad x \notin \mathsf{fv}|b|}{\Gamma \vdash \lambda x :^{\mathbf{I}} A.\, b : \forall x{:}A.\,B}$$

$$\text{E-APP}\;\frac{\Gamma \vdash b : \forall x{:}A.\,B \quad \Gamma \vdash a : A}{\Gamma \vdash b\ a^{\mathbf{I}} : B[a/x]}$$

# ICC: Implicit Calculus of Constructions

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

$$\textsc{E-Abs}\quad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash \forall x{:}A.\, B : \star \qquad x \notin \mathsf{fv}|b|}{\Gamma \vdash \lambda x :^{\mathbf{I}} A.\, b : \forall x{:}A.\, B}$$

$$\textsc{E-App}\quad \frac{\Gamma \vdash b : \forall x{:}A.\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a^{\mathbf{I}} : B[a/x]}$$

# Erasure during conversion

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

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

# Erasure: Implicit Calculus of Constructions

- Benefits
  - Simple!
  - Orthogonal: new features independent from the rest of the system
  - Directly connects to erasure in compilation
- Drawbacks
  - Direct implementation inefficient
  - Can't add irrelevant projections

# Filter is lazy in Haskell

```
filter : ∀(A:Type) → (A → Bool) → (List A) → (List A)
filter = λ A f v.
    case v of
      Nil ⇒ Nil
      Cons x xs ⇒
        let r = filter A f xs in
        if f x
          then (Cons x r)
          else r
```

Computed only when needed

```
Example:
   take 3 (filter isEven nats) → [0;2;4]
```

# Pattern matching : too strict!

```
vfilter : ∀(A:Type) → ∀(n:Nat)
        → (A → Bool) → (Vec n A) → ∃(m:Nat) × (Vec m A)
vfilter = λ A n f v.
    case v of
      Nil ⇒ (0, Nil)
      Cons m x xs ⇒
        case vfilter A m f xs of
          (m', r) ⇒ if f x
                    then (Succ m', Cons m' x r)
                    else (m', r)
```

# Irrelevant projections preserve laziness

```
vfilter : ∀(A:Type) → ∀(n:Nat)
        → (A → Bool) → (Vec n A) → ∃(m:Nat) × (Vec m A)
vfilter = λ A n f v.
    case v of
      Nil  ⇒ (
      Cons m
         let
         if f
            t
            e
```

Irrelevant projections:
UNSOUND addition to an
erasure-based calculus
| Vec p.1 A | = | Vec  q.1 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

# Modal types and modes

- Modal type marks irrelevant code: $\square A$
- Type system controlled by modes: $m ::= \mathbf{R} \mid \mathbf{I}$
  - Variable annotated in $\Gamma$, only $\mathbf{R}$ tagged usable

    $$\Gamma ::= \varepsilon \mid \Gamma,\ x :^{m} A$$

  - Resurrection ($\Gamma^{m}$): replace all $m$ tags with $\mathrm{R}$
  - Uniformity in abstractions:
    $\Pi x{:}^{m}A.B$ unifies $\Pi x{:}A.B$ and $\forall x{:}A.B$

# Modal types for irrelevance

Only relevant variables can be used

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

$$\textsc{M-Box}\quad \frac{\Gamma^{\mathbf{I}} \vdash a : A}{\Gamma \vdash \mathbf{box}\, a : \Box A}$$

$$\textsc{M-LetBox}\quad \frac{\Gamma \vdash a : \Box A \qquad \Gamma, x :^{\mathbf{I}} A \vdash b : B \qquad \Gamma \vdash B : \star}{\Gamma \vdash \mathbf{unbox}\, x = a\, \mathbf{in}\, b : B}$$

Modal types mark irrelevant subterms.
*Resurrection* means that any variable can be used inside a box.

The contents of the box are accessible only in other boxes.

# Modes annotate functions

Only relevant variables can be used

Π-bound variables always relevant in the type

Types checked with "resurrected" context

M-Var

$$\frac{x :^{\mathbf{R}} A \in \Gamma}{\Gamma \vdash x : A}$$

M-Pi

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x :^{\mathbf{R}} A \vdash B : \star}{\Gamma \vdash \Pi x :^{m} A.B : \star}$$

M-Abs

$$\frac{\Gamma^{\mathbf{I}} \vdash \Pi x :^{m} A.B : \star \qquad \Gamma, x :^{m} A \vdash b : B}{\Gamma \vdash \lambda x :^{m} A.\, a : \Pi x :^{m} A.B}$$

Mode on Π-type determines mode in the context

M-App

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

M-Conv

$$\frac{\Gamma \vdash a : A \qquad \vdash A \equiv B}{\Gamma \vdash a : B}$$

Irrelevant arguments checked with resurrected context

Conversion ignores irrelevant arguments

# Compile-time irrelevance

- Usual rules for beta-equivalence, plus
  - compare arguments marked $\mathbf{R}$
  - ignore arguments marked $\mathbf{I}$ or inside a box

$$\textsc{Eq-Rel}\quad \frac{\vdash b_1 \equiv b_2 \qquad \boxed{\vdash a_1 \equiv a_2}}{\vdash b_1\ {a_1}^{\mathbf{R}} \equiv b_2\ {a_2}^{\mathbf{R}}}$$

$$\textsc{Eq-Irr}\quad \frac{\vdash b_1 \equiv b_2}{\vdash b_1\ {a_1}^{\mathbf{I}} \equiv b_2\ {a_2}^{\mathbf{I}}}$$

# Modes for irrelevance

- Benefits
  - Modes identify patterns in the semantics: don't need two different functions
  - More direct implementation: mark variables when introduced in the context, mark the context for resurrection
- Drawbacks
  - Still no irrelevant projections
  - Formation rule for Π-types looks a bit strange

- Conjecture: equivalent to ICC*

# 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**

# Dependency tracking

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

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

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

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

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

# Typing rules with dependency levels $\boxed{\Gamma \vdash a :^{\ell} A}$

Variable usage restricted by observer level

D-VAR

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

D-PI

Vars have same level in terms and types

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

D-ABS

Terms do not observe types, level unimportant

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

Π-types record the dependency levels of their arguments

D-APP

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

Application requires compatible dependency levels

# DCOI: irrelevant projections

```
vfilter : (A:ᴵ Type) → (n:ᴵ Nat)
        → (A → Bool) → (Vec n A) → (m:ᴵ Nat) × (Vec m A)
vfilter = λ A n f v.
    case v of
      Nil ⇒ (0ᴵ, Nil)
      Cons mᴵ x xs ⇒
        let p = vfilter Aᴵ mᴵ f xs in
        if f x
          then ((Succ p.1)ᴵ, Cons p.1ᴵ x p.2)
          else p
```

Type is checked with I-observer

Definition checks with R observer, but contains I-marked subterms

First projection allowed in I-marked subterms only

# Indistinguishability: indexed definitional equality

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

Observer cannot distinguish between terms

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

$$\textsc{Eq-D-App-Dist}\quad \frac{\vdash b_0 \equiv^{\ell} b_1 \qquad \vdash a_0 \equiv^{\ell} a_1 \qquad \ell_0 \leq \ell}{\vdash b_0\ {a_0}^{\ell_0} \equiv^{\ell} b_1\ {a_1}^{\ell_0}}$$

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

$$\textsc{Eq-D-App-Indist}\quad \frac{\vdash b_0 \equiv^{\ell} b_1 \qquad \ell_0 \not\leq \ell}{\vdash b_0\ {a_0}^{\ell_0} \equiv^{\ell} b_1\ {a_1}^{\ell_0}}$$

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

$$\frac{\textsc{D-Conv} \quad \Gamma \vdash a :^{\ell} A \qquad \Gamma \vdash B :^{\ell_0} \star \qquad \vdash A \equiv^{\ell_0} B}{\Gamma \vdash a :^{\ell} B}$$

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

# DCOI: Dependent Calculus of Indistinguishability

- Liu, Chan, Shi and Weirich. *Internalizing Indistinguishability with Dependent Types.* POPL 2024
  - PTS version
  - Key results: Syntactic type soundness, noninterference

- Liu, Chan and Weirich. *Work in progress*
  - Predicative universe hierarchy
  - Observer-indexed propositional equality, J-eliminator
  - Key results: Consistency, normalization and decidable equality

- All results mechanized using Coq/Rocq proof assistant

# DCOI: Dependent Calculus of Indistinguishability

- Benefits
  - Irrelevant projection is sound!
  - General mechanism for dependency: applications besides irrelevance
  - Can reason about *indistinguishability* as a proposition
- Drawbacks (Future work)
  - Unknown compatibility with type-directed equality
  - Unknown compatibility with inductive datatypes
  - Unknown language ergonomics
    - Dependency level inference?
    - Dependency level quantification?

# Related work on Irrelevance

- Erasure-based
  - Miquel 2001,  Barras & Bernardo 2008
- Mode-based
  - Pfenning 2001, Mishra-Linger & Sheard 2008, Abel & Scherer 2012
- Dependency tracking
  - Type theory in color: Bernardy & Moulin 2013
  - Two level type theories: Kovács 2022, Annenkov et al. 2023
- sProp (Definitional proof irrelevance)
  - Gilbert et al. 2019
- Quantitative Type Theory
  - McBride 2016, Atkey 2018, Abel & Bernardy 2020,
    Choudhury et al. 2021, Moon et al. 2021

# Conclusions

- In dependent type systems, identifying irrelevant computations is important for *efficiency* and *expressivity*
- Type systems can track more than "types", they can also tell us what happens during computation
- Dependency analysis is a powerful hammer in type system design