Tracking how dependently-typed functions use their arguments

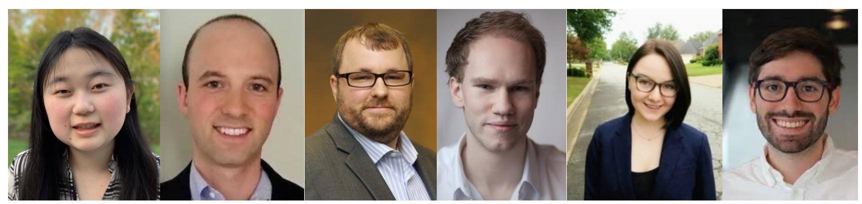
Stephanie Weirich

University of Pennsylvania
Philadelphia, USA

Collaborators

Yiyun Liu
Jonathan Chan
Pritam Choudhury

Jessica Shi
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 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 : Π(m:Nat) → A → (Vec m A) → Vec (Succ m) A

map : ∀(A B : Type) → Π(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

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 _ _ n 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 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
   – Erase arguments and their computation

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 : ∀(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

Type checkers for dependently-typed languages should identify irrelevant code

But how?

How should type checkers for dependently-typed languages identify and take advantage of irrelevant code?

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

Core dependent type system

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

Var

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

Pi

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

Abs

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

App

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

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

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

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

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

ICC: Implicit Calculus of Constructions

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

E-Abs

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

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
- 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

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

Erasure: Implicit Calculus of Constructions

- Benefits
  - Simple!
  - Directly connects to erasure in compilation
  - Orthogonal: features independent from the rest of the system

- Drawbacks
  - No irrelevant projections

Irrelevant projections

Length is not statically known
and irrelevant

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

UNSOUND addition to
compile-time erasure
| Vec p.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: $\Box A$
- 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 (Γ^{m}): replaces all m tags with R
  - Mode-annotated quantification:
    $\Pi x{:}^{m} A.B$ unifies $\Pi x{:}A.B$ and $\forall x{:}A.B$

Modal types for irrelevance

Only relevant variables can be used

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

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

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

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.

Mode-annotated functions

Only relevant variables can be used

M-Var

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

Π-bound variables always relevant in the type

M-Pi

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

Types checked with "resurrected" context

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

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

Irrelevant arguments checked with resurrected context

M-Conv

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

Conversion ignores irrelevant arguments

Compile-time irrelevance

- Usual rules, 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 function types
  - Easy 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*

Alternative rule for Π-types

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

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-Pi-Alt

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

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

Dependency tracking

- 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 $\leq L$

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

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, so 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

Type is checked with I-observer

```
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
```

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 at level ℓ 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

- **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 proof assistant

DCOI: Proof-specific features

False-elimination

D-Absurd

$$\frac{\Gamma \vdash a :^{m} \bot \qquad \Gamma \vdash A :^{\ell_1} \star}{\Gamma \vdash \mathbf{absurd}\ a :^{\ell} A}$$

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

Propositional indistinguishability

D-Eq

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

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

Level of proof must be less than current level

D-J

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

Equality between terms at level m witnessed via substitution at level m

Equality between proofs at level ℓ_0 witnessed via substitution at level ℓ_0

DCOI: Dependent Calculus of Indistinguishability

- Benefits
  - Irrelevant projection is sound!
  - Irrelevant absurdity is sound!
  - Can reason about *indistinguishability* as a proposition
- Future work
  - Compatibility with type-directed equality (relational model)
  - Language ergonomics
    - Dependency level inference?
    - Dependency level quantification?
  - Applications besides irrelevance?

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
- Proof irrelevance: Prop/sProp (Rcoq), Prop (Agda)
  - Hofmann& Streicher 1988, Gilbert et al. 2019
- Quantitative Type Theory (Run-time irrelevance / erasure)
  - McBride 2016, Atkey 2018, Abel & Bernardy 2020,
    Choudhury et al. 2021, Moon et al. 2021, Abel et al. 2023

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