Normalization by Evaluation for Modal Dependent Type Theory

Jason Hu Junyoung Jang Brigitte Pientka

McGill University

JFP First at ICFP 2024

HU, J. Z. S., JANG, J., & PIENTKA, B. (2023). Normalization by evaluation for modal dependent type theory. Journal of Functional Programming, 33, e7. doi:10.1017/S0956796823000060

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures
  - extends simply typed $\lambda^{\square}$ by Davies and Pfenning (2001)

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures
  - extends simply typed $\lambda^{\square}$ by Davies and Pfenning (2001)
  - MINT, **M**odal **IN**tuitionistic **T**ype theory

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures
  - extends simply typed $\lambda^{\square}$ by Davies and Pfenning (2001)
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- In meta-programming, $\square A$ reads "code of A"

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures
  - extends simply typed $\lambda^{\square}$ by Davies and Pfenning (2001)
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- In meta-programming, $\square A$ reads "code of A"
  - box t quotes the code of t

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\Box$ modality with different structures
  - extends simply typed $\lambda^{\Box}$ by Davies and Pfenning (2001)
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- In meta-programming, $\Box A$ reads "code of A"
  - $\texttt{box}\ t$ quotes the code of t
  - code splicing and code running are modelled by $\texttt{unbox}_n\ t$

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures
  - extends simply typed $\lambda^{\square}$ by Davies and Pfenning (2001)
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- In meta-programming, $\square A$ reads "code of A"
  - box t quotes the code of t
  - code splicing and code running are modelled by $\texttt{unbox}_n\ t$
  - Kripke-style formulation

High-level Ideas

- Combine Martin-Löf type theory (MLTT) and the $\square$ modality with different structures
  - extends simply typed $\lambda^{\square}$ by Davies and Pfenning (2001)
  - MINT, **M**odal **IN**tuitionistic **T**ype theory
- In meta-programming, $\square A$ reads "code of A"
  - box t quotes the code of t
  - code splicing and code running are modelled by $\texttt{unbox}_n\ t$
  - Kripke-style formulation
- means to explore meta-programming in dependent type theory

Contributions

- Mint, a modal dependent type theory

Contributions

- MINT, a modal dependent type theory
  - extension of MLTT

Contributions

- Mint, a modal dependent type theory
  - extension of MLTT
  - full cumulative universes

Contributions

- MINT, a modal dependent type theory
  - extension of MLTT
  - full cumulative universes
  - $\square$ in the Kripke style

Contributions

- MINT, a modal dependent type theory
  - extension of MLTT
  - full cumulative universes
  - $\square$ in the Kripke style
  - equational theory with explicit substitutions

Contributions

- MINT, a modal dependent type theory
  - extension of MLTT
  - full cumulative universes
  - $\square$ in the Kripke style
  - equational theory with explicit substitutions
- A complete and sound normalization-by-evaluation (NbE) algorithm based on an untyped domain

Contributions

- MINT, a modal dependent type theory
  - extension of MLTT
  - full cumulative universes
  - $\square$ in the Kripke style
  - equational theory with explicit substitutions
- A complete and sound normalization-by-evaluation (NbE) algorithm based on an untyped domain
- A full mechanization in Agda ($\sim$ 11k LoC) available online

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift n         = ?
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = ?
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = ?
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero       = box zero
lift (succ n) = box (succ ?)
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift n          = ?
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift n          = ?
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = ?
unbox-lift (succ n) = ?
```

- Base case:

```
unbox₀ (box zero) ≡ zero
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = ?
unbox-lift (succ n) = ?
```

- Base case:

```
zero ≡ zero
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = ?
```

- Base case:

```
zero ≡ zero
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = ?
```

- In the step case:

```
unbox₀ (box (succ (unbox₁ (lift n)))) ≡ succ n
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = ?
```

- In the step case:

```
succ (unbox₀ (lift n))   ≡ succ n
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = cong succ ?
```

- After congruence,

```
(unbox₀ (lift n))   ≡      n
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = cong succ ?
```

- Recursive call gives exactly that

```
unbox-lift n :    (unbox₀ (lift n))   ≡     n
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = cong succ (unbox-lift n)
```

- Recursive call gives exactly that

```
unbox-lift n :    (unbox₀ (lift n))   ≡      n
```

Meta-programs and Their Properties

- `lift n` lifts into a `box`; returns `box n`

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

- Soundness: evaluating `lift` gives the same number back

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl
unbox-lift (succ n) = cong succ (unbox-lift n)
```

- Recursive call gives exactly that

```
unbox-lift n :    (unbox₀ (lift n))   ≡       n
```

- MINT as a program logic for MetaML

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash t : T$$

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \overrightarrow{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \overrightarrow{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

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

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \overrightarrow{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

$$\frac{x : T \in \Gamma}{\overrightarrow{\Gamma}; \Gamma \vdash x : T} \qquad \frac{\overrightarrow{\Gamma}; \cdot \vdash t : T}{\overrightarrow{\Gamma} \vdash \texttt{box}\ t : \square T}$$

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

$$\frac{x : T \in \Gamma}{\vec{\Gamma}; \Gamma \vdash x : T} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \texttt{box}\ t : \square T} \qquad \frac{\vec{\Gamma} \vdash t : \square T \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \texttt{unbox}_n\ t : T}$$

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \overrightarrow{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

$$\frac{x : T \in \Gamma}{\overrightarrow{\Gamma}; \Gamma \vdash x : T} \qquad \frac{\overrightarrow{\Gamma}; \cdot \vdash T : \mathtt{Ty}}{\overrightarrow{\Gamma} \vdash \square T : \mathtt{Ty}} \qquad \frac{\overrightarrow{\Gamma}; \cdot \vdash t : T}{\overrightarrow{\Gamma} \vdash \mathtt{box}\ t : \square T} \qquad \frac{\overrightarrow{\Gamma} \vdash t : \square T \qquad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \mathtt{unbox}_n\ t : T}$$

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots ; \Gamma_n \vdash t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

$$\frac{x : T \in \Gamma}{\vec{\Gamma}; \Gamma \vdash x : T} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}}{\vec{\Gamma} \vdash \square T : \mathtt{Ty}} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \mathtt{box}\ t : \square T} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty} \quad \vec{\Gamma} \vdash t : \square T \quad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathtt{unbox}_n\ t : T}$$

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

$$\frac{x : T \in \Gamma}{\vec{\Gamma}; \Gamma \vdash x : T} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}}{\vec{\Gamma} \vdash \square T : \mathtt{Ty}} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \mathtt{box}\ t : \square T} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty} \quad \vec{\Gamma} \vdash t : \square T \quad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathtt{unbox}_n\ t : T[\vec{I}; \Uparrow^n]}$$

Kripke Style and Kripke Semantics

- A stack of contexts: each context represents a Kripke world

$$\epsilon; \Gamma_1; \cdots; \Gamma_n \vdash t : T \qquad \text{or} \qquad \vec{\Gamma} \vdash t : T$$

- $\square$ denotes the next world; move among worlds by its introduction and elimination
- Our formulation extends $\lambda^{\square}$ by Davies and Pfenning (2001)

$$\frac{x : T \in \Gamma}{\vec{\Gamma}; \Gamma \vdash x : T} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}}{\vec{\Gamma} \vdash \square T : \mathtt{Ty}} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : T}{\vec{\Gamma} \vdash \mathtt{box}\ t : \square T} \qquad \frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty} \quad \vec{\Gamma} \vdash t : \square T \quad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \mathtt{unbox}_n\ t : T[\vec{I}; \Uparrow^n]}$$

Modal extension $\vec{I}; \Uparrow^n$ is a special substitution which fixes the context stack.

Handling Context Stack with Substitutions

- A modal extension adds contexts to the domain stack (n could be 0)

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad |\overrightarrow{\Gamma}'| = n}{(\overrightarrow{\Gamma}; \overrightarrow{\Gamma}') \vdash (\overrightarrow{\sigma}; \Uparrow^n) : (\overrightarrow{\Delta}; \cdot)}$$

Handling Context Stack with Substitutions

- A modal extension adds contexts to the domain stack (n could be 0)

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad |\vec{\Gamma}'| = n}{(\vec{\Gamma}; \vec{\Gamma}') \vdash (\vec{\sigma}; \Uparrow^n) : (\vec{\Delta}; \cdot)}$$

- Important to get the β rule right

$$\frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma}; \cdot \vdash t : T \qquad |\vec{\Gamma}'| = n}{\vec{\Gamma}; \vec{\Gamma}' \vdash \mathtt{unbox}_n\ (\mathtt{box}\ t) \approx t[\vec{I}; \Uparrow^n] : T[\vec{I}; \Uparrow^n]}$$

Handling Context Stack with Substitutions

- A modal extension adds contexts to the domain stack (n could be 0)

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad |\vec{\Gamma}'| = n}{(\vec{\Gamma}; \vec{\Gamma}') \vdash (\vec{\sigma}; \Uparrow^n) : (\vec{\Delta}; \cdot)}$$

- Important to get the β rule right

$$\frac{\vec{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma}; \cdot \vdash t : T \qquad |\vec{\Gamma}'| = n}{\vec{\Gamma}; \vec{\Gamma}' \vdash \mathtt{unbox}_n\ (\mathtt{box}\ t) \approx t[\vec{I}; \Uparrow^n] : T[\vec{I}; \Uparrow^n]}$$

- $\Box T$ is η expandable

$$\frac{\vec{\Gamma} \vdash t : \Box T}{\vec{\Gamma} \vdash t \approx \mathtt{box}\ (\mathtt{unbox}_1\ t) : \Box T}$$

Normalization by Evaluation

- Based on an untyped domain model (Abel, 2013)

Normalization by Evaluation

- Based on an untyped domain model (Abel, 2013)
- We give an explicit normalization algorithm to β-η normal forms

Normalization by Evaluation

- Based on an untyped domain model (Abel, 2013)
- We give an explicit normalization algorithm to β-η normal forms
- Two steps:

Normalization by Evaluation

- Based on an untyped domain model (Abel, 2013)
- We give an explicit normalization algorithm to β-η normal forms
- Two steps:
  1. evaluation: evaluate a well-typed term into a domain value and eliminate all β redexes

Normalization by Evaluation

- Based on an untyped domain model (Abel, 2013)
- We give an explicit normalization algorithm to β-η normal forms
- Two steps:
  1. evaluation: evaluate a well-typed term into a domain value and eliminate all β redexes
  2. readback: read from a domain value back to a normal form and perform type-directed η expansion during the process.

Completeness and Soundness

- Completeness: equivalent terms have equal normal forms.

Completeness and Soundness

- Completeness: equivalent terms have equal normal forms.
- Soundness: well-typed terms are equivalent to their normal forms.

Completeness and Soundness

- Completeness: equivalent terms have equal normal forms.
- Soundness: well-typed terms are equivalent to their normal forms.
- Deciding whether t and t' of type T are equivalent: just compare normal forms of t and t'

Completeness and Soundness

- Completeness: equivalent terms have equal normal forms.
- Soundness: well-typed terms are equivalent to their normal forms.
- Deciding whether t and t' of type T are equivalent: just compare normal forms of t and t'
- The normalization algorithm and its completeness and soundness theorems are mechanized in Agda ($\sim$ 11k LoC)

Limitations and Big Picture

- MINT models meta-programming and serves as a program logic for MetaML

Limitations and Big Picture

- MINT models meta-programming and serves as a program logic for MetaML
- MINT does not support intensional analysis (recursion on the structure of code)

Limitations and Big Picture

- MINT models meta-programming and serves as a program logic for MetaML
- MINT does not support intensional analysis (recursion on the structure of code)
- A different flavor: layered modal type theory: simple types (Hu and Pientka, 2024b); dependent types (Hu and Pientka, 2024a)

Limitations and Big Picture

- MINT models meta-programming and serves as a program logic for MetaML
- MINT does not support intensional analysis (recursion on the structure of code)
- A different flavor: layered modal type theory: simple types (Hu and Pientka, 2024b); dependent types (Hu and Pientka, 2024a)
- Modal type theories have solutions in different flavors

Takeaways

- MINT, extending MLTT with $\Box$ and explicit substitutions, equipped with an equational theory

Takeaways

- Mint, extending MLTT with $\square$ and explicit substitutions, equipped with an equational theory
- A complete and sound normalization-by-evaluation (NbE) algorithm based on an untyped domain

Takeaways

- MINT, extending MLTT with $\square$ and explicit substitutions, equipped with an equational theory
- A complete and sound normalization-by-evaluation (NbE) algorithm based on an untyped domain
- A full mechanization in Agda ($\sim$ 11k LoC), available at `https://doi.org/10.5281/zenodo.13363186`

Takeaways

- MINT, extending MLTT with □ and explicit substitutions, equipped with an equational theory
- A complete and sound normalization-by-evaluation (NbE) algorithm based on an untyped domain
- A full mechanization in Agda ($\sim$ 11k LoC), available at `https://doi.org/10.5281/zenodo.13363186`

zhong.s.hu at mail.mcgill.ca

Bibliography

Abel, A. (2013). *Normalization by evaluation: dependent types and impredicativity*. Habilitation thesis, Ludwig-Maximilians-Universität München.

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

Hu, J. Z. S. and Pientka, B. (2024a). Delam: A dependent layered modal type theory for meta-programming. *CoRR*, abs/2404.17065.

Hu, J. Z. S. and Pientka, B. (2024b). Layered modal type theory - where meta-programming meets intensional analysis. In Weirich, S., editor, *Programming Languages and Systems - 33rd European Symposium on Programming, 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, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 52–82. Springer.

Other Theorems

Theorem (Completeness)

If $\overrightarrow{\Gamma} \vdash t \approx t' : T$, *then* $\text{nbe}^{T}_{\vec{\Gamma}}(t) = \text{nbe}^{T}_{\vec{\Gamma}}(t')$.

Theorem (Soundness)

If $\overrightarrow{\Gamma} \vdash t : T$, *then* $\overrightarrow{\Gamma} \vdash t \approx \text{nbe}^{T}_{\vec{\Gamma}}(t) : T$.

Theorem (Consistency)

There is no closed term of type $\Pi(x : Ty_i).x$.