Tracking how dependently-typed functions use their arguments

Stephanie Weirich
University of Pennsylvania
Philadelphia, USA
Collaborators

Yiyun Liu
Jonathan Chan
Jessica Shi
Pritam Choudhury
Richard Eisenberg
Harley Eades III
Antoine Voizard
Pedro Henrique Avezedo de Amorim
Anastasiya Kravchuk-Kirilyuk
Let's talk about constant functions
id : \forall (A : \text{Type}) \to A \to A

id = \lambda A \ x. \ x
id = \lambda \_ \_ \_ x. x

Erasure semantics for type polymorphism
Erasure semantics for polymorphism

```haskell
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

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

definition EvenNat = { n : Nat | isEven n }

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

\[\text{plus} = \lambda \text{en enm. case en, em of}
(n, _), (m, _) \Rightarrow (n + m, _)\]
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)

All proofs of equal properties are equal

-- 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

proof of equality comes directly from type checker

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

\[ \Gamma \vdash a : A \]

\[ \begin{align*}
\text{VAR} & \\
& x : A \in \Gamma \\
& \frac{}{\Gamma \vdash x : A}
\end{align*} \]

\[ \begin{align*}
\text{Pi} & \\
& \frac{\Gamma \vdash A : \star \quad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi x : A. B : \star}
\end{align*} \]

\[ \begin{align*}
\text{Abs} & \\
& \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \Pi x : A. B : \star}{\Gamma \vdash \lambda x : A. b : \Pi x : A. B}
\end{align*} \]

\[ \begin{align*}
\text{App} & \\
& \frac{\Gamma \vdash b : \Pi x : A. B \quad \Gamma \vdash a : A}{\Gamma \vdash b \ a : B[a/x]}
\end{align*} \]

\[ \begin{align*}
\text{Conv} & \\
& \frac{\Gamma \vdash a : A \quad \vdash A \equiv B}{\Gamma \vdash a : B}
\end{align*} \]
You can't use something that is not there
ICC: Implicit Calculus of Constructions

- Extend core language with irrelevant (implicit) abstractions

\[
\text{E-Abs} \quad \begin{array}{c}
\Gamma, x : A \vdash b : B \\
\Gamma \vdash \forall x : A. B : \star \\
x \notin \text{fv} |b| \\
\hline
\Gamma \vdash \lambda x : I A. b : \forall x : A. B
\end{array}
\]

\[
\text{E-App} \quad \begin{array}{c}
\Gamma \vdash b : \forall x : A. B \\
\Gamma \vdash a : A \\
\hline
\Gamma \vdash b a^I : B[a/x]
\end{array}
\]
ICC: Implicit Calculus of Constructions

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

\[
\begin{align*}
\text{E-Abs} & \quad \Gamma, x : A \vdash b : B \\
& \quad \Gamma \vdash \forall x : A. B : \star \\
& \quad x \not\in \text{fv}\{b\} \\
\Gamma & \vdash \lambda x : ^I A. b : \forall x : A. B
\end{align*}
\]

\[
\begin{align*}
\text{E-App} & \quad \Gamma \vdash b : \forall x : A. B \\
& \quad \Gamma \vdash a : A \\
& \quad \Gamma \vdash b \ a^I : B[a/x]
\end{align*}
\]
ICC: Implicit Calculus of Constructions

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

\[
\text{E-Abs} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash \forall x : A. B : \star}{\Gamma \vdash \lambda x : \mathbf{I} A. b : \forall x : A. B \quad x \notin \text{fv}\,|b|}
\]

\[
\text{E-App} \quad \frac{\Gamma \vdash b : \forall x : A. B \quad \Gamma \vdash a : A}{\Gamma \vdash b\,a^\mathbf{I} : B[a/x]}
\]
Erasure during conversion

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

\[
\Gamma \vdash a : A \\
\Gamma \vdash B : \star \\
\vdash |A| \equiv |B| \\
\hline
\Gamma \vdash a : B
\]
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

\[
\text{filter} : \forall (A: \text{Type}) \rightarrow (A \rightarrow \text{Bool}) \rightarrow (\text{List } A) \rightarrow (\text{List } A)
\]

\[
\text{filter} = \lambda A \ f \ v. \\
\text{case } v \text{ of} \\
\text{Nil } \Rightarrow \text{Nil} \\
\text{Cons } x \ xs \Rightarrow \\
\text{let } r = \text{filter } A \ f \ xs \text{ in} \\
\text{if } f \ x \text{ then } (\text{Cons } x \ r) \text{ else } r
\]

Example:

\[
\text{take 3 (filter isEven nats)} \rightarrow [0;2;4]
\]
Pattern matching : too strict!

\[
\text{vfilter} : \forall (A : \text{Type}) \to \forall (n : \text{Nat}) \\
\to (A \to \text{Bool}) \to (\text{Vec} \ n \ A) \to \exists (m : \text{Nat}) \times (\text{Vec} \ m \ A)
\]

\[
\text{vfilter} = \lambda A \ n \ f \ v. \\
\text{case } v \text{ of } \\
\text{Nil } \Rightarrow (0, \text{Nil}) \\
\text{Cons } m \ x \ xs \Rightarrow \\
\text{case } \text{vfilter} \ A \ m \ f \ xs \text{ of } \\
(m', r) \Rightarrow \text{if } f \ x \\
\text{then } (\text{Succ } m', \text{Cons } m' \ x \ r) \\
\text{else } (m', r)
\]
Irrelevant projections preserve laziness

\[
\text{vfilter} : \forall (A : \text{Type}) \rightarrow \forall (n : \text{Nat}) \\
\rightarrow (A \rightarrow \text{Bool}) \rightarrow (\text{Vec} \ n \ A) \rightarrow \exists (m : \text{Nat}) \times (\text{Vec} \ m \ A)
\]

\[
vfilter = \lambda A \ n \ f \ v. \\
\text{case } v \text{ of } \\
\text{Nil } \Rightarrow (0, \text{Nil}) \\
\text{Cons } m \ x \ xs \Rightarrow \text{let } \\
\text{let } p = \text{vfilter} \ A \ m \ f \ xs \text{ in } \\
\text{if } f \ x \ \text{then } \\
(\text{Succ} \ p.1, \text{Cons} \ p.1 \ x \ p.2) \\
\text{else } p
\]

Irrelevant projections:
UN SOUND addition to an erasure-based calculus
| \text{Vec} \ p.1 \ A | = | \text{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: \( A \)
- Type system controlled by modes: \( m ::= R | I \)
  - Variable annotated in \( \Gamma \), only \( R \) tagged usable

\[
\Gamma ::= \varepsilon \mid \Gamma, \ x :^mA
\]

- Resurrection \((\Gamma^m)\): replace all \( m \) tags with \( R \)

- Uniformity in abstractions:

\[
\Pi x.^mA. B \quad \text{unifies} \quad \Pi x:A. B \quad \text{and} \quad \forall x:A. B
\]
Modal types for irrelevance

Only relevant variables can be used

\[
\frac{\text{M-VAR}}{\text{M-Box}} \quad \frac{x : \mathbb{R} A \in \Gamma}{\Gamma \vdash x : A}
\]

\[
\frac{\Gamma^I \vdash a : A}{\Gamma \vdash \text{box } a : \square A}
\]

\[
\frac{\Gamma \vdash a : \square A \quad \Gamma, x : \square A \vdash b : B}{\Gamma \vdash \text{unbox } x = a \text{ in } b : B}
\]

M-LETBox

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

**M-VAR**

\[
\begin{align*}
\Gamma \vdash x : A \\
\& x : R \ A \in \Gamma
\end{align*}
\]

**M-Π**

\[
\begin{align*}
\Gamma \vdash A : * \\
\Gamma, x : R \ A \vdash B : * \\
\Gamma \vdash \Pi x : m A.B : *
\end{align*}
\]

**M-ABS**

\[
\begin{align*}
\Gamma^I \vdash \Pi x : m A.B : * \\
\Gamma, x : m A \vdash b : B \\
\Gamma \vdash \lambda x : m A. a : \Pi x : m A.B
\end{align*}
\]

π-bound variables always relevant in the type

Types checked with "resurrected" context

Irrelevant arguments checked with resurrected context

**M-APP**

\[
\begin{align*}
\Gamma \vdash b : \Pi x : m A.B \\
\Gamma^m \vdash a : A \\
\Gamma \vdash b a^m : B[a/x]
\end{align*}
\]

**M-CONV**

\[
\begin{align*}
\Gamma \vdash a : A \\
\vdash A \equiv B \\
\Gamma \vdash a : B
\end{align*}
\]

Mode on Π-type determines mode in the 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

\[
\text{EQ-REL} \quad \begin{array}{c}
\vdash b_1 \equiv b_2 \\
\vdash a_1 \equiv a_2 \\
\hline
\vdash b_1 a_1^\mathbf{R} \equiv b_2 a_2^\mathbf{R}
\end{array}
\]

\[
\text{EQ-IRR} \quad \begin{array}{c}
\vdash b_1 \equiv b_2 \\
\vdash b_1 a_1^\mathbf{I} \equiv b_2 a_2^\mathbf{I}
\end{array}
\]
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 ($\text{Low} < \text{Med} < \text{High}$)
    Staged computation ($0 < 1 < 2...$)

- 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 $\leq L$
Typing rules with dependency levels

\[ \Gamma \vdash a :^\ell A \]

**D-VAR**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-VAR</td>
<td>Variable usage restricted by observer level</td>
</tr>
<tr>
<td>( x :^m A \in \Gamma )</td>
<td>( m \leq \ell )</td>
</tr>
<tr>
<td>( \Gamma \vdash x :^\ell A )</td>
<td></td>
</tr>
</tbody>
</table>

**D-Pi**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Pi</td>
<td>Vars have same level in terms and types</td>
</tr>
<tr>
<td>( \Gamma \vdash A :^\ell \ast )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma, x :^m A \vdash B :^\ell \ast )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \Pi x :^m A.B :^\ell \ast )</td>
<td></td>
</tr>
</tbody>
</table>

**D-Abs**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Abs</td>
<td>Terms do not observe types, level unimportant</td>
</tr>
<tr>
<td>( \Gamma, x :^m A \vdash b :^\ell B )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \Pi x :^m A.B :^{\ell_1} \ast )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \lambda x :^m A. b :^\ell \Pi x :^m A.B )</td>
<td></td>
</tr>
</tbody>
</table>

**D-App**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-App</td>
<td>Application requires compatible dependency levels</td>
</tr>
<tr>
<td>( \Gamma \vdash b :^\ell \Pi x :^m A.B )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash a :^m A )</td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash b \ a :^m \ast B[a/x] )</td>
<td></td>
</tr>
</tbody>
</table>

\( \Pi \)-types record the dependency levels of their arguments

Terms do not observe types, level unimportant
vfilter : (A:I Type) \to (n:I Nat) \to (A \to Bool) \to (Vec n A) \to (m:I Nat) \times (Vec m A)

vfilter = \lambda A n f v. case v of
  Nil \Rightarrow (0, Nil)
  Cons m x xs \Rightarrow
    let p = vfilter A m f xs in
    if f x
    then ((Succ p.1), Cons p.1 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

\[ \frac{a_0 \equiv^\ell a_1 \quad \ell_0 \leq \ell}{\vdash b_0 a_0^{\ell_0} \equiv^\ell a_1 a_1^{\ell_0}} \]

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

\[ \frac{b_0 \equiv^\ell b_1 \quad \ell_0 \not\leq \ell}{\vdash b_0 a_0^{\ell_0} \equiv^\ell b_1 a_1^{\ell_0}} \]
Conversion can be used at any observer level

\[
\text{D-Conv}
\]

\[
\Gamma \vdash a : \ell A \\
\Gamma \vdash B : \ell_0 \star \\
\vdash A \equiv_{\ell_0} B
\]

\[
\Gamma \vdash a : \ell B
\]

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
• Mode-based
• 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