A Modal Analysis of Dependently-Typed Meta-Programming

Brigitte Pientka

McGill University
Montreal, Canada

joint work with
Jason Z. Hu and Junyoung Jang

Special Session in honor of Frank Pfenning at LFMTP'22

Blast from the past – 25 years ago

Some time in the summer of 1997:

> *”In the report ’The practice of logical frameworks’ by Frank Pfenning (FB Mathematik, February 1996, pre-print 1813) it is stated that the current degree of automation in logical frameworks is not satisfactory. Therefore it is suggested to look for ways to apply techniques from inductive theorem proving in the realm of logical frameworks to automate some of these proofs. ”*

Blast from the past – 25 years ago

Some time in the summer of 1997:

> *"In the report 'The practice of logical frameworks' by Fra*
> *Mathematik, February 1996, pre-print 1813) i*
> *degree of automation in logical f*
> *suggested to look f ... eorem proving*
> *in the ... e of these proofs. "*

This Talk:
A Modal Analysis of Dependently-Typed Meta-Programming

What is meta-programming?

What is **meta-programming**?

The **art of writing programs** that
produce other programs.

What is **meta-programming**?

The **art of writing programs** that
produce and manipulate other programs
within the same language.

What is **meta-programming**?

The **art of writing programs** that
produce and manipulate other programs
within the same language.

- *(Quasi)quotation* of `box(2 + 2)` represents an abstract syntax tree (AST) of the expression `2 + 2`.
- *Unquote* splices in code fragments, for example `box(2 + unbox(square 2))` evaluates to the code `box(2 + 2 * 2)` provided `square 2` generates code `box(2 * 2)`.

Good and Bad of Meta-Programming

The Good:

- Widely used and supported
  (Lisp, Scheme, TemplateHaskell, MetaML, TemplateCoq, ...)
- Performance gains due to domain-specific optimizations
  (example: matrix / vector multiplication in machine learning)

The Bad:

- Hard to test the code generation and the generated code
- Hard to reason about code generation and generated code
- Errors are caught during run-time instead of generation time

Good and Bad of Meta-Programming

The Good:

- Widely used and supported
  (Lisp, Scheme, TemplateHaskell, MetaML, TemplateCoq, ...)
- Performance gains due to domain-specific optimizations
  (example: matrix / vector multiplication in machine learning)

The Bad:

- Hard to test the code ge
- Hard to reas
- Errors are cau

Types for the rescue!

- Catch errors early at compile-time / generation time
- Reason about code generation and generated code

Promise and Reality of Meta-Programming in a Type Theory

Promise and Reality of Meta-Programming in a Type Theory

The Promise: Write type-safe tactics or macros within the same language.

- users do not need to learn and use a separate tactic language
- we can prove properties about tactics themselves
- ultimately increases the trust in the overall proof and tactic development

Current Reality: Some external mechanisms requiring extra maintenance of their tactic engines and tactics.

A glimpse of the past

Simply-Typed Meta-Programming

Early 2000: A Modal Analysis of Simply-Typed Metaprogramming

Early 2000: A Modal Analysis of Simply-Typed Metaprogramming

- 1995 *F. Pfenning and Hao-Chi Wong*. On a modal lambda calculus for S4, MFPS'95
- 1996 *R. Davies and F. Pfenning*. A A modal analysis of staged computation, POPL'96
- 2001 *R. Davies and F. Pfenning*. A modal analysis of staged computation, JACM'01

  F. Pfenning and R. Davies. A judgmental reconstruction of modal logic, MSCS'01

Key Idea: $\Box\tau$ describes closed code of type τ

Early 2000...

1995 *F. Pfenning and Hao-Chi Wong*. On a modal lambda calculus for S4, MFPS'95

1996 *R. Davies and F. Pfenning*. A A modal analysis of staged computation, POPL'96

2001 *R. Davies and F. Pfenning*. A modal analysis of staged computation, JACM'01

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic, MSCS'01

Simply Typed Modal Lambda Calculus with box-modality

- Explicit Modal Lambda Calculus – with box and let-box

  Distinguish between global and local assumptions – two zones

- Implicit **Kripke-style Modal Lambda-Calculus** with box and unbox

  Kripke-style context stack $\Gamma_0; \ldots; \Gamma_n$ where variables at stage i are bound in the context Γ_i

Early 2000...

1995 *F. Pfenning and Hao-Chi Wong*. On a modal lambda calculus for S4, MFPS'95

1996 *R. Davies and F. Pfenning*. A A modal analysis of staged computation, POPL'96

2001 *R. Davies and F. Pfenning*. A modal analysis of staged computation, JACM'01

F. Pfenning and R. Davies. A judgmental reconstruction of modal logic, MSCS'01

Simply Typed Modal Lambda Calculus with box-modality

- Explicit Modal Lambda Calculus – with box and let-box
  Distinguish between global and local assumptions – two zones
- Implicit **Kripke-style Modal Lambda-Calculus** with box and unbox
  Kripke-style context stack $\Gamma_0; \ldots; \Gamma_n$ where variables at stage i are bound in the context Γ_i

The Unusual Effectiveness of Modal Types

Modal Lambda-Calculi directly derived from Pfenning and Davies' work have been used in a wide range of seemingly unconnected applications:

- Contextual Modal Type Theory [Pfenning, Nanevski, Pientka 2008]
- Modelling meta-variables and unification alg. [Abel, Pientka, Pfenning 2003, 2006, 2011]
- Mechanizing meta-theory [Pientka et al 2008,. . ., 2019]
- Modelling holes in proofs and programs [Pfenning, Pientka, Nanevski 2008, Cyrus et al 2019]
- Programming with algebraic effects [Nanevski et al 2021]
- Reasoning about universes in homotopy type [Licata, Shulman, et al 2015, 2018]
- . . .

The Unusual Effectiveness of Modal Types

Modal Lambda-Calculi directly derived from Pfenning and Davies' work have been used in a wide range of seemingly unconnected applications:

- Contextual Modal Type Theory [Pfenning, Nanevski, Pientka 2008]
- Modelling meta-variables and unification alg. [Abel, Pientka, Pfenning 2003, 2006, 2011]
- Mechanizing meta-theory [Pientka et al 2008,. . ., 2019]
- Modelling holes in proofs and programs [Pfenning, Pientka, Nanevski 2008, Cyrus et al 2019]
- Programming with algebraic effects [Nanevski et al 2021]
- Reasoning about universes in homotopy type [Licata, Shulman, et al 2015, 2018]
- . . .

Yet, a practical and sound modal type theory that supports meta-programming has been elusive.

A glimpse of the future

MINTS– a Modal INtuitionistic Type theory with Stages

Kripke-Style Martin-Löf type theory that supports homogeneous multi-staged programming in the spirit of Scheme or Racket's quote-unquote style.

supports large elimination and a full cumulative universe hierarchy.

MINTS– a Modal INtuitionistic Type theory with Stages

Kripke-Style Martin-Löf type theory that supports homogeneous multi-staged programming in the spirit of Scheme or Racket's quote-unquote style.

supports large elimination and a full cumulative universe hierarchy.

- Extends Pfenning and Davies' work to dependent types
- Generate and share code across multiple stages
- Specify strong guarantees of multi-staged program using dependent types (especially large eliminations)
- Reason about multi-staged programs and prove them correct

How do we get there . . .?

Kripke-style modal lambda-calculus – simply typed!

$$\underbrace{\Gamma_0; \ldots ; \Gamma_n}_{\text{Kripke Context Stack } \overrightarrow{\Gamma}} \Vdash \underbrace{t}_{\text{Term}} : \underbrace{\tau}_{\text{Type}}$$

- Each Kripke world of modal logic corresponds to a stage in the computation
- A term of type $\Box\tau$ corresponds to the code of a program of type τ in a future stage of computation

Kripke-style modal lambda-calculus – simply typed!

Local Variable

$$\frac{x{:}\tau \in \Gamma_n}{\Gamma_0; \ldots; \Gamma_n \Vdash x : \tau}$$

Kripke-style modal lambda-calculus – simply typed!

Local Variable

$$\frac{x{:}\tau \in \Gamma_n}{\Gamma_0; \ldots; \Gamma_n \Vdash x : \tau}$$

Box Introduction (push context onto context stack)

$$\frac{\vec{\Gamma}; \Gamma; \cdot \Vdash t : \tau}{\vec{\Gamma}; \Gamma \Vdash \mathrm{box}\ t : \Box\tau}$$

Kripke-style modal lambda-calculus – simply typed!

Local Variable

$$\frac{x{:}\tau \in \Gamma_n}{\Gamma_0; \ldots; \Gamma_n \Vdash x : \tau}$$

Box Introduction (push context onto context stack)

$$\frac{\vec{\Gamma}; \Gamma; \cdot \Vdash t : \tau}{\vec{\Gamma}; \Gamma \Vdash \text{box } t : \Box\tau}$$

Unbox Elimination (pop context(s) of context stack)

$$\frac{\vec{\Gamma}; \Gamma_0 \Vdash t : \Box\tau}{\vec{\Gamma}; \Gamma_0; \Gamma_1; \ldots; \Gamma_n \Vdash \text{unbox}_n\ t : \tau}$$

The modal offset n corresponds to reflexivity and transitivity of the accessibility relation between worlds in the Kripke semantics.

Local structural properties and modal transformations

Local structural properties of a context Γ_i

- Weakening of a context Γ_i
- Substitution for assumptions in a context Γ_i

$$\dfrac{\dfrac{\overrightarrow{\Gamma}; \Gamma_0, x{:}\tau' \Vdash t : \tau}{\overrightarrow{\Gamma}; \Gamma_0 \Vdash \lambda x.t : \tau' \to \tau} \quad \overrightarrow{\Gamma}; \Gamma_0 \Vdash t' : \tau'}{\overrightarrow{\Gamma}; \Gamma_0 \Vdash (\lambda x.t)\ t' : \tau} \quad \Longrightarrow \quad \overrightarrow{\Gamma}; \Gamma_0 \Vdash [t'/x]t : \tau$$

Local structural properties and modal transformations

Local structural properties of a context Γ_i

- Weakening of a context Γ_i
- Substitution for assumptions in a context Γ_i

Modal transformations (MoTs) between context stacks

- Modal weakening and fusion of context stack $\overrightarrow{\Gamma}; \Gamma_0$

$$\dfrac{\dfrac{\overrightarrow{\Gamma}; \Gamma_0; \cdot \Vdash t : \tau}{\overrightarrow{\Gamma}; \Gamma_0 \Vdash \text{box } t : \Box\tau}}{\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; \ldots; \Gamma_n \Vdash \text{unbox}_n \ (\text{box } t) : \tau} \quad \Longrightarrow \quad \overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; \ldots; \Gamma_n \Vdash t' : \tau$$

where t' is obtained from t by modal weakening / fusion.

Local structural properties and modal transformations

Local structural properties of a context Γ_i

- Weakening of a context Γ_i
- Substitution for assumptions in a context Γ_i

Modal transformations (Mo

- Modal

$\overrightarrow{\Gamma}$; ... $\Gamma_1; \ldots; \Gamma_n \Vdash t' : \tau$

where t' is c ... akening / fusion.

Typically, local context and modal context stack transformations are thought of as separate concepts.

This is problematic, since simultaneous substitutions are central in

- Implementations based on explicit substitution
- Normalization proofs using logical relations
- Mechanizations based on de Bruijn

Towards MINTS by examples

Example: Staged Power Function

```
pow : Nat → □ (Nat → Nat)
pow zero     = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

Example: Staged Power Function

```
pow : Nat → □ (Nat → Nat)
pow zero     = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

Typically, code generation does not evaluate code inside a box.

```
pow 2 = box λ x → ((unbox₁ (pow 1)) x) * x
      = box λ x → ((unbox₁ (box λ y → ((unbox₁ (pow 0)) y) * y)) x) * x
      = box λ x → ((λ y → ((unbox₁ (pow 0)) y) * y) x) * x
      = box λ x → ((λ y → ((unbox₁ (box λ z → 1)) y) * y) x) * x
      = box λ x → ((λ y → ((λ z → 1) y) * y) x) * x
```

Example: Staged Power Function

```
pow : Nat → □ (Nat → Nat)
pow zero      = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

Typically, code generation does not evaluate code inside a box.

```
pow 2 = box λ x → ((unbox₁ (pow 1)) x) * x
      = box λ x → ((unbox₁ (box λ y → ((unbox₁ (pow 0)) y) * y)) x) * x
      = box λ x → ((λ y → ((unbox₁ (pow 0)) y) * y) x) * x
      = box λ x → ((λ y → ((unbox₁ (box λ z → 1)) y) * y) x) * x
      = box λ x → ((λ y → ((λ z → 1) y) * y) x) * x
```

What code shall we generate in a type theory?

Example: Staged Power Function

```
pow : Nat → □ (Nat → Nat)
pow zero     = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

Typically, code generation does not evaluate code inside a box.

```
pow 2 = box λ x → ((unbox₁ (pow 1)) x) * x
      = box λ x → ((unbox₁ (box λ y → ((unbox₁ (pow 0)) y) * y)) x) * x
      = box λ x → ((λ y → ((unbox₁ (pow 0)) y) * y) x) * x
      = box λ x → ((λ y → ((unbox₁ (box λ z → 1)) y) * y) x) * x
      = box λ x → ((λ y → ((λ z → 1) y) * y) x) * x
```

What code shall we generate in a type theory? Evaluation = Normalization!

Example: Staged Power Function

```
pow : Nat → □ (Nat → Nat)
pow zero     = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

Typically, code generation does not evaluate code inside a box.

```
pow 2 = box λ x → ((unbox₁ (pow 1)) x) * x
      = box λ x → ((unbox₁ (box λ y → ((unbox₁ (pow 0)) y) * y)) x) * x
      = box λ x → ((λ y → ((unbox₁ (pow 0)) y) * y) x) * x
      = box λ x → ((λ y → ((unbox₁ (box λ z → 1)) y) * y) x) * x
      = box λ x → ((λ y → ((λ z → 1) y) * y) x) * x
```

What code shall we generate in a type theory? Evaluation = Normalization!

We typically identify terms up to $\beta\eta$ (and for example normalize under a λ-abstraction.)
$\Longrightarrow$ **Consequently we normalize also under a box!**

Example: Staged Power Function

```
pow : Nat → □ (Nat → Nat)
pow zero     = box λ x → 1
pow (succ n) = box λ x → ((unbox₁ (pow n)) x) * x
```

Evaluation by Normalization :

```
pow 2 = box λ x → ((unbox₁ (pow 1)) x) * x
      = box λ x → ((unbox₁ (box λ y → ((unbox₁ (pow 0)) y) * y)) x) * x
      = box λ x → ((λ y → ((unbox₁ (pow 0)) y) * y) x) * x
      = box λ x → (((unbox₁ (pow 0)) x) * x) * x
      = box λ x → (((λ z → 1) x) * x) * x
      = box λ x → x * x
```

Avoids administrative redeces!

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e.
  `box λ x y → x + y`.
- etc.

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e. `box λ x y → x + y`.
- etc.

Step 1: Type-level function `nary n` which computes the type of an `n`-ary function:

```
nary : Nat → Se
nary zero     = Nat
nary (succ n) = Nat → nary n
```

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e. `box λ x y → x + y`.
- etc.

Step 1: Type-level function `nary n` which computes the type of an `n`-ary function:

```
nary : Nat → Se
nary zero     = Nat
nary (succ n) = Nat → nary n
```

Step 2: Define `n-ary-sum :(n : Nat) → □ (nary n)`

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e. `box λ x y → x + y`.
- etc.

Step 1: Type-level function `nary n` which computes the type of an `n`-ary function:

```
nary : Nat → Se
nary zero     = Nat
nary (succ n) = Nat → nary n
```

Step 2: Define `n-ary-sum : (n : Nat) → □ (nary (unbox₁ (lift n)))`

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e.
  `box λ x y → x + y`.
- etc.

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e. `box λ x y → x + y`.
- etc.

Step 2: Implementation `n-ary-sum`

```
nary-sum : (n : Nat) → □ (nary (unbox₁ (lift n)))
nary-sum zero              = box zero
nary-sum (succ zero)       = box λ x → x
nary-sum (succ (succ n))   = box λ x y → (unbox₁ (nary-sum (succ n))) (x + y)
```

Example: Generating N-ary Sum Function

Idea:

- If n is zero, then we return `zero`;
- If n is one, then we return the identity function;
- If n is two, then we return the function that sums up two arguments, i.e. `box λ x y → x + y`.
- etc.

Step 2: Implementation `n-ary-sum`

```
nary-sum : (n : Nat) → □ (nary (unbox₁ (lift n)))
nary-sum zero              = box zero
nary-sum (succ zero)       = box λ x → x
nary-sum (succ (succ n))  = box λ x y → (unbox₁ (nary-sum (succ n))) (x + y)
```

Example: Checking that implementation works as intended

Step 3: Test that `n-ary-sum` works as intended.

```
nary-sum-3 : nary-sum 3 ≡ box λ x y z → (x + y) + z
nary-sum-3 = refl
```

Step 4: Prove more general soundness theorems

> Summing over a list `l` of `n` natural numbers returns the same result as generating code using `nary-sum n` and then applying it to all the numbers in `l`.

A glimpse of MINTS:
A Kripke-Style Modal Type Theory

Kripke-style Explicit Formulation: box - unbox (REVISITED)

$$\underbrace{\Gamma_0; \ldots; \Gamma_n}_{\text{Kripke Context Stack } \vec{\Gamma}} \quad \Vdash \quad \underbrace{t}_{\text{Term}} \quad : \quad \underbrace{\tau}_{\text{Type}}$$

Local Variable

$$\frac{x{:}\tau \in \Gamma_n}{\Gamma_0; \ldots; \Gamma_n \Vdash x : \tau}$$

Kripke-style Explicit Formulation: box - unbox (REVISITED)

$$\underbrace{\Gamma_0; \ldots ; \Gamma_n}_{\text{Kripke Context Stack } \vec{\Gamma}} \quad \Vdash \quad \underbrace{t}_{\text{Term}} \quad : \quad \underbrace{\tau}_{\text{Type}}$$

Local Variable

$$\frac{x{:}\tau \in \Gamma_n}{\Gamma_0; \ldots ; \Gamma_n \Vdash x : \tau}$$

The Problem in the Dependently Typed Setting:

We have dependencies within one context and across the context stack!

Kripke-style Explicit Formulation: box - unbox (REVISITED)

$$\underbrace{\Gamma_0; \ldots; \Gamma_n}_{\text{Kripke Context Stack } \vec{\Gamma}} \quad \Vdash \quad \underbrace{\mathrm{t}}_{\text{Term}} \quad : \quad \underbrace{\tau}_{\text{Type}}$$

Local Variable

$$\frac{\Gamma_n = \Gamma, x{:}\tau, \Gamma'}{\Gamma_0; \ldots; \Gamma_n \Vdash x : [\mathrm{wk}]\tau}$$

The Problem in the Dependently Typed Setting:

We have dependencies within one context and across the context stack!

Solution: Ordinary weakening wrt to a context

Revisiting Box Elimination

Unbox Elimination (pop context(s) of context stack)

$$\frac{\overrightarrow{\Gamma}; \Gamma_0 \Vdash t : \Box\tau}{\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1 \ldots ; \Gamma_n \Vdash \mathsf{unbox}_n\ t : \tau}$$

Revisiting Box Elimination

Unbox Elimination (pop context(s) of context stack)

$$\frac{\overrightarrow{\Gamma}; \Gamma_0 \Vdash t : \Box\tau}{\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1 \ldots; \Gamma_n \Vdash \mathsf{unbox}_n \ t : \tau}$$

The Problem in the Dependently Typed Setting:

We have dependencies within one context and across the context stack!

Revisiting Box Elimination

Unbox Elimination (pop context(s) of context stack)

$$\frac{\vec{\Gamma}; \Gamma_0 \Vdash t : \Box\tau}{\vec{\Gamma}; \Gamma_0; \Gamma_1 \ldots; \Gamma_n \Vdash \mathsf{unbox}_n\ t : \tau}$$

The Problem in the Dependently Typed Setting:

For the premise we have:

- $\vec{\Gamma}; \Gamma_0; \cdot \Vdash \tau : \mathsf{Se}$

In the conclusion we need:

- $\vec{\Gamma}; \Gamma_0; \Gamma_1 \ldots; \Gamma_n \Vdash \tau : \mathsf{Se}$

Required: Weaken with Γ_1 and Modal Weakening wrt context stack $\Gamma_2; \ldots \Gamma_n$

A general transformation between context stacks

Unified (Simultaneous) Substitutions

Local Context	Γ, Δ	$:= \cdot \mid \Gamma, x{:}\tau$	Local Substitution	σ	$:= \cdot \mid \sigma, t/x$
Context Stack	$\vec{\Gamma}, \vec{\Delta}$	$:= \epsilon; \Gamma \mid \vec{\Gamma}; \Gamma$	Substitution Stack	$\vec{\sigma}$	$:= \varepsilon; \sigma \mid \vec{\sigma}; \Uparrow^n \sigma$

$$\frac{\vec{\Gamma} \Vdash \vec{\sigma} : \vec{\Delta} \qquad \vec{\Gamma}; \Gamma_1; \ldots; \Gamma_n \Vdash \sigma : \Delta}{\vec{\Gamma}; \Gamma_1; \ldots; \Gamma_n \Vdash \vec{\sigma}; \Uparrow^n \sigma : \vec{\Delta}; \Delta}$$

Unified Substitution Operation

$$
\begin{array}{lll}
[\overrightarrow{\sigma}; \Uparrow^k \sigma]x & := \sigma(x) & \text{lookup } x \text{ in } \sigma \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](\lambda x.t) & := \lambda x.[\overrightarrow{\sigma}; \Uparrow^k (\sigma, x/x)]t & \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](s\ \ t) & := [\overrightarrow{\sigma}; \Uparrow^k \sigma]s\ \ [\overrightarrow{\sigma}; \Uparrow^k \sigma]t & \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](\texttt{box}\ \ t) & := \texttt{box}\ [\overrightarrow{\sigma}; \Uparrow^k \sigma; \Uparrow^1 \cdot]t & \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](\texttt{unbox}_n\ \ t) & :=? &
\end{array}
$$

Unified Substitution Operation

$$
\begin{aligned}
[\overrightarrow{\sigma}; \Uparrow^k \sigma]x &:= \sigma(x) && \text{lookup } x \text{ in } \sigma \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](\lambda x.t) &:= \lambda x.[\overrightarrow{\sigma}; \Uparrow^k (\sigma, x/x)]t \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](s\ \ t) &:= [\overrightarrow{\sigma}; \Uparrow^k \sigma]s\ \ [\overrightarrow{\sigma}; \Uparrow^k \sigma]t \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](\texttt{box}\ \ t) &:= \texttt{box}\ [\overrightarrow{\sigma}; \Uparrow^k \sigma; \Uparrow^1 \cdot]t \\
[\overrightarrow{\sigma}; \Uparrow^k \sigma](\texttt{unbox}_n\ \ t) &:= ?
\end{aligned}
$$

Recall – **Unbox Elimination** (pop context(s) of context stack)

$$
\frac{\overrightarrow{\Gamma}; \Gamma_0 \Vdash t : \Box\tau}{\overrightarrow{\Gamma}; \Gamma_0; \ldots; \Gamma_n \Vdash \mathsf{unbox}_n\ t : \tau}
$$

$\Longrightarrow$ truncate the unified substitution $[\overrightarrow{\sigma}; \Uparrow^k \sigma]$ to apply it to t on the rhs

Truncation and Truncation Offset

Typing of unified substitution: $\overrightarrow{\Gamma} \Vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$**.**

Truncation : $\overrightarrow{\sigma} \mid n$

- Returns a prefix of $\overrightarrow{\sigma}$ with domain context stack $\overrightarrow{\Delta} \mid n$

Truncation Offset $\mathcal{O}(\overrightarrow{\sigma}, n) = k$

- Sums over all the modal offsets in the the truncated part of $\overrightarrow{\sigma}$
- Used to adjust the range $\overrightarrow{\Gamma}$ s.t. $\overrightarrow{\sigma} \mid n$ remains well-typed.

Truncation and Truncation Offset

Typing of unified substitution: $\overrightarrow{\Gamma} \Vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$**.**

Truncation : $\overrightarrow{\sigma} \mid n$

- Returns a prefix of $\overrightarrow{\sigma}$ with domain context stack $\overrightarrow{\Delta} \mid n$

Truncation Offset $\mathcal{O}(\overrightarrow{\sigma}, n) = k$

- Sums over all the modal offsets in the the truncated part of $\overrightarrow{\sigma}$
- Used to adjust the range $\overrightarrow{\Gamma}$ s.t. $\overrightarrow{\sigma} \mid n$ remains well-typed.

Typing of truncated substitution: $(\overrightarrow{\Gamma} \mid k) \Vdash (\overrightarrow{\sigma} \mid n) : (\overrightarrow{\Delta} \mid n)$

Revisiting Unbox

Typing

$$\frac{\overrightarrow{\Gamma}; \Gamma_0 \Vdash t : \Box\tau \quad \overrightarrow{\Gamma}; \Gamma_0; \cdot \Vdash \tau : \mathtt{Se}_i \quad \Vdash \overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; \ldots; \Gamma_n}{\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; \ldots; \Gamma_n \Vdash \mathtt{unbox}_n\ t : [\overrightarrow{I}; \Uparrow^n \cdot]\tau}$$

Applying unified substitution

$$[\overrightarrow{\sigma}; \Uparrow^k \sigma](\mathtt{unbox}_n\ t) \quad := \mathtt{unbox}_{n'}\ [\overrightarrow{\sigma}']t$$

where $\underbrace{\overrightarrow{\sigma}' = (\overrightarrow{\sigma}; \Uparrow^k \sigma) \mid n}_{\text{Truncate } \overrightarrow{\sigma}; \Uparrow^k \sigma}$ and $\underbrace{n' = \mathcal{O}((\overrightarrow{\sigma}; \Uparrow^k \sigma)\ ,\ n)}_{\text{Compute the truncation offset}}$

Unified Substitutions are key.

Point 1: Unified Substitutions enable normalization for MINTS

Normalization by Evaluation algorithm for MINTS following Abel'13

- Algebraic uniform characterization of the Kripke-structure based on unified substitutions (syntax) using untyped modal transformation (semantics)
- A core modal dependent type theory as an explicit substitution calculus
  $\longrightarrow$ our NbE algorithm applies to all 4 subsystems of S4
- Completeness proof for the NbE algorithm uses a partial equivalence relation (PER) model for the untyped domain terms together with untyped modal transformations.
- Soundness proof for the NbE algorithm based on a Kripke glueing model
  Exploits a special class of unified substitutions, restricted to weakenings.
- NbE soundness and completeness mechanized in Agda (11K) only exploits function extensionality and induction-recursion
- Mechanization exposes common oversimplification in how cumulativity of universes

Point 2: Contextual Box and Unbox to Handle Open Code

Point 2: Contextual Box and Unbox to Handle Open Code

$$
\begin{array}{lll}
\text{Types} & \tau & := \ldots \mid \texttt{box}\ (\overrightarrow{\Delta} \vdash \tau) \\
\text{Terms} & t & := \ldots \mid \texttt{box}\ (\overrightarrow{\Delta} \vdash t) \mid \texttt{unbox}\ (t\ ,\ \overrightarrow{\sigma})\ \ \text{where } \overrightarrow{\sigma} \text{ is a partial unified substitution.}
\end{array}
$$

Point 2: Contextual Box and Unbox to Handle Open Code

Types $\quad \tau \quad ::= \ldots \mid \texttt{box}\ (\overrightarrow{\Delta} \vdash \tau)$

Terms $\quad t \quad ::= \ldots \mid \texttt{box}\ (\overrightarrow{\Delta} \vdash t) \mid \texttt{unbox}\ (t\ ,\ \overrightarrow{\sigma})$ where $\overrightarrow{\sigma}$ is a partial unified substitution.

Introduction

$$\frac{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \Vdash t : \tau}{\overrightarrow{\Gamma} \Vdash \texttt{box}\ (\overrightarrow{\Delta} \vdash t)\ :\ \texttt{box}\ (\overrightarrow{\Delta} \vdash \tau)}$$

Point 2: Contextual Box and Unbox to Handle Open Code

Types $\tau \quad := \ldots \mid \texttt{box}\ (\overrightarrow{\Delta} \vdash \tau)$

Terms $t \quad := \ldots \mid \texttt{box}\ (\overrightarrow{\Delta} \vdash t) \mid \texttt{unbox}\ (t\ ,\ \vec{\sigma})$ where $\vec{\sigma}$ is a partial unified substitution.

Introduction

$$\frac{\overrightarrow{\Gamma};\overrightarrow{\Delta} \Vdash t : \tau}{\overrightarrow{\Gamma} \Vdash \texttt{box}\ (\overrightarrow{\Delta} \vdash t)\ :\ \texttt{box}\ (\overrightarrow{\Delta} \vdash \tau)}$$

Elimination

$$\frac{\overrightarrow{\Gamma} \mid n \Vdash t : \texttt{box}\ (\overrightarrow{\Delta} \vdash \tau) \qquad \overrightarrow{\Gamma} \Vdash \vec{\sigma} :: \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \Vdash \texttt{unbox}\ (t\ ,\ \vec{\sigma}) : [\overrightarrow{I};\vec{\sigma}]\tau} \text{ where } \mathcal{O}(\vec{\sigma}) = n$$

- **Previously:** The modal offset at the $\texttt{unbox}$ only allowed for modal weakening
- **Now:** Unified substitution allow for modal weakening and instantiation of the variables $\overrightarrow{\Delta}$

Current Status

- **MINTS: Kripke-style modal type theory (joint work with J. Z. Hu and J. Jang)**
- Categorical view of modal lambda-calculi (joint work with J. Z. Hu) [MFPS'22]
- System F-style meta-programming with pattern matching (joint work with J. Jang, S. Gélineau, S. Monnier) [POPL'22]

What I Learned

Lesson 1: Logic through the Curry-Howard isomorphism allows us to gain a deeper understanding of computational phenomena.

Lesson 2: While other approaches exist to support type-safe generation of typed code, they are not logically motivated.

Lesson 3: Unified substitutions provide a general concept to capture transformation between context stacks both syntactically and semantically.

Lesson 4: Good first step towards a dependently typed foundation for meta-programming!

What's next? – How to support pattern matching in MINTS.