Layered Modal Type Theory

Where Meta-programming Meets Intensional Analysis

Jason Hu Brigitte Pientka

McGill University

at ESOP 2024

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)
  - Work above has no type-theoretic foundation

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)
  - Work above has no type-theoretic foundation
- Foundational studies with modality:

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)
  - Work above has no type-theoretic foundation
- Foundational studies with modality:
  - (Contextual) $\lambda^{\square}$ (Davies and Pfenning, 2001; Nanevski et al., 2008)

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)
  - Work above has no type-theoretic foundation
- Foundational studies with modality:
  - (Contextual) $\lambda^{\Box}$ (Davies and Pfenning, 2001; Nanevski et al., 2008)
  - Cocon (Pientka et al., 2019)

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)
  - Work above has no type-theoretic foundation
- Foundational studies with modality:
  - (Contextual) $\lambda^{\square}$ (Davies and Pfenning, 2001; Nanevski et al., 2008)
  - Cocon (Pientka et al., 2019)
  - 2LTT (Kovács, 2022)

Need for Foundations of Meta-programming in Type Theory

- Meta-programming is a common practice in type theory:
  - Reflection in Agda (van der Walt and Swierstra, 2012), Lean (Ebner et al., 2017)
  - Instrumentation in Coq: Mtac (Ziliani et al., 2013), Mtac2 (Kaiser et al., 2018)
  - MetaCoq (Sozeau et al., 2020)
  - Work above has no type-theoretic foundation
- Foundational studies with modality:
  - (Contextual) $\lambda^{\Box}$ (Davies and Pfenning, 2001; Nanevski et al., 2008)
  - Cocon (Pientka et al., 2019)
  - 2LTT (Kovács, 2022)
  - Work above does not have all features we want

What We Want (WWW)

- quotation: internally represent syntax;

What We Want (WWW)

- quotation: internally represent syntax;
- intensional analysis: *covering* pattern matching on code;

What We Want (WWW)

- quotation: internally represent syntax;
- intensional analysis: *covering* pattern matching on code;
- running: evaluate a code of type A and obtain an A;

What We Want (WWW)

- quotation: internally represent syntax;
- intensional analysis: *covering* pattern matching on code;
- running: evaluate a code of type A and obtain an A;
- a type theory: normalization algorithm and proof

A Quick Comparison

System	quotation	intensional analysis	running	normalization
reflection, instrumentation	✓	✓	✓	
(Contextual) $\lambda^{\Box}$	✓		✓	✓
Cocon	✓	✓		✓
2LTT	✓			✓

A Quick Comparison

System	quotation	intensional analysis	running	normalization
reflection, instrumentation	✓	✓	✓	
(Contextual) $\lambda^{\Box}$	✓		✓	✓
Cocon	✓	✓		✓
2LTT	✓			✓
?	✓	✓	✓	✓

A Quick Comparison

System	quotation	intensional analysis	running	normalization
reflection, instrumentation	✓	✓	✓	
(Contextual) $\lambda^{\Box}$	✓		✓	✓
Cocon	✓	✓		✓
2LTT	✓			✓
Our Work	✓	✓	✓	✓

We focus on simple types as a stepping stone

Outline

- An example for meta-programming and pattern matching on code

Outline

- An example for meta-programming and pattern matching on code
- Typing judgment for 2-layered modal simple type theory

Outline

- An example for meta-programming and pattern matching on code
- Typing judgment for 2-layered modal simple type theory
- The static code lemma and how it enables pattern matching on code

Outline

- An example for meta-programming and pattern matching on code
- Typing judgment for 2-layered modal simple type theory
- The static code lemma and how it enables pattern matching on code
- The lifting lemma and how it enables code running

Outline

- An example for meta-programming and pattern matching on code
- Typing judgment for 2-layered modal simple type theory
- The static code lemma and how it enables pattern matching on code
- The lifting lemma and how it enables code running
- A normalization algorithm and its completeness and soundness proof

Multiplication: An Example for Meta-programming

-
```
mult : Nat → □ (x : Nat ⊢ Nat)
mult zero     = box (x. 0)
mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
```

Multiplication: An Example for Meta-programming


- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero     = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```
  ```

Multiplication: An Example for Meta-programming

- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero      = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```
  ```

Multiplication: An Example for Meta-programming


- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero     = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```
  ```

Multiplication: An Example for Meta-programming


- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero     = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```

- However, these 0's are redundant:

  ```
  mult 1 ≈ box (x. 0 + x)        ≉ box (x. x)
  mult 2 ≈ box (x. (0 + x) + x)  ≉ box (x. x + x)
  ```
  ```

Multiplication: An Example for Meta-programming


- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero     = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```
- However, these 0's are redundant:

  ```
  mult 1 ≈ box (x. 0 + x)       ≉ box (x. x)
  mult 2 ≈ box (x. (0 + x) + x) ≉ box (x. x + x)
  ```
  ```

Multiplication: An Example for Meta-programming

- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero     = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```

- However, these 0's are redundant:

  ```
  mult 1 ≈ box (x. 0 + x)           ≉ box (x. x)
  mult 2 ≈ box (x. (0 + x) + x)     ≉ box (x. x + x)
  ```

- Use letbox to run the generated function:

  ```
  letbox u ← mult 2 in u[5/x] ≈ 10
  letbox u ← mult 2 in λ y. u[y/x] ≈ λ y. (0 + y) + y ≈ λ y. y + y
  ```
  ```

Multiplication: An Example for Meta-programming


- ```
  mult : Nat → □ (x : Nat ⊢ Nat)
  mult zero     = box (x. 0)
  mult (succ n) = letbox u ← mult n in box (x. u[x/x] + x)
  ```

- However, these `0`'s are redundant:

  ```
  mult 1 ≈ box (x. 0 + x)         ≉ box (x. x)
  mult 2 ≈ box (x. (0 + x) + x)   ≉ box (x. x + x)
  ```

- Use `letbox` to run the generated function:

  ```
  letbox u ← mult 2 in u[5/x] ≈ 10
  letbox u ← mult 2 in λ y. u[y/x] ≈ λ y. (0 + y) + y ≈ λ y. y + y
  ```
  ```

An Arithmetic Simplifier via Pattern Matching


```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp y = match y with
         | 0 + ?u ⇒ box (x. u)
         | ?u + ?u' ⇒
           letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
         | _ ⇒ y
```

An Arithmetic Simplifier via Pattern Matching


- ```
  simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
  simp y = match y with
         | 0 + ?u ⇒ box (x. u)
         | ?u + ?u' ⇒
             letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
         | _ ⇒ y
  ```
  ```

An Arithmetic Simplifier via Pattern Matching


- ```
  simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
  simp y = match y with
          | 0 + ?u ⇒ box (x. u)
          | ?u + ?u' ⇒
             letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
          | _ ⇒ y
  ```
  ```

An Arithmetic Simplifier via Pattern Matching

-
```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp y = match y with
         | 0 + ?u ⇒ box (x. u)
         | ?u + ?u' ⇒
             letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
         | _ ⇒ y
```

An Arithmetic Simplifier via Pattern Matching


- ```
  simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
  simp y = match y with
           | 0 + ?u ⇒ box (x. u)
           | ?u + ?u' ⇒
               letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
           | _ ⇒ y
  ```
  ```

An Arithmetic Simplifier via Pattern Matching


- ```
  simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
  simp y = match y with
           | 0 + ?u ⇒ box (x. u)
           | ?u + ?u' ⇒
               letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
           | _ ⇒ y
  ```
  ```

An Arithmetic Simplifier via Pattern Matching


- ```
  simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
  simp y = match y with
           | 0 + ?u ⇒ box (x. u)
           | ?u + ?u' ⇒
              letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
           | _ ⇒ y
  ```
- Use `simp` to simplify `0`'s away:

```
mult' : Nat → □ (Nat → Nat)
mult' n = letbox u ← simp (mult n) in box (λ x. u[x/x])
```
```

An Arithmetic Simplifier via Pattern Matching


- ```
  simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
  simp y = match y with
            | 0 + ?u ⇒ box (x. u)
            | ?u + ?u' ⇒
                letbox u1 ← simp (box (x. u)) in box (x. u1 + u')
            | _ ⇒ y
  ```
- Use `simp` to simplify `0`'s away:

  ```
  mult' : Nat → □ (Nat → Nat)
  mult' n = letbox u ← simp (mult n) in box (λ x. u[x/x])
  ```
- Finally we have the simplest forms:

  ```
  mult' 1 ≈ box (λ x. x)
  mult' 2 ≈ box (λ x. x + x)
  ```
  ```

Secret Ingredient: Layered Typing Judgment

In $\lambda^{\square}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$

Secret Ingredient: Layered Typing Judgment

In $\lambda^{\Box}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$

In our work, $\Psi; \Gamma \vdash_i t : T$: term t is well-typed in contexts Ψ and Γ at layer i where $i \in [0, 1]$

Secret Ingredient: Layered Typing Judgment

In $\lambda^{\Box}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$

In our work, $\Psi; \Gamma \vdash_i t : T$: term t is well-typed in contexts Ψ and Γ at layer i where $i \in [0, 1]$

- Γ: a regular context; bindings are $x : T$

Secret Ingredient: Layered Typing Judgment

In $\lambda^{\square}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$

In our work, $\Psi; \Gamma \vdash_i t : T$: term t is well-typed in contexts Ψ and Γ at layer i where $i \in [0, 1]$

- Γ: a regular context; bindings are $x : T$
- Ψ: a global(meta) context; bindings are $u : (\Delta \vdash T)$

Secret Ingredient: Layered Typing Judgment

In $\lambda^{\square}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$

In our work, $\Psi; \Gamma \vdash_i t : T$: term t is well-typed in contexts Ψ and Γ at layer i where $i \in [0, 1]$

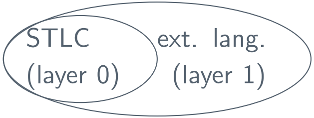
- Γ: a regular context; bindings are $x : T$
- Ψ: a global(meta) context; bindings are $u : (\Delta \vdash T)$
- $\Psi; \Gamma \vdash_0 t : T$ describes a term in STLC

Secret Ingredient: Layered Typing Judgment

In $\lambda^{\Box}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$

In our work, $\Psi; \Gamma \vdash_i t : T$: term t is well-typed in contexts Ψ and Γ at layer i where $i \in [0, 1]$

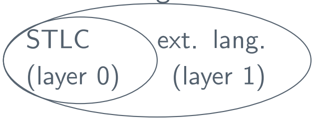
- Γ: a regular context; bindings are $x : T$
- Ψ: a global(meta) context; bindings are $u : (\Delta \vdash T)$
- $\Psi; \Gamma \vdash_0 t : T$ describes a term in STLC
- $\Psi; \Gamma \vdash_1 t : T$ extends STLC with ability to do meta-programming: quotation, intensional analysis and code running



Secret Ingredient: Layered Typing Judgment

In $\lambda^{\square}$ by Davies and Pfenning (2001), typing judgment is $\Psi; \Gamma \vdash t : T$
In our work, $\Psi; \Gamma \vdash_i t : T$: term t is well-typed in contexts Ψ and Γ at layer i where $i \in [0, 1]$

- Γ: a regular context; bindings are $x : T$
- Ψ: a global(meta) context; bindings are $u : (\Delta \vdash T)$
- $\Psi; \Gamma \vdash_0 t : T$ describes a term in STLC
- $\Psi; \Gamma \vdash_1 t : T$ extends STLC with ability to do meta-programming: quotation, intensional analysis and code running
- important lemmas: static code and lifting



Layering Maintains Static Status of Code

$$\frac{\Delta\ \mathtt{wf}^{0} \qquad T\ \mathtt{wf}^{0}}{\square(\Delta \vdash T)\ \mathtt{wf}^{1}}$$

Layering Maintains Static Status of Code

$$\frac{\Delta\ \mathtt{wf}^0 \qquad T\ \mathtt{wf}^0}{\square(\Delta \vdash T)\ \mathtt{wf}^1}$$

$$\frac{\Gamma\ \mathtt{wf}^1 \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t : \square(\Delta \vdash T)}$$

Layering Maintains Static Status of Code

$$\frac{\Delta\ \texttt{wf}^0 \qquad T\ \texttt{wf}^0}{\Box(\Delta \vdash T)\ \texttt{wf}^1}$$

$$\frac{\Gamma\ \texttt{wf}^1 \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \texttt{box}\ t : \Box(\Delta \vdash T)}$$

$$\frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T}$$

Layering Maintains Static Status of Code

$$\frac{\Delta\ \mathtt{wf}^0 \qquad T\ \mathtt{wf}^0}{\square(\Delta \vdash T)\ \mathtt{wf}^1} \qquad \frac{\Gamma\ \mathtt{wf}^1 \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T}$$

$$\frac{\Gamma\ \mathtt{wf}^1 \qquad \Psi; \Delta \vdash_0 t \approx t' : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t \approx \mathtt{box}\ t' : \square(\Delta \vdash T)}$$

Layering Maintains Static Status of Code

$$\frac{\Delta\ \texttt{wf}^0 \qquad T\ \texttt{wf}^0}{\square(\Delta \vdash T)\ \texttt{wf}^1} \qquad \frac{\Gamma\ \texttt{wf}^1 \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \texttt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T}$$

$$\frac{\Gamma\ \texttt{wf}^1 \qquad \Psi; \Delta \vdash_0 t \approx t' : T}{\Psi; \Gamma \vdash_1 \texttt{box}\ t \approx \texttt{box}\ t' : \square(\Delta \vdash T)}$$

- Code does not compute:

Lemma (Static Code)

If $\Psi; \Gamma \vdash_0 t \approx s : T$, then $t = s$.

Layering Maintains Static Status of Code

$$\frac{\Delta \ \texttt{wf}^0 \qquad T \ \texttt{wf}^0}{\Box(\Delta \vdash T) \ \texttt{wf}^1} \qquad \frac{\Gamma \ \texttt{wf}^1 \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \texttt{box} \ t : \Box(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T}$$

$$\frac{\Gamma \ \texttt{wf}^1 \qquad \Psi; \Delta \vdash_0 t \approx t' : T}{\Psi; \Gamma \vdash_1 \texttt{box} \ t \approx \texttt{box} \ t' : \Box(\Delta \vdash T)} \qquad \boxed{\frac{\Psi; \Gamma, x : S \vdash_1 t : T \qquad \Psi; \Gamma \vdash_1 s : S}{\Psi; \Gamma \vdash_1 (\lambda x.t) \ s \approx t[s/x] : T}}$$

- Code does not compute:

Lemma (Static Code)

If $\Psi; \Gamma \vdash_0 t \approx s : T$, *then* $t = s$.

Layering Maintains Static Status of Code

$$\frac{\Delta\ \mathtt{wf}^0 \qquad T\ \mathtt{wf}^0}{\square(\Delta \vdash T)\ \mathtt{wf}^1} \qquad \frac{\Gamma\ \mathtt{wf}^1 \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T}$$

$$\frac{\Gamma\ \mathtt{wf}^1 \qquad \Psi; \Delta \vdash_0 t \approx t' : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t \approx \mathtt{box}\ t' : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma, x : S \vdash_1 t : T \qquad \Psi; \Gamma \vdash_1 s : S}{\Psi; \Gamma \vdash_1 (\lambda x.t)\ s \approx t[s/x] : T}$$

- Code does not compute:

Lemma (Static Code)

If $\Psi; \Gamma \vdash_0 t \approx s : T$, then $t = s$.

- All β and η rules only occur at layer 1 $\Longrightarrow$ static code lemma

Static Code Enables Pattern Matching

$$\frac{\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \text{match } s \text{ with } \overrightarrow{b} : T'}$$

Static Code Enables Pattern Matching

$$\frac{\Psi; \Gamma \vdash_1 s : \boxed{\Box(\Delta \vdash T)} \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \boxed{\Delta \vdash T} \Rightarrow T'}{\Psi; \Gamma \vdash_1 \text{match } s \text{ with } \overrightarrow{b} : T'}$$

Static Code Enables Pattern Matching

$$\frac{\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \text{match } s \text{ with } \overrightarrow{b} : T'}$$

One of branches:

$$\frac{\Psi, u : (\Delta, x : S \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \lambda x.?u \Rightarrow t : \Delta \vdash \boxed{S \longrightarrow T} \Rightarrow T'}$$

Static Code Enables Pattern Matching

$$\frac{\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \text{match } s \text{ with } \overrightarrow{b} : T'}$$

One of branches:

$$\frac{\Psi, u : (\Delta, x : S \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \lambda x.\boxed{?u} \Rightarrow t : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

Static Code Enables Pattern Matching

$$\frac{\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \text{match } s \text{ with } \overrightarrow{b} : T'}$$

One of branches:

$$\frac{\Psi, \boxed{u : (\Delta, x : S \vdash T)}; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \lambda x.?u \Rightarrow t : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

What We Want (WWW)

- ✓ quotation: internally represent syntax (`box`);
- ✓ intensional analysis: *covering* pattern matching on code;
- running: evaluate a code of type A and obtain an A;
- a type theory: normalization algorithm and proof

Layering Lifts Code to Programs

Lifting lemma: characterization of layering

Lemma (Lifting)

If $\Psi; \Gamma \vdash_0 t : T$, *then* $\Psi; \Gamma \vdash_1 t : T$.

Layering Lifts Code to Programs

Lifting lemma: characterization of layering

Lemma (Lifting)

If $\Psi; \Gamma \vdash_0 t : T$, *then* $\Psi; \Gamma \vdash_1 t : T$.

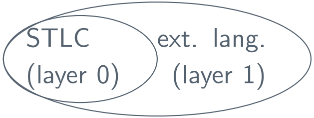
Layer 1 **subsumes** layer 0.

Lifting lemma: characterization of layering

Lemma (Lifting)

If $\Psi; \Gamma \vdash_0 t : T$, *then* $\Psi; \Gamma \vdash_1 t : T$.

Layer 1 **subsumes** layer 0.



Lifting Enables Running Code

`letbox` composes and runs $\Box(\Delta \vdash T)$:

$$\frac{\Psi; \Gamma \vdash_1 s : \boxed{\Box(\Delta \vdash T)} \qquad \Psi, u : (\boxed{\Delta \vdash T}); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t : T'}$$

Lifting Enables Running Code

`letbox` composes and runs $\Box(\Delta \vdash T)$:

$$\frac{\Psi; \Gamma \vdash_1 s : \Box(\Delta \vdash T) \qquad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t : T'}$$

Remark: computation is only suspended at layer 0, not lost!

```
  letbox u ← mult 2 in λ y. u[y/x]
≈ letbox u ← box (x. (0 + x) + x) in λ y. u[y/x]
≈ λ y. (0 + y) + y     -- lifting occurs
≈ λ y. y + y
```

`(0 + x) + x` is frozen in `box` but eventually computes.

Lifting Enables Running Code

`letbox` composes and runs $\Box(\Delta \vdash T)$:

$$\frac{\Psi; \Gamma \vdash_1 s : \Box(\Delta \vdash T) \qquad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t : T'}$$

Remark: computation is only suspended at layer 0, not lost!

```
  letbox u ← mult 2 in λ y. u[y/x]
≈ letbox u ← box (x. (0 + x) + x) in λ y. u[y/x]
≈ λ y. (0 + y) + y      -- lifting occurs
≈ λ y. y + y
```

`(0 + x) + x` is frozen in `box` but eventually computes.

Lifting Enables Running Code

`letbox` composes and runs $\Box(\Delta \vdash T)$:

$$\frac{\Psi; \Gamma \vdash_1 s : \Box(\Delta \vdash T) \qquad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t : T'}$$

Remark: computation is only suspended at layer 0, not lost!

```
  letbox u ← mult 2 in λ y. u[y/x]
≈ letbox u ← box (x. (0 + x) + x) in λ y. u[y/x]
≈ λ y. (0 + y) + y     -- lifting occurs
≈ λ y. y + y
```

`(0 + x) + x` is frozen in `box` but eventually computes.

What We Want (WWW)

- ✓ quotation: internally represent syntax (`box`);
- ✓ intensional analysis: *covering* pattern matching on code;
- ✓ running: evaluate a code of type A and obtain an A;
- a type theory: normalization algorithm and proof

Normalization by Evaluation

- A moderate extension of the standard presheaf model (Altenkirch et al., 1995)

Normalization by Evaluation

- A moderate extension of the standard presheaf model (Altenkirch et al., 1995)
- An algorithm is extracted from the presheaf model

Normalization by Evaluation

- A moderate extension of the standard presheaf model (Altenkirch et al., 1995)
- An algorithm is extracted from the presheaf model
- Complete and sound:

Normalization by Evaluation

- A moderate extension of the standard presheaf model (Altenkirch et al., 1995)
- An algorithm is extracted from the presheaf model
- Complete and sound:

Theorem (Completeness)

If $\Psi; \Gamma \vdash_1 t \approx t' : T$, *then* $nbe^{T}_{\Psi;\Gamma}(t) = nbe^{T}_{\Psi;\Gamma}(t')$.

Normalization by Evaluation

- A moderate extension of the standard presheaf model (Altenkirch et al., 1995)
- An algorithm is extracted from the presheaf model
- Complete and sound:

Theorem (Completeness)

If $\Psi; \Gamma \vdash_1 t \approx t' : T$, *then* $nbe^T_{\Psi;\Gamma}(t) = nbe^T_{\Psi;\Gamma}(t')$.

Theorem (Soundness)

If $\Psi; \Gamma \vdash_1 t : T$, *then* $\Psi; \Gamma \vdash_1 t \approx nbe^T_{\Psi;\Gamma}(t) : T$.

Normalization by Evaluation

- A moderate extension of the standard presheaf model (Altenkirch et al., 1995)
- An algorithm is extracted from the presheaf model
- Complete and sound:

Theorem (Completeness)

If $\Psi; \Gamma \vdash_1 t \approx t' : T$*, then* $nbe^{T}_{\Psi;\Gamma}(t) = nbe^{T}_{\Psi;\Gamma}(t')$*.*

Theorem (Soundness)

If $\Psi; \Gamma \vdash_1 t : T$*, then* $\Psi; \Gamma \vdash_1 t \approx nbe^{T}_{\Psi;\Gamma}(t) : T$*.*

- The algorithm is implemented in Agda

What We Have Achieved

- ✓ quotation: internally represent syntax (`box`);
- ✓ intensional analysis: *covering* pattern matching on code;
- ✓ running: evaluate a code of type A and obtain an A;
- ✓ a type theory: normalization algorithm and proof

Takeaways

- Layering is key to enable both code running and pattern matching on code in a coherent type theory

Takeaways

- Layering is key to enable both code running and pattern matching on code in a coherent type theory
- A complete and sound normalization algorithm based on a presheaf model

Takeaways

- Layering is key to enable both code running and pattern matching on code in a coherent type theory
- A complete and sound normalization algorithm based on a presheaf model
- Our paper gives three possible future directions; reach out if you are interested!
  - System F and MLTT
  - Extending operations based on rewrite rules
  - *n* layers

zhong.s.hu at mail.mcgill.ca

Bibliography

Altenkirch, T., Hofmann, M., and Streicher, T. (1995). Categorical reconstruction of a reduction free normalization proof. In Pitt, D. H., Rydeheard, D. E., and Johnstone, P. T., editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer.

Davies, R. and Pfenning, F. (2001). A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604.

Ebner, G., Ullrich, S., Roesch, J., Avigad, J., and de Moura, L. (2017). A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP):34:1–34:29.

Kaiser, J.-O., Ziliani, B., Krebbers, R., Régis-Gianas, Y., and Dreyer, D. (2018). Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages*, 2(ICFP):78:1–78:31.

Kovács, A. (2022). Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP):540–569.

Nanevski, A., Pfenning, F., and Pientka, B. (2008). Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):23:1–23:49.

Pientka, B., Thibodeau, D., Abel, A., Ferreira, F., and Zucchini, R. (2019). A type theory for defining logics and proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE.

Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., and Winterhalter, T. (2020). The metacoq project. *J. Autom. Reason.*, 64(5):947–999.

van der Walt, P. and Swierstra, W. (2012). Engineering proof by reflection in agda. In Hinze, R., editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer.

Ziliani, B., Dreyer, D., Krishnaswami, N. R., Nanevski, A., and Vafeiadis, V. (2013). Mtac: a monad for typed tactic programming in coq. In Morrisett, G. and Uustalu, T., editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 87–100. ACM.

Comparison of Expressive Power

Without pattern matching, layered modal type theory is strictly weaker than $\lambda^{\square}$.

$$\begin{aligned}\mathsf{lift} &: \mathtt{Nat} \to \square\mathtt{Nat}\\ \mathsf{lift}(\mathsf{zero}) &:= \mathtt{box}\ \mathsf{zero}\\ \mathsf{lift}(\mathsf{succ}\ x) &:= \mathtt{letbox}\ u \leftarrow \mathsf{lift}(x)\ \mathtt{in}\ \mathtt{box}\ (\mathsf{succ}\ u)\end{aligned}$$

supported by both systems; turn succ $(\cdots(\mathsf{succ\ zero}))$ into box $(\mathsf{succ}\ (\cdots(\mathsf{succ\ zero})))$

$$\begin{aligned}\mathsf{nest} &: \mathtt{Nat} \to \square\mathtt{Nat}\\ \mathsf{nest}(\mathsf{zero}) &:= \mathtt{box}\ \mathsf{zero}\\ \mathsf{nest}(\mathsf{succ}\ x) &:= \mathtt{letbox}\ u \leftarrow \mathsf{nest}(x)\ \mathtt{in}\ \mathtt{box}\ (\mathtt{letbox}\ u' \leftarrow \mathsf{nest}(u)\ \mathtt{in}\ u')\end{aligned}$$

supported only by $\lambda^{\square}$ because m varies:

$$\mathsf{nest}(m) \approx \mathtt{box}\ (\mathtt{letbox}\ u_m \leftarrow \mathtt{box}\ (\cdots(\mathtt{letbox}\ u_1 \leftarrow \mathtt{box}\ \mathsf{zero}\ \mathtt{in}\ u_1)\cdots)\ \mathtt{in}\ u_m)$$

n-layered Generalization

$$\Gamma_{n-1}; \cdots; \Gamma_1; \Gamma_0 \vdash_i t : T \qquad \text{or} \qquad \overrightarrow{\Gamma} \vdash_i t : T \qquad \text{where } i \in [0, n-1].$$

Lemma (Static code)

If $i \in [0, n-2]$ and $\Psi; \Gamma \vdash_i t \approx s : T$, then $t = s$.

Lemma (Lifting)

If $\overrightarrow{\Gamma} \vdash_i t : T$ and $0 \leq i \leq j < n$, then $\overrightarrow{\Gamma} \vdash_j t : T$.