Foundations and Applications of Modal Type Theories

Jason Z. S. Hu

McGill University

PhD Defense

Nov. 21st, 2024

Outline

- Motivations

Outline

- Motivations
  - Type theory: what and why?

Outline

- Motivations
  - Type theory: what and why?
  - Problem: how to extend type theory with meta-programming?

Outline

- Motivations
  - Type theory: what and why?
  - Problem: how to extend type theory with meta-programming?
- Contributions: meta-programming in type theory

Outline

- Motivations
  - Type theory: what and why?
  - Problem: how to extend type theory with meta-programming?
- Contributions: meta-programming in type theory
  - MINT: Quasi-quotation

Outline

- Motivations
  - Type theory: what and why?
  - Problem: how to extend type theory with meta-programming?
- Contributions: meta-programming in type theory
  - MINT: Quasi-quotation
  - DELAM: Recursion on syntactic objects

Outline

- Motivations
  - Type theory: what and why?
  - Problem: how to extend type theory with meta-programming?
- Contributions: meta-programming in type theory
  - MINT: Quasi-quotation
  - DELAM: Recursion on syntactic objects
- Conclusions

Type Theory: What and Why

- Theoretic foundation of popular proof assistants (Coq, Agda, Lean)

Type Theory: What and Why

- Theoretic foundation of popular proof assistants (Coq, Agda, Lean)
  - CompCert, CertikOS

Type Theory: What and Why

- Theoretic foundation of popular proof assistants (Coq, Agda, Lean)
    - CompCert, CertikOS
    - 4 color theorem, mathlib

Type Theory: What and Why

- Theoretic foundation of popular proof assistants (Coq, Agda, Lean)
    - CompCert, CertikOS
    - 4 color theorem, mathlib
- Easy to understand and implement

Type Theory: What and Why

- Theoretic foundation of popular proof assistants (Coq, Agda, Lean)
    - CompCert, CertikOS
    - 4 color theorem, mathlib
- Easy to understand and implement
- Propositions-as-types: same language for programming and proving

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult m          n = ?
```

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult zero     n = ?
mult (succ m) n = ?
```

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult zero     n = 0
mult (succ m) n = ?
```

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult zero     n = 0
mult (succ m) n = ?
```

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult zero     n = 0
mult (succ m) n = mult m n + n
```

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult zero     n = 0
mult (succ m) n = mult m n + n
```

- Type theory is also a proving language!

```
left-id-mult : ∀ (m : Nat) → mult 1 m ≡ m
left-id-mult m = refl
```

Programming and Proving in Type Theory

- Type theory is a programming language!

```
mult : Nat → Nat → Nat
mult zero     n = 0
mult (succ m) n = mult m n + n
```

- Type theory is also a proving language!

```
left-id-mult : ∀ (m : Nat) → mult 1 m ≡ m
left-id-mult m = refl
```

- Type theory knows `mult 1 m` ≈ `mult 0 m + m` ≈ `0 + m` ≈ `m`

Spend Less Time on Proving

- Good news: an algorithm to check whether a program has the specified type

Spend Less Time on Proving

- Good news: an algorithm to check whether a program has the specified type
- Good news: computer is faster than human

Spend Less Time on Proving

- Good news: an algorithm to check whether a program has the specified type
- Good news: computer is faster than human
- Bad news: proving in type theory requires every last detail

```
right-id-mult : ∀ (m : Nat) → mult m 1 ≡ m
right-id-mult m          = ?
```

Spend Less Time on Proving

- Good news: an algorithm to check whether a program has the specified type
- Good news: computer is faster than human
- Bad news: proving in type theory requires every last detail

```
right-id-mult : ∀ (m : Nat) → mult m 1 ≡ m
right-id-mult zero     = refl -- 0 ≡ 0
right-id-mult (succ m) = begin
  mult m 1 + 1  =< right-id-mult m >
  m + 1         =< +-comm >
  1 + m         qed
```

Spend Less Time on Proving

- Good news: an algorithm to check whether a program has the specified type
- Good news: computer is faster than human
- Bad news: proving in type theory requires every last detail

```
right-id-mult : ∀ (m : Nat) → mult m 1 ≡ m
right-id-mult zero     = refl -- 0 ≡ 0
right-id-mult (succ m) = begin
  mult m 1 + 1  =< right-id-mult m >
  m + 1         =< +-comm >
  1 + m         qed
```

- Solution: meta-programming, i.e. write programs to generate programs and proofs

Spend Less Time on Proving

- Good news: an algorithm to check whether a program has the specified type
- Good news: computer is faster than human
- Bad news: proving in type theory requires every last detail

```
right-id-mult : ∀ (m : Nat) → mult m 1 ≡ m
right-id-mult m         = crush
```

- Solution: meta-programming, i.e. write programs to generate programs and proofs

Research question:

can a type theory directly support meta-programming?

Spectrum of Meta-programming in Proof Assistants

untyped

dependently typed
(this thesis)

Spectrum of Meta-programming in Proof Assistants

untyped

dependently typed
(this thesis)

Ltac
reflection

Spectrum of Meta-programming in Proof Assistants

untyped

dependently typed
(this thesis)

Ltac
reflection

Ltac2

Spectrum of Meta-programming in Proof Assistants

untyped

dependently typed
(this thesis)

Ltac
reflection

Ltac2

Mtac

Spectrum of Meta-programming in Proof Assistants

untyped

dependently typed
(this thesis)

Ltac
reflection

Ltac2

Mtac

Mint

Part I:

- Hu and Pientka (2022), A Categorical Normalization Proof for the Modal Lambda-Calculus, MFPS'22
- Hu et al. (2023), Normalization by Evaluation for Modal Dependent Type Theory, JFP

Spectrum of Meta-programming in Proof Assistants

untyped

dependently typed
(this thesis)

Ltac
reflection

Ltac2

Mtac

Mint
DeLaM

Part I:

- Hu and Pientka (2022), A Categorical Normalization Proof for the Modal Lambda-Calculus, MFPS'22
- Hu et al. (2023), Normalization by Evaluation for Modal Dependent Type Theory, JFP

Part II:

- Hu and Pientka (2024), Layered Modal Type Theory: Where Meta-programming Meets Intensional Analysis, ESOP'24
- Hu and Pientka (2025), A Dependent Type Theory for Meta-programming with Intensional Analysis, POPL'25

Part I

MINT and Quasi-quotation

Quasi-quotation in MINT

- Extend dependent type theory with the $\Box$ modality

Quasi-quotation in MINT

- Extend dependent type theory with the $\square$ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory

Quasi-quotation in MINT

- Extend dependent type theory with the $\square$ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- $\square$ `A` reads “code of `A`”

Quasi-quotation in MINT

- Extend dependent type theory with the $\square$ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- $\square$ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`

Quasi-quotation in MINT

- Extend dependent type theory with the $\square$ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- $\square$ `A` reads “code of `A`”
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by $\texttt{unbox}_{\mathsf{n}}$ `t`

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by $\texttt{unbox}_n$ `t`
- meta-programming in MINT:

```
mult  : Nat →   Nat → Nat
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ A reads "code of A"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by `unbox`$_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 m         = ?
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
    - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
    - `box t` quotes the code of `t`
    - code splicing and code running are modelled by $\texttt{unbox}_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero      = ?
mult2 (succ m)  = ?
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by `unbox`$_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero     = box ?
mult2 (succ m) = ?
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by $\texttt{unbox}_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero     = box (λ n. 0)
mult2 (succ m) = ?
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by $\texttt{unbox}_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero     = box (λ n. 0)
mult2 (succ m) = ?
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
    - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
    - `box t` quotes the code of `t`
    - code splicing and code running are modelled by `unbox`$_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero      = box (λ n. 0)
mult2 (succ m)  = box (λ n.          ?           )
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by $\mathtt{unbox}_{n}$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero     = box (λ n. 0)
mult2 (succ m) = box (λ n.          (mult2 m)   n + n)
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads “code of `A`”
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by `unbox`$_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero     = box (λ n. 0)
mult2 (succ m) = box (λ n. (unbox₁ (mult2 m)) n + n)
```

Quasi-quotation in MINT

- Extend dependent type theory with the □ modality
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- □ `A` reads "code of `A`"
  Quasi-quotation:
  - `box t` quotes the code of `t`
  - code splicing and code running are modelled by `unbox`$_n$ `t`
- meta-programming in MINT:

```
mult2 : Nat → □(Nat → Nat)
mult2 zero     = box (λ n. 0)
mult2 (succ m) = box (λ n. (unbox₁ (mult2 m)) n + n)
```

- Running meta-programs:

```
unbox₀ (mult2 2) ≈ (λ n. n + n)
```

Proving in MINT

- MINT has dependent types:

Proving in MINT

- MINT has dependent types: it also proves!

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound m        n = ?
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound m         n = ?
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = ?
sound (succ m) n = ?
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = ?
sound (succ m) n = ?
```

LHS:

$$(\mathtt{unbox}_0\ (\mathtt{mult2}\ 0))\ n \approx (\mathtt{unbox}_0\ (\mathtt{box}\ (\lambda\ n.\ 0)))\ n \approx (\lambda\ n.\ 0)\ n \approx 0$$

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = ?
sound (succ m) n = ?
```

LHS:

```
(unbox₀ (mult2 0)) n ≈ (unbox₀ (box (λ n. 0))) n ≈ (λ n. 0) n ≈ 0
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = ?
sound (succ m) n = ?
```

LHS:

```
(unbox₀ (mult2 0)) n ≈ (unbox₀ (box (λ n. 0))) n ≈ (λ n. 0) n ≈ 0
```

RHS:

```
mult 0 n ≈ 0
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox_0 (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

LHS:

```
(unbox_0 (mult2 0)) n ≈ (unbox_0 (box (λ n. 0))) n ≈ (λ n. 0) n ≈ 0
```

RHS:

```
mult 0 n ≈ 0
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

LHS:

```
  (unbox₀ (mult2 (succ m))) n
≈ (unbox₀ (box (λ n. (unbox₁ (mult2 m)) n + n))) n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

LHS:

```
    (unbox₀ (mult2 (succ m))) n
≈            (λ n. (unbox₀ (mult2 m)) n + n)   n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

LHS:

```
   (unbox₀ (mult2 (succ m))) n
≈                    (unbox₀ (mult2 m)) n + n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

LHS:

```
   (unbox₀ (mult2 (succ m))) n
≈                     (unbox₀ (mult2 m)) n + n
```

RHS:

```
   mult (succ m) n ≈ mult m n + n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = ?
```

LHS:

```
(unbox₀ (mult2 (succ m))) n
≈                    (unbox₀ (mult2 m)) n + n
```

RHS:

```
mult (succ m) n ≈ mult m n + n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = cong (_+ n) ?
```

LHS:

```
  (unbox₀ (mult2 (succ m))) n
≈                  (unbox₀ (mult2 m)) n + n
```

RHS:

```
  mult (succ m) n ≈ mult m n + n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = cong (_+ n) ?
```

```
LHS:
       (unbox₀ (mult2 (succ m))) n
    ≈                   (unbox₀ (mult2 m)) n + n
RHS:
       mult (succ m) n ≈ mult m n + n
```

Proving in MINT

- MINT has dependent types: it also proves!
- Soundness: evaluating `mult2` computes the same as `mult`

```
sound : ∀ (m n : Nat) → (unbox₀ (mult2 m)) n ≡ mult m n
sound zero     n = refl
sound (succ m) n = cong (_+ n) (sound m n)
```

LHS:

```
  (unbox₀ (mult2 (succ m))) n
≈                    (unbox₀ (mult2 m)) n + n
```

RHS:

```
  mult (succ m) n ≈ mult m n + n
```

Applications and Limitations of MINT

- Advantage: MINT models quasi-quotation and can prove correctness of meta-programs

Applications and Limitations of MINT

- Advantage: MINT models quasi-quotation and can prove correctness of meta-programs
    - We can extract proven-correct meta-programs to MetaML, MetaOCaml, etc.

Applications and Limitations of Mint

- Advantage: Mint models quasi-quotation and can prove correctness of meta-programs
  - We can extract proven-correct meta-programs to MetaML, MetaOCaml, etc.
- Limitation: Mint supports composition only; does not support recursion on syntactic objects

Applications and Limitations of MINT

- Advantage: MINT models quasi-quotation and can prove correctness of meta-programs
  - We can extract proven-correct meta-programs to MetaML, MetaOCaml, etc.
- Limitation: MINT supports composition only; does not support recursion on syntactic objects
  - We frequently use it when implementing proof heuristics and tactics in proof assistants!

Applications and Limitations of MINT

- Advantage: MINT models quasi-quotation and can prove correctness of meta-programs
  - We can extract proven-correct meta-programs to MetaML, MetaOCaml, etc.
- Limitation: MINT supports composition only; does not support recursion on syntactic objects
  - We frequently use it when implementing proof heuristics and tactics in proof assistants!

Can we support recursion on syntactic objects in a type theory?

Part II

DeLaM and Recursion on Syntactic Objects

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal
  - If the goal is a known truth, then it's done;

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal
  - If the goal is a known truth, then it's done;
  - if the goal is a conjunction, then we need to prove each component;

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal
    - If the goal is a known truth, then it's done;
    - if the goal is a conjunction, then we need to prove each component;
    - etc.

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal
  - If the goal is a known truth, then it's done;
  - if the goal is a conjunction, then we need to prove each component;
  - etc.
- MINT does not support this kind of analysis!

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal
  - If the goal is a known truth, then it's done;
  - if the goal is a conjunction, then we need to prove each component;
  - etc.
- MINT does not support this kind of analysis!
- In general, we need to do recursion on the syntactic object of the current goal.

Analysis / Recursion on Syntactic Objects

- A proof heuristic often needs to know the shape of the goal
  - If the goal is a known truth, then it's done;
  - if the goal is a conjunction, then we need to prove each component;
  - etc.
- MINT does not support this kind of analysis!
- In general, we need to do recursion on the syntactic object of the current goal.
- A *different* type theory is needed.

Layers and DELAM

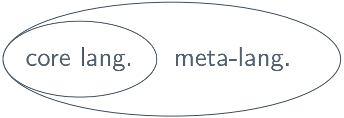
- DELAM: **De**pendent **La**yered **M**odal type theory

Layers and DeLaM

- DeLaM: **De**pendent **La**yered **M**odal type theory
    - extends dependent type theory with *layers*

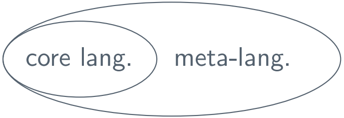
Layers and DELAM

- DELAM: **De**pendent **La**yered **M**odal type theory
  - extends dependent type theory with *layers*
  -



Layers and DeLaM

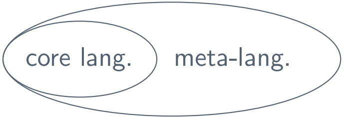
- DeLaM: **De**pendent **La**yered **M**odal type theory
  - extends dependent type theory with *layers*
  -



- Meta-language is an extension of core language and is strictly more expressive:

Layers and DELAM

- DELAM: **De**pendent **La**yered **M**odal type theory
  - extends dependent type theory with *layers*
  -



- Meta-language is an extension of core language and is strictly more expressive:
  - coherent recursion only on syntactic objects of the core language

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 m          = ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 m        = ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = ?
mult3 (succ m) = ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← ?          in ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in ?
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero      = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in box (x. u[x/x] + x)
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in box (x. u[x/x] + x)
```

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

```
mult3 1 ≈ box (x. 0 + x)         ≉ box (x. x)
mult3 2 ≈ box (x. (0 + x) + x) ≉ box (x. x + x)
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in box (x. u[x/x] + x)
```

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

```
mult3 1 ≈ box (x. 0 + x)           ≉ box (x. x)
mult3 2 ≈ box (x. (0 + x) + x) ≉ box (x. x + x)
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in box (x. u[x/x] + x)
```

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

```
mult3 1 ≈ box (x. 0 + x)         ≉ box (x. x)
mult3 2 ≈ box (x. (0 + x) + x) ≉ box (x. x + x)
```

- Use `letbox` to run the generated function:

```
letbox u ← mult3 2 in u[5/x] ≈ 10
letbox u ← mult3 2 in λ y. u[y/x] ≈ λ y. (0 + y) + y ≈ λ y. y + y
```

Meta-programming in DeLaM

- Multiplication in DeLaM:

```
mult3 : Nat → □ (x : Nat ⊢ Nat)
mult3 zero     = box (x. 0)
mult3 (succ m) = letbox u ← mult3 m in box (x. u[x/x] + x)
```

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

```
mult3 1 ≈ box (x. 0 + x)         ≉ box (x. x)
mult3 2 ≈ box (x. (0 + x) + x) ≉ box (x. x + x)
```

- Use `letbox` to run the generated function:

```
letbox u ← mult3 2 in u[5/x] ≈ 10
letbox u ← mult3 2 in λ y. u[y/x] ≈ λ y. (0 + y) + y ≈ λ y. y + y
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp y                = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = ?
simp (box (a + b)) = ?

simp (box a)       = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) = ?

simp (box a)       = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) = ?

simp (box a)       = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) =
  letbox a' ← simp (box (x. a)) in box (x. a' + b)
simp (box a)       = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) =
  letbox a' ← simp (box (x. a)) in box (x. a' + b)
simp (box a)       = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) =
  letbox a' ← simp (box (x. a)) in box (x. a' + b)
simp (box a)       = ?
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) =
  letbox a' ← simp (box (x. a)) in box (x. a' + b)
simp (box a)       = box a
```

Code Manipulation in DeLaM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) =
  letbox a' ← simp (box (x. a)) in box (x. a' + b)
simp (box a)       = box a
```

- Use `simp` to simplify `0`'s away:

```
mult4 : Nat → □ (x : Nat ⊢ Nat)
mult4 n = simp (mult3 n)
```

Code Manipulation in DELAM

- Getting rid of redundant `0`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp (box (0 + b)) = box (x. b)
simp (box (a + b)) =
  letbox a' ← simp (box (x. a)) in box (x. a' + b)
simp (box a)       = box a
```

- Use `simp` to simplify `0`'s away:

```
mult4 : Nat → □ (x : Nat ⊢ Nat)
mult4 n = simp (mult3 n)
```

- Finally we have the simplest forms:

```
mult4 1 ≈ box (x. x)
mult4 2 ≈ box (x. x + x)
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound y             m = ?
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound y                    m = ?
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = ?
simp-sound (box (a + b)) m = ?


simp-sound (box a)         m = ?
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = ?
simp-sound (box (a + b)) m = ?


simp-sound (box a)       m = ?
```

LHS: `b[m/x]`

Proving in DELAM

Similar to MINT, we can also prove properties about meta-programs in DELAM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = ?
simp-sound (box (a + b)) m = ?


simp-sound (box a)       m = ?
```

LHS: `b[m/x]`

RHS: `0 + b[m/x] ≈ b[m/x]`

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m = ?


simp-sound (box a)       m = ?
```

LHS: `b[m/x]`

RHS: `0 + b[m/x] ≈ b[m/x]`

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m = ?


simp-sound (box a)       m = ?
```

Goal becomes

```
letbox s' ← simp (box (a + b)) in s'[m/x] ≡ (a + b)[m/x]
```

Also

```
simp (box (a + b)) ≈ letbox sa' ← simp (box a) in box (sa' + b)
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in ?

simp-sound (box a)       m = ?
```

Goal becomes

```
letbox s' ← simp (box (a + b)) in s'[m/x] ≡ (a + b)[m/x]
```

Also

```
simp (box (a + b)) ≈ letbox sa' ← simp (box a) in box (sa' + b)
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in ?

simp-sound (box a)       m = ?
```

Goal is unblocked

```
(sa'      + b)[m/x] ≡ (a      + b)[m/x]
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in ?

simp-sound (box a)        m = ?
```

Goal is unblocked

```
sa'[m/x] + b [m/x] ≡  a[m/x] + b [m/x]
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in ?

simp-sound (box a)         m = ?
```

Goal is unblocked

```
sa'[m/x] + b [m/x] ≡  a[m/x] + b [m/x]
```

Proving in DELAM

Similar to MINT, we can also prove properties about meta-programs in DELAM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in
  cong (_+ b[m/x]) ?
simp-sound (box a)       m = ?
```

Goal is unblocked

```
 sa'[m/x] + b [m/x] ≡  a[m/x] + b [m/x]
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in
  cong (_+ b[m/x]) ?
simp-sound (box a)       m = ?
```

Goal is unblocked

```
sa'[m/x] + b [m/x] ≡ a[m/x] + b [m/x]
```

Proving in DELAM

Similar to MINT, we can also prove properties about meta-programs in DELAM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in
  cong (_+ b[m/x]) (simp-sound (box a) m)
simp-sound (box a)         m = ?
```

Goal is unblocked

```
sa'[m/x] + b [m/x] ≡ a[m/x] + b [m/x]
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in
  cong (_+ b[m/x]) (simp-sound (box a) m)
simp-sound (box a)       m = ?
```

Proving in DeLaM

Similar to Mint, we can also prove properties about meta-programs in DeLaM

```
simp-sound : ∀ (y : □ (x : Nat ⊢ Nat)) (m : Nat) →
  letbox y' ← y; s' ← simp (box y') in s'[m/x] ≡ y'[m/x]
simp-sound (box (0 + b)) m = refl
simp-sound (box (a + b)) m =
  letbox sa' ← simp (box a) in
  cong (_+ b[m/x]) (simp-sound (box a) m)
simp-sound (box a)       m = refl
```

Summary of DeLaM

- Recursion on syntactic objects:
    - manipulate terms,
    - analyze and prove goals

Summary of DeLaM

- Recursion on syntactic objects:
    - manipulate terms,
    - analyze and prove goals
- Run code ⇒ conflate the proving and meta-programming languages

Summary of DELAM

- Recursion on syntactic objects:
  - manipulate terms,
  - analyze and prove goals
- Run code ⇒ conflate the proving and meta-programming languages
- DELAM is a basic setup; need empirical study to understand practical needs

To Conclude

Conclusions

This PhD thesis explored ways to support meta-programming in type theory.

Conclusions

This PhD thesis explored ways to support meta-programming in type theory.

- MINT supports quasi-quotation but not recursion on syntactic objects

Conclusions

This PhD thesis explored ways to support meta-programming in type theory.

- MINT supports quasi-quotation but not recursion on syntactic objects
- DELAM supports recursion on syntactic objects but mandates a less familiar programming style

Conclusions

This PhD thesis explored ways to support meta-programming in type theory.

- MINT supports quasi-quotation but not recursion on syntactic objects
- DELAM supports recursion on syntactic objects but mandates a less familiar programming style
- Both type theories are logically consistent and can serve as foundations for proof assistants!

Bibliography

Hu, J. Z. S., Jang, J., and Pientka, B. (2023). Normalization by evaluation for modal dependent type theory. *J. Funct. Program.*, 33.

Hu, J. Z. S. and Pientka, B. (2022). A categorical normalization proof for the modal lambda-calculus. In Hsu, J. and Tasson, C., editors, *Proceedings of the 38th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2022, Cornell University, Ithaca, New York, USA, with a satellite event at IRIF, Denis Diderot University, Paris, France, and online, July 11-13, 2022*, volume 1 of *EPTICS*. EpiSciences.

Hu, J. Z. S. and Pientka, B. (2024). Layered modal type theory: Where meta-programming meets intensional analysis. In Weirich, S., editor, *Proceedings of the 33rd European Symposium on Programming on Programming Languages and Systems, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 52–82. Springer.

Hu, J. Z. S. and Pientka, B. (2025). A dependent type theory for meta-programming with intensional analysis. *Proc. ACM Program. Lang.*, (POPL). To Appear.

Criteria for A Valid Type Theory

- How do we know a type theory works?

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.
- Two conclusive properties:

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.
- Two conclusive properties:
  - **Normalization**: every well-typed program must terminate and compute to a normal form

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.
- Two conclusive properties:
    - **Normalization**: every well-typed program must terminate and compute to a normal form
    - **Consistency** is a corollary of normalization.

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.
- Two conclusive properties:
  - **Normalization**: every well-typed program must terminate and compute to a normal form
  - **Consistency** is a corollary of normalization.
  - **Decidability of convertibility**: decide whether two terms are equivalent

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.
- Two conclusive properties:
    - **Normalization**: every well-typed program must terminate and compute to a normal form
    - **Consistency** is a corollary of normalization.
    - **Decidability of convertibility**: decide whether two terms are equivalent
    - Computers can always decide whether two terms are the "same"

Criteria for A Valid Type Theory

- How do we know a type theory works?
- Intuitively, a type theory should be consistent, i.e. not every proposition is provable.
- Two conclusive properties:
  - **Normalization**: every well-typed program must terminate and compute to a normal form
  - **Consistency** is a corollary of normalization.
  - **Decidability of convertibility**: decide whether two terms are equivalent
  - Computers can always decide whether two terms are the "same"
- Two properties allow to do type-checking, i.e. checking whether a program is a member of the given type

Contributions

Thesis	Part I	Part II
Type theory	MINT	DELAM
Normalization	Yes	Yes
Decidability of convertibility		
Main feature		
Mechanization		

Contributions

Thesis	Part I	Part II
Type theory	MINT	DELAM
Normalization	Yes	Yes
Decidability of convertibility	Yes	Yes
Main feature		
Mechanization		

Contributions

Thesis	Part I	Part II
Type theory	Mint	DeLaM
Normalization	Yes	Yes
Decidability of convertibility	Yes	Yes
Main feature	quasi-quotation	recursion on syntactic objects
Mechanization		

Contributions

Thesis	Part I	Part II
Type theory	MINT	DELAM
Normalization	Yes	Yes
Decidability of convertibility	Yes	Yes
Main feature	quasi-quotation	recursion on syntactic objects
Mechanization	Yes	No

Components of Type Theory

- In type theory, we study judgments.

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

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

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : \Pi(x : S).T}$$

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : \Pi(x : S).T} \qquad \frac{\Gamma \vdash t : \Pi(x : S).T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[s/x]}$$

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : \Pi(x : S).T} \qquad \frac{\Gamma \vdash t : \Pi(x : S).T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[s/x]}$$

A substitution replaces a variable with a term.

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : \Pi(x : S).T} \qquad \frac{\Gamma \vdash t : \Pi(x : S).T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[s/x]}$$

A substitution replaces a variable with a term.

- $\boxed{\Gamma \vdash t \approx t' : T}$ terms t and t' are equivalent.

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : \Pi(x : S).T} \qquad \frac{\Gamma \vdash t : \Pi(x : S).T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[s/x]}$$

A substitution replaces a variable with a term.

- $\boxed{\Gamma \vdash t \approx t' : T}$ terms t and t' are equivalent.

$$\frac{\Gamma, x : S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x.t)\ s \approx t[s/x] : T[s/x]}$$

Components of Type Theory

- In type theory, we study judgments.
- $\boxed{\Gamma \vdash t : T}$ term t has type T in context Γ.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : \Pi(x : S).T} \qquad \frac{\Gamma \vdash t : \Pi(x : S).T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T[s/x]}$$

A substitution replaces a variable with a term.

- $\boxed{\Gamma \vdash t \approx t' : T}$ terms t and t' are equivalent.

$$\frac{\Gamma, x : S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x.t)\ s \approx t[s/x] : T[s/x]}$$

- Equivalence applies to types as well.

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g F                       = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = ?
crush g (box (F₁ ∧ F₂))     = ?


crush g (box ((x:Nat) → F)) = ?

crush g (box _)             = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.

-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))
  crush g (box (Eq Nat a b))   = nat-eq-solve g (box a) (box b)
  crush g (box (F₁ ∧ F₂))      = ?


  crush g (box ((x:Nat) → F)) = ?

  crush g (box _)              = ?
  ```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
crush g (box (F₁ ∧ F₂))     = ?


crush g (box ((x:Nat) → F)) = ?

crush g (box _)             = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-

```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
crush g (box (F₁ ∧ F₂))     = crush g (box F₁)>>=λ (r₁ : □ (g⊢F₁)).
                              crush g (box F₂)>>=λ (r₂ : □ (g⊢F₂)).
  letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
crush g (box ((x:Nat) → F)) = ?

crush g (box _)             = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.

-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))
  crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
  crush g (box (F₁ ∧ F₂))      = crush g (box F₁) >>= λ (r₁ : □ (g ⊢ F₁)).
                                crush g (box F₂) >>= λ (r₂ : □ (g ⊢ F₂)).
    letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
  crush g (box ((x:Nat) → F)) = ?

  crush g (box _)             = ?
  ```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.

-

```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
crush g (box (F₁ ∧ F₂))     = crush g (box F₁) >>= λ (r₁ : □ (g ⊢ F₁)).
                              crush g (box F₂) >>= λ (r₂ : □ (g ⊢ F₂)).
  letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
crush g (box ((x:Nat) → F)) = ?

crush g (box _)             = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
crush g (box (F₁ ∧ F₂))     = crush g (box F₁)>>=λ (r₁ : □ (g⊢F₁)).
                              crush g (box F₂)>>=λ (r₂ : □ (g⊢F₂)).
  letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
crush g (box ((x:Nat) → F)) = ?

crush g (box _)             = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.

-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))
  crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
  crush g (box (F₁ ∧ F₂))     = crush g (box F₁)>>=λ (r₁ : □ (g⊢F₁)).
                                crush g (box F₂)>>=λ (r₂ : □ (g⊢F₂)).
    letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
  crush g (box ((x:Nat) → F)) = crush (g, x:Nat) (box F)>>=
    λ (r : □ (g,x:Nat⊢F)). letbox pf ← r in Some (box (λ x. pf))
  crush g (box _)             = ?
  ```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
crush g (box (F1 ∧ F2))     = crush g (box F1)>>=λ (r1 : □ (g⊢F1)).
                              crush g (box F2)>>=λ (r2 : □ (g⊢F2)).
  letbox pf1 ← r1 ; pf2 ← r2 in Some (box (pf1, pf2))
crush g (box ((x:Nat) → F)) = crush (g, x:Nat) (box F)>>=
  λ (r : □ (g,x:Nat⊢F)). letbox pf ← r in Some (box (λ x. pf))
crush g (box _)             = ?
```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))
  crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
  crush g (box (F₁ ∧ F₂))     = crush g (box F₁)>>=λ (r₁ : □ (g⊢F₁)).
                                crush g (box F₂)>>=λ (r₂ : □ (g⊢F₂)).
    letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
  crush g (box ((x:Nat) → F)) = crush (g, x:Nat) (box F)>>=
    λ (r : □ (g,x:Nat⊢F)). letbox pf ← r in Some (box (λ x. pf))
  crush g (box _)             = ?
  ```

Tactics in DeLaM

- Tactics in proof assistants usually analyze the structure of the current goal as a type.
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = nat-eq-solve g (box a) (box b)
crush g (box (F₁ ∧ F₂))     = crush g (box F₁)>>=λ (r₁ : □ (g⊢F₁)).
                              crush g (box F₂)>>=λ (r₂ : □ (g⊢F₂)).
  letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
crush g (box ((x:Nat) → F)) = crush (g, x:Nat) (box F)>>=
  λ (r : □ (g,x:Nat⊢F)). letbox pf ← r in Some (box (λ x. pf))
crush g (box _)             = None
```

Invoking Tactics in DeLaM

- Use tactics to avoid tedious proving steps

Invoking Tactics in DeLaM

- Use tactics to avoid tedious proving steps

-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))
  ```

Invoking Tactics in DeLaM

- Use tactics to avoid tedious proving steps
-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))

  lem : (x y : Nat) → Eq Nat (x + y) (y + x) ∧
                 ((z : Nat) → Eq Nat (x + (y + z)) (z + (y + x)))
  lem =
    let Some pf ← crush ()
      (box ((x y : Nat) → Eq Nat (x + y) (y + x) ∧
              ((z : Nat) → Eq Nat (x + (y + z)) (z + (y + x)))))
    in letbox u ← pf in u
  ```

Invoking Tactics in DeLaM

- Use tactics to avoid tedious proving steps
-
```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))

lem : (x y : Nat) → Eq Nat (x + y) (y + x) ∧
                  ((z : Nat) → Eq Nat (x + (y + z)) (z + (y + x))
lem =
  let Some pf ← crush ()
    (box ((x y : Nat) → Eq Nat (x + y) (y + x) ∧
                ((z : Nat) → Eq Nat (x + (y + z)) (z + (y + x)))))
  in letbox u ← pf in u
```

Invoking Tactics in DeLaM

- Use tactics to avoid tedious proving steps

-
  ```
  crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
    letbox F' ← F in Option (□ (g ⊢ F'))

  lem : (x y : Nat) → Eq Nat (x + y) (y + x) ∧
               ((z : Nat) → Eq Nat (x + (y + z)) (z + (y + x)))
  lem =
    let Some pf ← crush ()
      (box ((x y : Nat) → Eq Nat (x + y) (y + x) ∧
              ((z : Nat) → Eq Nat (x + (y + z)) (z + (y + x)))))
    in letbox u ← pf in u
  ```