A Type-Theoretic Framework for Meta-Programming

Lessons Learned from Writing Meta-Theoretic Proofs as Programs

Brigitte Pientka

McGill University
Montreal, Canada





McGill

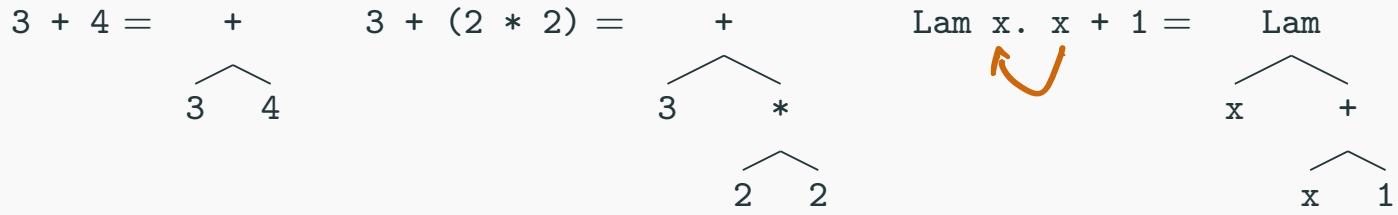
joint work with
Francisco Ferreira, Antoine Gaulin, Jason Z. Hu, Junyoung Jang, Chuta Sano, David Thibodeau

What Is Meta-Programming?

What is meta-programming?

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

- *(Quasi)quotation* of **box** `(3 + 4)` represents a **closed** abstract syntax tree (AST)



- *Unquote* splices in code fragments.

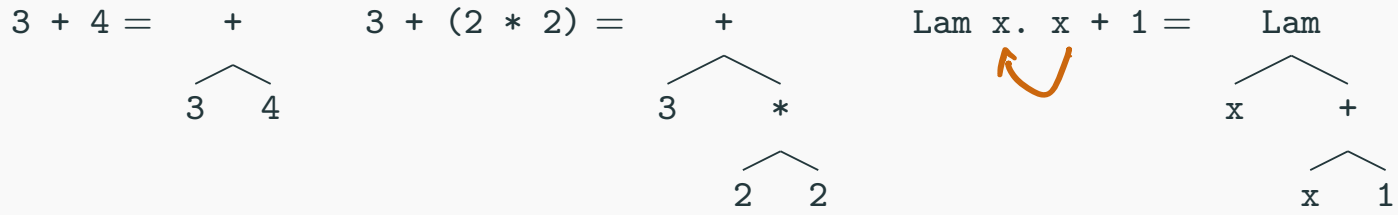
  Let `code_gen 2` $\Longrightarrow$ **box** `(2 * 2)`.

  box`(3 +` **unbox**`(code_gen 2))` $\Longrightarrow$ **box**`(3 + (2 * 2))` $\neq$ **box**`(3 + 4)`

What is meta-programming?

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

- *(Quasi)quotation* of **box** `(3 + 4)` represents a **closed** abstract syntax tree (AST)



- *Unquote* splices in code fragments.

  Let `code_gen 2` $\Longrightarrow$ **box** `(2 * 2)`.

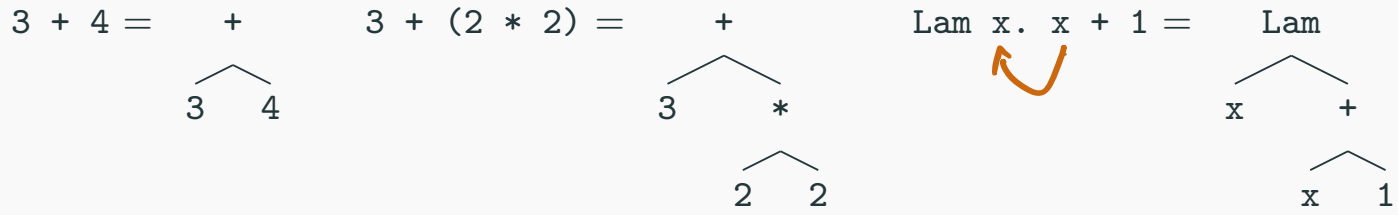
  box`(3 +` **unbox**`(code_gen 2))` $\Longrightarrow$ **box**`(3 + (2 * 2))` $\neq$ **box**`(3 + 4)`

- *Evaluate* code: `eval` **box**`(3 + (2 * 2))` $\Longrightarrow$ 7

What is meta-programming?

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

- *(Quasi)quotation* of **box** `(3 + 4)` represents a **closed** abstract syntax tree (AST)



- *Unquote* splices in code fragments.
  Let `code_gen 2` $\Longrightarrow$ **box** `(2 * 2)`.

  box`(3 +` **unbox**`(code_gen 2))` $\Longrightarrow$ **box**`(3 + (2 * 2))` $\neq$ **box**`(3 + 4)`

- *Evaluate* code: `eval` **box**`(3 + (2 * 2))` $\Longrightarrow$ 7
- *Transform* code from **box**`(lam x. x + (x * 0))` to **box** `(lam x. x)`

Good and Not so Good of Meta-Programming

The Good:

- Widely used and supported
  (Lisp, Scheme, TemplateHaskell, MetaML, TemplateCoq / MetaCoq, Lean, ...)
- Eliminate boilerplate code (programming/proof perspective)
- Proofs using reflection (proof perspective)

Not so Good:

- Hard to test and reason about code generation and generated code
- Errors are caught during run-time instead of generation time
- **Quote is external to the type-theoretical foundation**

This Talk

A Type-Theoretic Framework for Type-Safe Meta-Programming

- Catch errors early at compile-time / generation time
- Reason about code generation and generated code
- Provide a proper logical foundation
- Support meta-programming over a range of sub-languages
- Derive general principles of designing meta-programming frameworks

How to develop a good foundation for type-safe meta-programming?

Let's start at the beginning...If we want to write code transformations, we must represent code.

Representing Types and Terms – A Prototypical Example

Types $A, B ::= \mathsf{nat} \mid A \Rightarrow B$

Terms $M ::= x \mid \mathsf{lam}\ x{:}A.M \mid \mathsf{app}\ M\ N$



Representing Types and Terms – A Prototypical Example

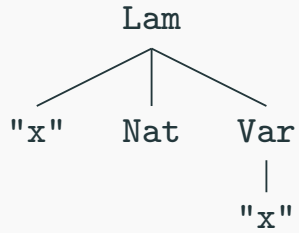
Types $A, B ::= \text{nat} \mid A \Rightarrow B$ Terms $M ::= x \mid \text{lam}\ x{:}A.M \mid \text{app}\ M\ N$

OCaml

```
1 type obj = Nat | Arr of obj * obj
2 type tm  = Var of string
3          | Lam of string * obj * tm
4          | App of tm * tm
```

On Paper (Object Language)	In OCaml (Meta Language)
lam x:nat.x (Identity)	`Lam ("x", Nat, Var "x")`
lam x:nat. lam x:nat⇒nat.x	`Lam ("x", Nat,` `  Lam("x", (arr nat nat), Var "x"))`

lam x:nat. lam f:nat⇒nat.app f x	`Lam("x", Nat,`
	`  Lam("f", (arr nat nat),`
	`  App (Var "f" , Var "x")))`



Representing Types and Terms – A Prototypical Example

Typ… $\to B$ Terms $M ::= x \mid \text{lam } x{:}A.M \mid \text{app } M\ N$

OC…

- There is a zoo of possible representation techniques
  deBruijn, Locally Nameless, Nominal, ...
- Usually requires explicit renaming and substitution operations
- Requires lemmas about substitution, etc.

Boilerplate code

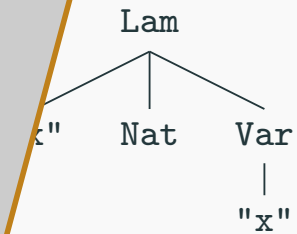
On Paper (Object…

lam x:nat.x (Identity)

lam x:nat. lam x:nat⇒nat.x

lam x:nat. lam f:nat⇒nat.app f x

```
La…
      Lam("x…
Lam("x", Nat,
      Lam("f", (arr nat nat),
            App (Var "f" , Var "x")))
```



The situation Is tedious

Over 70% of the proofs are tedious infrastructure proofs . . .

> In the end, out of a total of around 550 lemmas, approximately 400 were tedious "infrastructure" lemmas; only the remainder had direct relevance to the meta-theory of F_ω or elaboration. The number of required infrastructure lemmas appears to be quadratic in the number of variable classes (type and value variables for us), the number of "substitution" operations needed per class (we got away with only using LN's `subst` and `open`, and avoiding `close`) and the arity classes (unary and n-ary) of binding constructs. So we cannot, hand-on-heart, recommend the vanilla LN style for anything but small, kernel language developments. It would, however, be interesting to see whether more recent

A. Rossberg, C. Russo, D. Dreyer. F-ing Modules. JFP, 2014

Abstraction, Abstraction, Abstraction

"The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal." *B. Liskov [1974]*



“To know your future you must know your past.” – G. Santayana

Back in the 80s...

Back in the 80s...

1987 — *R. Harper, F. Honsell, G. Plotkin:* A Framework for Defining Logics, LICS'87

1988 — *F. Pfenning and C. Elliott*: Higher-Order Abstract Syntax, PLDI'88

- **LF = Dependently Typed Lambda Calculus (λ^{Π})** serves as a Meta-Language for representing formal systems
- **Higher-order Abstract Syntax (HOAS)** :
  Uniformly model binding structures in Object Language with (intensional) functions in LF

Representing Types and Terms in LF – In a Nutshell

Types $A, B ::= \mathsf{nat} \mid A \Rightarrow B$

Terms $M ::= x \mid \mathsf{lam}\ x{:}A.M \mid \mathsf{app}\ M\ N$



Representing Types and Terms in LF – In a Nutshell

Types $A, B ::= \text{nat} \mid A \Rightarrow B$ Terms $M ::= x \mid \text{lam}\ x{:}A.M \mid \text{app}\ M\ N$



LF Representation

```
obj: type.                    tm: type.
nat: obj.                     lam: obj → (tm → tm) → tm.
arr: obj → obj  → obj.        app: tm → tm  → tm.
```

On Paper (Object Language)	In LF (Meta Language)
lam x:nat.x	`lam nat λx.x`
lam x:nat. (lam x:nat⇒nat.x)	`lam nat λx.(lam (arr nat nat) λx.x)`
lam x:nat. (lam f:nat⇒nat.app f x)	`lam nat λx.(lam (arr nat nat) λf.app f x)`

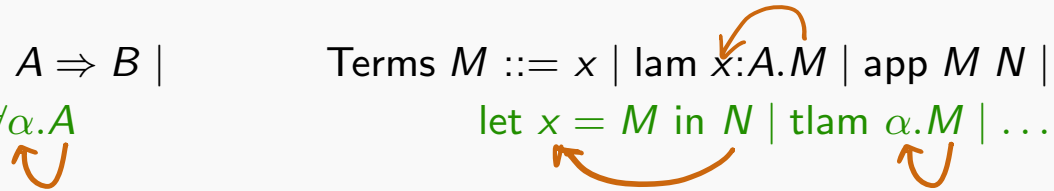
Higher-order Abstract Syntax (HOAS):

- Uniformly model bindings with (intensional) functions in LF
- Inherit α-renaming and single substitutions

Uniformly Model Binding Structures Using LF Functions

Types $A, B ::= \mathsf{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha.A$

Terms $M ::= x \mid \mathsf{lam}\ x{:}A.M \mid \mathsf{app}\ M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid \mathsf{tlam}\ \alpha.M \mid \ldots$



Uniformly Model Binding Structures Using LF Functions

Types $A, B ::= \text{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha.A$

Terms $M ::= x \mid \text{lam } x{:}A.M \mid \text{app } M\ N \mid \text{let } x = M \text{ in } N \mid \text{tlam } \alpha.M \mid \ldots$

LF Representation

```
obj: type.
nat: obj.
arr: obj → obj  → obj.
all: (obj → obj) → obj.
```

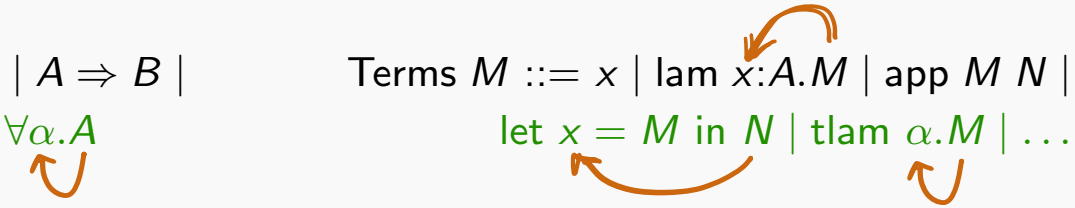
```
tm: type.
lam: obj → (tm → tm) → tm.
app: tm → tm  → tm.
let: tm → (tm → tm) → tm.
tlam: (obj → tm) → tm.
```

On Paper (Object Language)	In LF (Meta Language)
tlam α. (lam $x{:}\alpha.x$)	`tlam λa.(lam a λx.x)`
$\forall\alpha.\forall\beta.\alpha \Rightarrow \beta$	`all λa.all λb.arr a b`

Uniformly Model Typed Binding Structures Using LF Functions

Types $A, B ::= \mathsf{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha.A$

Terms $M ::= x \mid \mathsf{lam}\ x{:}A.M \mid \mathsf{app}\ M\ N \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid \mathsf{tlam}\ \alpha.M \mid \ldots$



Uniformly Model Typed Binding Structures Using LF Functions

Types $A, B ::= \text{nat} \mid A \Rightarrow B \mid \alpha \mid \forall\alpha.A$

Terms $M ::= x \mid \text{lam } x{:}A.M \mid \text{app } M\ N \mid \text{let } x = M \text{ in } N \mid \text{tlam } \alpha.M \mid \ldots$

LF Representation

```
obj: type.
nat: obj.
arr: obj → obj → obj.
all: (obj → obj) → obj.
```

```
tm:  obj → type.
lam: {A:obj} → (tm A → tm B) → tm (arr A B).
app: tm (arr A B) → tm A → tm B.
let: tm A → (tm A → tm B) → tm B.
tlam: ({a:obj} → tm B a) → tm (all B).
```

On Paper (Object Language)	In LF (Meta Language)	
tlam α. (lam $x{:}\alpha.x$)	`tlam λa.(lam a λx.x)`	`tm (all λa. arr a a)`
$\forall\alpha.\forall\beta.\alpha \Rightarrow \beta$	`all λa.all λb.arr a b`	`obj`

Uniformly Model Typed Binding Structures Using LF F

Types $A, B ::= \text{nat} \mid A \Rightarrow B \mid$

$\alpha \mid$

C

tl

$\forall\alpha$

(all λa. arr a a)

a b | obj

LF = Dependently Typed Lambda Calculus λ^{Π}

- **LF functions only encode variable scope**
  no recursion, no pattern matching, etc.
- **HOAS trees = Syntax trees with binders in normal form**
- **Benefit:** α-renaming and substitution principles
- **Scales:**
  - Intrinsically typed syntax trees with binders
  - Proof trees as HOAS trees

OK... so, how do we write recursive programs over with HOAS trees?

We clearly want pattern matching, since a HOAS tree is a data structure.

An Attempt to Compute the Size of a Term

$$
\begin{array}{lll}
 & \texttt{size} & (\texttt{lam}\ \lambda \texttt{x}.\texttt{lam}\ \lambda \texttt{f}.\ \texttt{app f x}) \\
\Longrightarrow & \texttt{size} & (\texttt{lam}\ \lambda \texttt{f}.\ \texttt{app f x}) + 1 \\
\Longrightarrow & \texttt{size} & (\texttt{app f x}) + 1 + 1 \\
\Longrightarrow & \texttt{size f} + \texttt{size x} + 1 + 1 + 1 & \\
\Longrightarrow & 0 + 0 + 1 + 1 + 1 &
\end{array}
$$

"the whole HOAS approach by its very nature disallows a feature that we regard of key practical importance: the ability to manipulate names of bound variables explicitly in computation and proof." *[Pitts, Gabbay'97]*

An Attempt to Compute the Size of a Term

$$
\begin{array}{lllllll}
 & \texttt{size} & (\texttt{lam}\ \lambda \texttt{x}.\texttt{lam}\ \lambda \texttt{f}.\ \texttt{app}\ \texttt{f}\ \texttt{x}) & & & & \\
\Longrightarrow & \texttt{size} & (\texttt{lam}\ \lambda \texttt{f}.\ \texttt{app}\ \texttt{f}\ \texttt{x}) & + 1 & & & \\
\Longrightarrow & \texttt{size} & (\texttt{app}\ \texttt{f}\ \texttt{x}) & + 1 + 1 & & & \\
\Longrightarrow & \texttt{size}\ \texttt{f} & + & \texttt{size}\ \texttt{x} & + 1 + 1 + 1 & & \\
\Longrightarrow & 0 & + & 0 & + 1 + 1 + 1 & &
\end{array}
$$

Contextual objects and types to the rescue!

Back in 2008...

LF and Holes in HOAS Trees – Revisited

In LF (Meta Lang.)

`lam` λ`x.` `lam` λ`f.app f x`

`lam` λ`x.` `lam` λ`f.` `app f x`

LF Typing Judgment:

$$\underset{\text{LF Context}}{\Psi} \vdash \underset{\text{LF Term}}{M} : \underset{\text{LF Type}}{A}$$

LF and Holes in HOAS Trees – Revisited

In LF (Meta Lang.)

$\texttt{lam}\ \ \lambda \texttt{x}.\boxed{\texttt{lam}\ \lambda \texttt{f}.\texttt{app}\ \texttt{f}\ \texttt{x}}$

$\texttt{lam}\ \ \lambda \texttt{x}.\ \texttt{lam}\ \lambda \texttt{f}.\boxed{\texttt{app}\ \texttt{f}\ \texttt{x}}$

LF Typing Judgment:

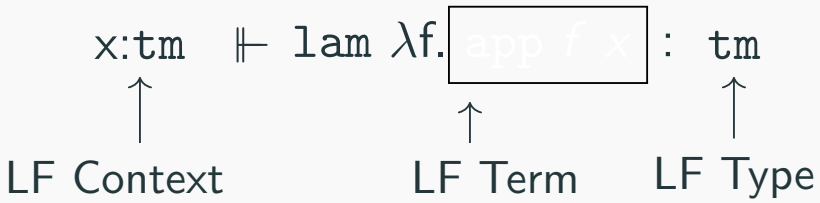
$$\texttt{x}{:}\texttt{tm} \vdash \texttt{lam}\ \lambda \texttt{f}.\texttt{app}\ \texttt{f}\ \texttt{x} : \texttt{tm}$$

LF Context — LF Term — LF Type

LF and Holes in HOAS Trees – Revisited

In LF (Meta Lang.)
lam λx. lam λf.app f x
lam λx. lam λf. app f x

LF Typing Judgment:



LF and Holes in HOAS Trees – Revisited

In LF (Meta Lang.)	Contextual Type
lam λx. $\boxed{\texttt{lam}\ \lambda\texttt{f}.\texttt{app f x}}$	$\lceil$x:tm $\vdash$ tm$\rceil$
lam λx. lam λf. $\boxed{\texttt{app f x}}$	$\lceil$x:tm, f:tm $\vdash$ tm$\rceil$

LF Typing Judgment:

$$\texttt{x:tm} \Vdash \texttt{lam}\ \lambda\texttt{f}.\boxed{\texttt{app f x}} : \texttt{tm}$$

LF Context — LF Term — LF Type

Question: What is the type of $\boxed{\texttt{app f x}}$?

LF and Holes in HOAS Trees – Revisited

In LF (Meta Lang.)	Contextual Type
lam λx. $\boxed{\texttt{lam}\ \lambda\texttt{f}.\texttt{app f x}}$	$\lceil$x:tm $\vdash$ tm$\rceil$
lam λx. lam λf. $\boxed{\texttt{app f x}}$	$\lceil$x:tm, f:tm $\vdash$ tm$\rceil$

LF Typing Judgment:

$$\texttt{x:tm} \Vdash \texttt{lam}\ \lambda\texttt{f}.\boxed{\texttt{app f x}} : \texttt{tm}$$

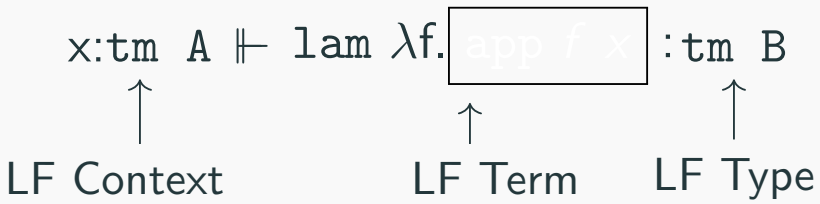
LF Context — LF Term — LF Type

Question: What is the type of $\boxed{\texttt{app f x}}$?

LF and Holes in Dependently Typed HOAS Trees

In LF (Meta Lang.)	Contextual Type
lam λx. lam λf. app f x	⌈x:tm A ⊢ tm B⌉
lam λx. lam λf. app f x	⌈x:tm A, f:tm (arr A B) ⊢ tm B⌉

LF Typing Judgment:



LF and Holes in Dependently Typed HOAS Trees

In LF (Me	Contextual Type
lam λx	⊢ tm B]
lam λx	⊢ tm B]

LF Typing

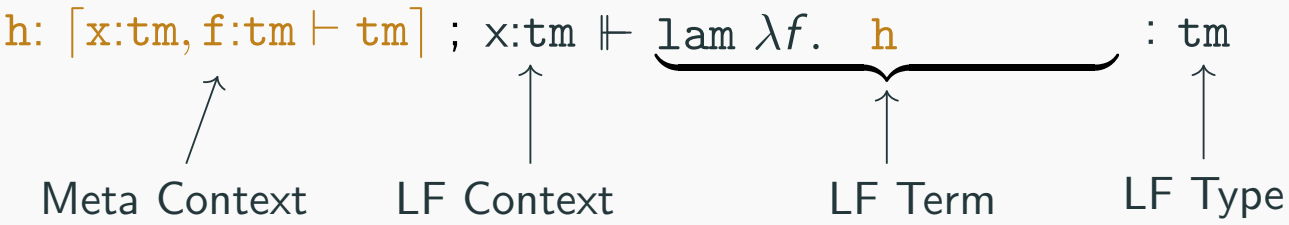
x:tm A ⊩ la

LF Context LF Term LF Type

Contextual Modal Type Theory [TOCL'08]
joint work with A. Nanevski and F. Pfenning

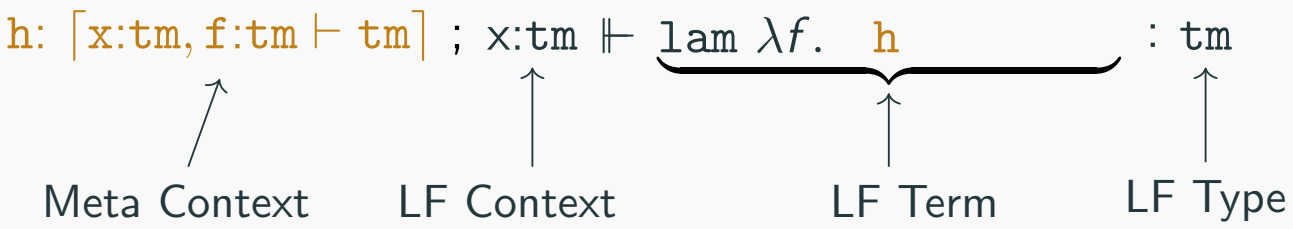
- Contextual types (⊢) reify LF typing derivations (⊩)
- Generalization of the closed box modality in S4

Contextual Type Theory [Nanevski, Pfenning, Pientka'08]



- h is a contextual variable
- It has the contextual type ⌈x:tm, f:tm ⊢ tm⌉
- It can be instantiated with a contextual term ⌈x, f ⊢ app f x⌉

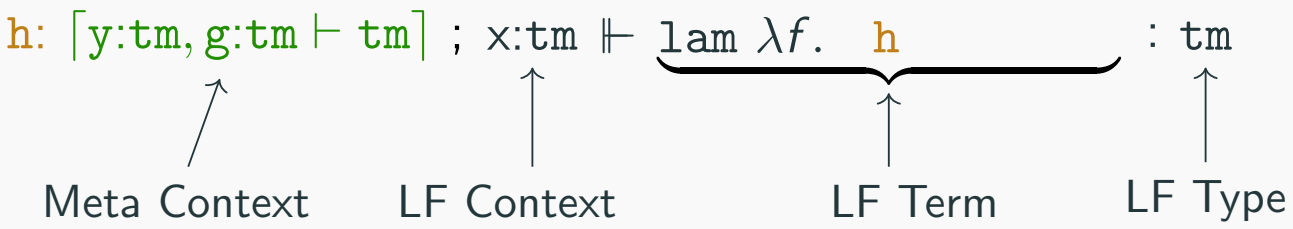
Contextual Type Theory [Nanevski, Pfenning, Pientka'08]



- h is a contextual variable
- It has the contextual type ⌈x:tm, f:tm ⊢ tm⌉
- It can be instantiated with a contextual term ⌈x, f ⊢ app f x⌉

WAIT! . . . whatever we plug in for h may contain free LF variables?

Contextual Type Theory [Nanevski, Pfenning, Pientka'08]



- h is a contextual variable
- It has the contextual type ⌈y:tm, g:tm ⊢ tm⌉
- It can be instantiated with a contextual term ⌈y, g ⊢ app g y⌉

WAIT! . . . whatever we plug in for h may contain free LF variables?
and we want it to be stable under α-renaming . . .

Contextual Type Theory [Nanevski, Pfenning, Pientka'08]

$$\texttt{h}: \lceil \texttt{y:tm}, \texttt{g:tm} \vdash \texttt{tm} \rceil \;;\; \texttt{x:tm} \Vdash \underbrace{\texttt{lam}\ \lambda f.\ \texttt{h[x/y, f/g]}}\ : \texttt{tm}$$

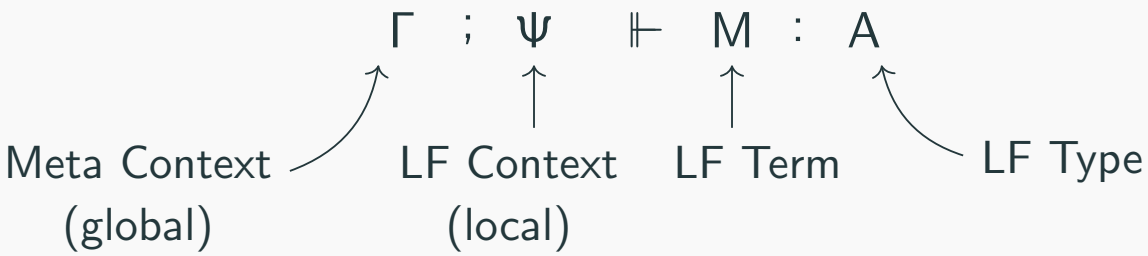
Meta Context — LF Context — LF Term — LF Type

- $\texttt{h}$ is a contextual variable
- It has the contextual type $\lceil \texttt{y:tm}, \texttt{g:tm} \vdash \texttt{tm} \rceil$
- It can be instantiated with a contextual term $\lceil \texttt{y}, \texttt{g} \vdash \texttt{app g y} \rceil$

WAIT! . . . whatever we plug in for $\texttt{h}$ may contain free LF variables?
and we want it to be stable under α-renaming . . .

Solution: Contextual variables are associated with LF substitutions

Contextual Type Theory[1] (CTT) [Nanevski, Pfenning, Pientka'08]



LF Variable

$$\frac{x{:}A \in \Psi}{\Gamma; \Psi \Vdash x : A}$$

Contextual Variable

$$\frac{x : \lceil \Phi \vdash A \rceil \in \Gamma \quad \Gamma; \Psi \Vdash \sigma : \Phi}{\Gamma; \Psi \Vdash \underbrace{x[\sigma]}_{\text{Closure}} : \underbrace{[\sigma]A}_{\text{Apply subst. } \sigma}}$$

Revisiting the Program `size`

$$
\begin{aligned}
& \texttt{size} \quad \lceil \vdash \texttt{lam}\ \lambda\texttt{x.lam}\ \lambda\texttt{f.}\ \texttt{app f x} \rceil \\
\Longrightarrow\ & \texttt{size} \quad \lceil \texttt{x} \vdash \texttt{lam}\ \lambda\texttt{f.}\ \texttt{app f x} \rceil + 1 \\
\Longrightarrow\ & \texttt{size} \quad \lceil \texttt{x,f} \vdash \texttt{app f x} \rceil + 1 + 1 \\
\Longrightarrow\ & \texttt{size}\ \lceil \texttt{x,f} \vdash \texttt{f} \rceil + \texttt{size}\ \lceil \texttt{x,f} \vdash \texttt{x} \rceil + 1 + 1 + 1 \\
\\
\Longrightarrow\ & 0 + 0 + 1 + 1 + 1
\end{aligned}
$$

Revisiting the Program `size`

$$
\begin{array}{ll}
 & \texttt{size}\quad \lceil \vdash \texttt{lam}\ \lambda\texttt{x.lam}\ \lambda\texttt{f. app f x} \rceil \\
\Longrightarrow & \texttt{size}\quad \lceil \texttt{x} \vdash \texttt{lam}\ \lambda\texttt{f. app f x} \rceil + 1 \\
\Longrightarrow & \texttt{size}\quad \lceil \texttt{x,f} \vdash \texttt{app f x} \rceil + 1 + 1 \\
\Longrightarrow & \texttt{size}\ \lceil \texttt{x,f} \vdash \texttt{f} \rceil + \texttt{size}\ \lceil \texttt{x,f} \vdash \texttt{x} \rceil + 1 + 1 + 1 \\
 & \\
\Longrightarrow & \quad 0 \quad + \quad 0 \quad + 1 + 1 + 1
\end{array}
$$

- Type of `size` $\Pi\gamma{:}\texttt{ctx}.\ \lceil \gamma \vdash \texttt{tm} \rceil \to \texttt{int}$
- Written by pattern matching on $\lceil \gamma \vdash \texttt{tm} \rceil$

```
size ⌈γ⌉ ⌈γ ⊢ lam λx. M⌉ = ….
size ⌈γ⌉ ⌈γ ⊢ app M N⌉ = ….
size ⌈γ⌉ ⌈γ ⊢ #p⌉ = …          % generic variable case
```

- Instead of ML-style pattern matching uses higher-order pattern matching [Miller'91]
- Possible because HOAS syntax trees are in normal form
- First-class context variables to abstract over the set of "free" variables

Revisiting the Program `size`

$$
\begin{aligned}
& \texttt{size}\ \ \lceil \vdash \texttt{lam}\ \ \lambda x.\texttt{lam}\ \ \lambda f.\ \texttt{app f x} \rceil \\
\Longrightarrow\ & \texttt{size}\ \ \lceil x \vdash \texttt{lam}\ \ \lambda f.\ \texttt{app f x} \rceil + 1 \\
\Longrightarrow\ & \texttt{size}\ \ \lceil x,f \vdash \texttt{app f x} \rceil + 1 \\
\Longrightarrow\ & \texttt{size}\ \lceil x,f \vdash f \rceil \ \ + \ \ \texttt{size}\ \lceil x,f \vdash x \rceil + 1 + 1 \\
\Longrightarrow\ & 0 + 1 + 1
\end{aligned}
$$

- Type of
- Written

```
size ⌈
size ⌈γ
size ⌈γ⌉
```

- Instead of M
- Possible beca
- First-class con

General Lesson: Support for Pattern Matching on Code

- Think of code as **contextual** HOAS trees with binders in normal form

  Benefit: α-renaming and substitution principles; well-scoped representation; well-typed (if desired)
- Pattern variables are **contextual meta-variables**
- Pattern matching via higher-order pattern matching [Miller, Pientka'03, Abel & Pientka'11]

[Miller'91]

set of "free" variables

How does this help us with meta-programming?

Example : Staged Multiplication Using Box / Unbox

- Defining a code generator for multiplication:

```
mult : Nat → ⌈x : tm nat ⊢ tm nat⌉
mult     0     = box (x. 0)
mult (succ m) = box (x. (unbox (mult m)) + x) % inline call
```

Example : Staged Multiplication Using Box / Unbox

- Defining a code generator for multiplication:

```
mult : Nat → ⌈x : tm nat ⊢ tm nat⌉
mult     0     = box (x. 0)
mult (succ m) = box (x. (unbox (mult m)) + x) % inline call
```

- Generating Code:

What equivalences should hold?

```
mult 1 ≡ box (x. 0 + x)         ≢ box (x. x)
mult 2 ≡ box (x. (0 + x) + x) ≢ box (x. x + x)
```

Example : Staged Multiplication Using Box / Unbox

- Defining a code generator for multiplication:

```
mult : Nat → ⌈x : tm nat ⊢ tm nat⌉
mult     0     = box (x. 0)
mult (succ m) = box (x. (unbox mult m with x) + x) % inline call
```

- Generating Code: **What reductions / equivalences should hold?**

```
mult 1 ≡ box (x. 0 + x)       ≢ box (x. x)
mult 2 ≡ box (x. (0 + x) + x) ≢ box (x. x + x)
```

Remark: We omit the **with** `x` if it is the identity.

Example: Nth Element – Polymorphic Code Generation

LF Representation

```
hd : tm (list A) → tm A.          tl: tm (list A) → tm (list A).
```

Computation

```
nth : Nat →⌈a: obj, v:tm (list a) ⊢ tm a⌉
nth n =
  if n <= 0
  then ⌈a, v ⊢ hd v⌉
  else ⌈a, v ⊢ ⌊nth (n - 1)⌋ with (tl v)⌉
```

Result of polymorphic code generation:

```
nth 2   ⟹*   ⌈a, x ⊢ hd (tl (tl x))⌉
```

Example: Polymorphic Code Generator for Map

LF Representation

```
nil: tm (list A)        cons: tm A → tm (list A) → tm (list A).
```

Computation

```
map :

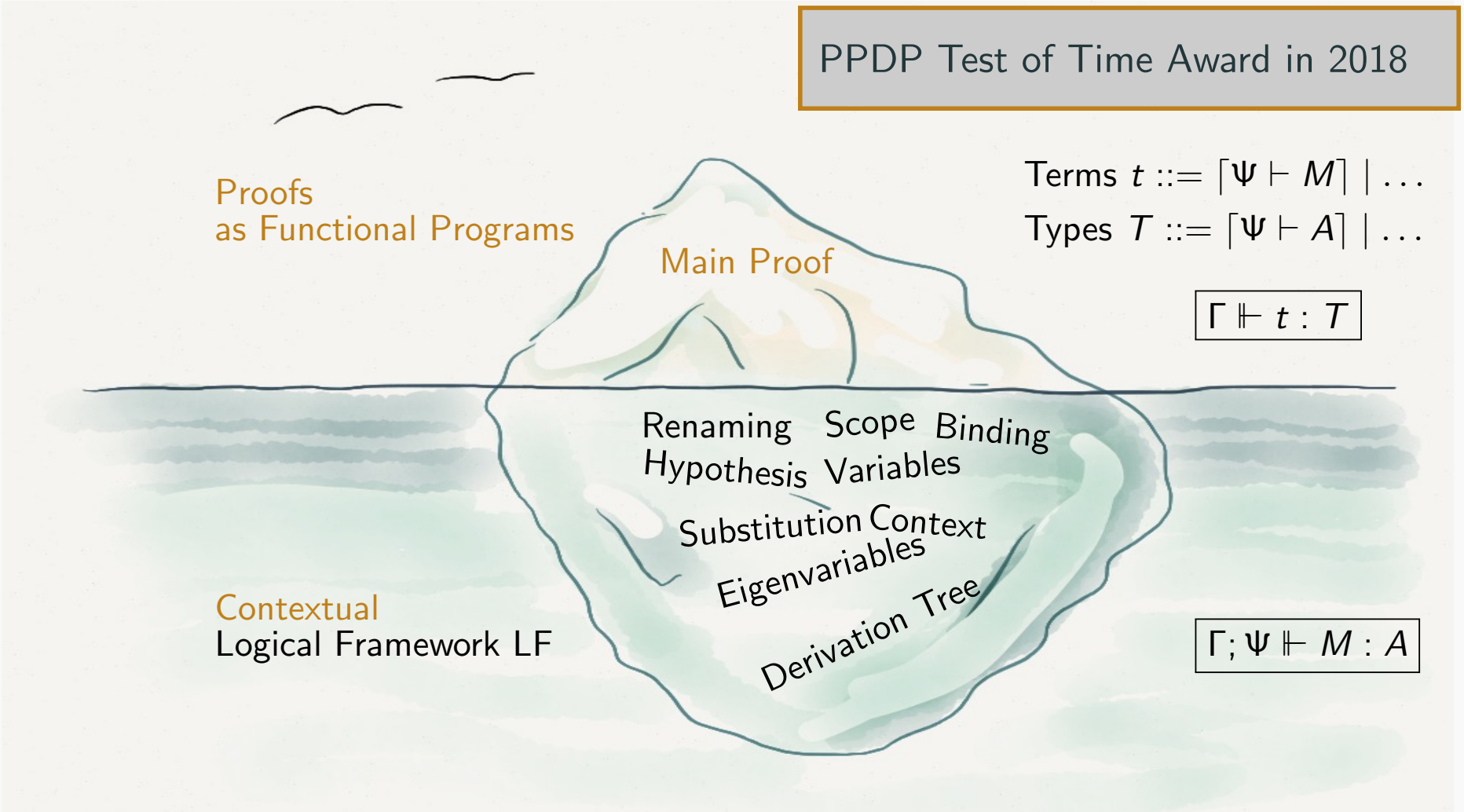
map ⌈
map ⌈
```

Cross-st

BELUGA: Programming with (contextual) HOAS trees
= Meta-Programming with Code

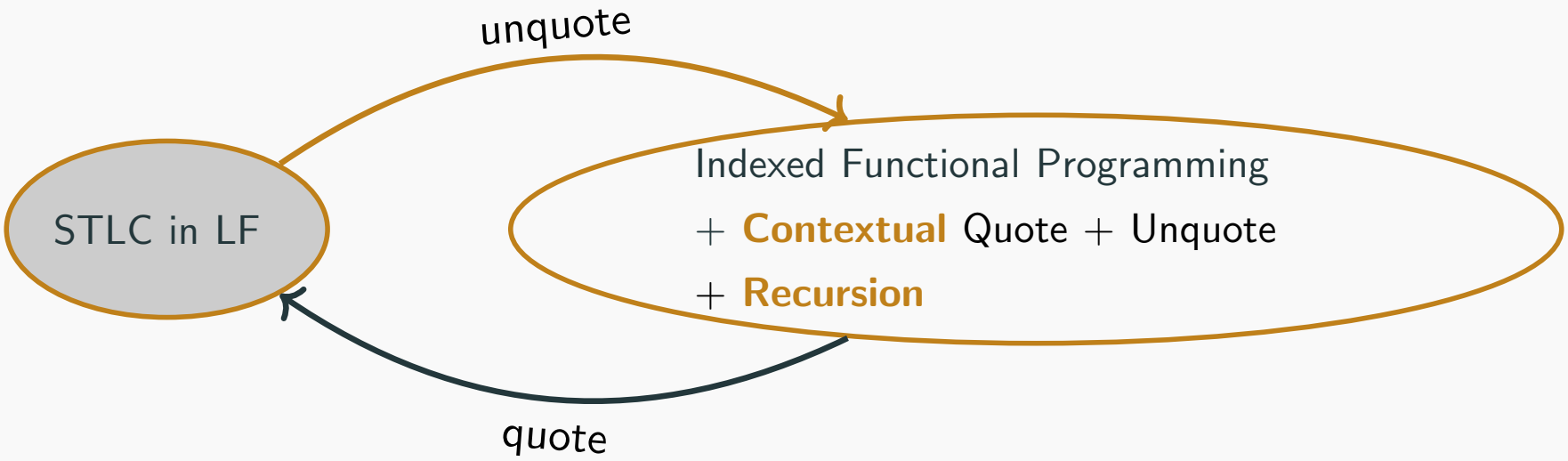
- generate well-typed open code that is in normal form
- compose well-typed open code
- recursively analyze well-typed open code

The Tip of the Iceberg: Beluga [POPL'08, JCAR'10, POPL'12, CADE'15, ICFP'16,...]



Beluga as 2-Layered Meta-Programming Framework

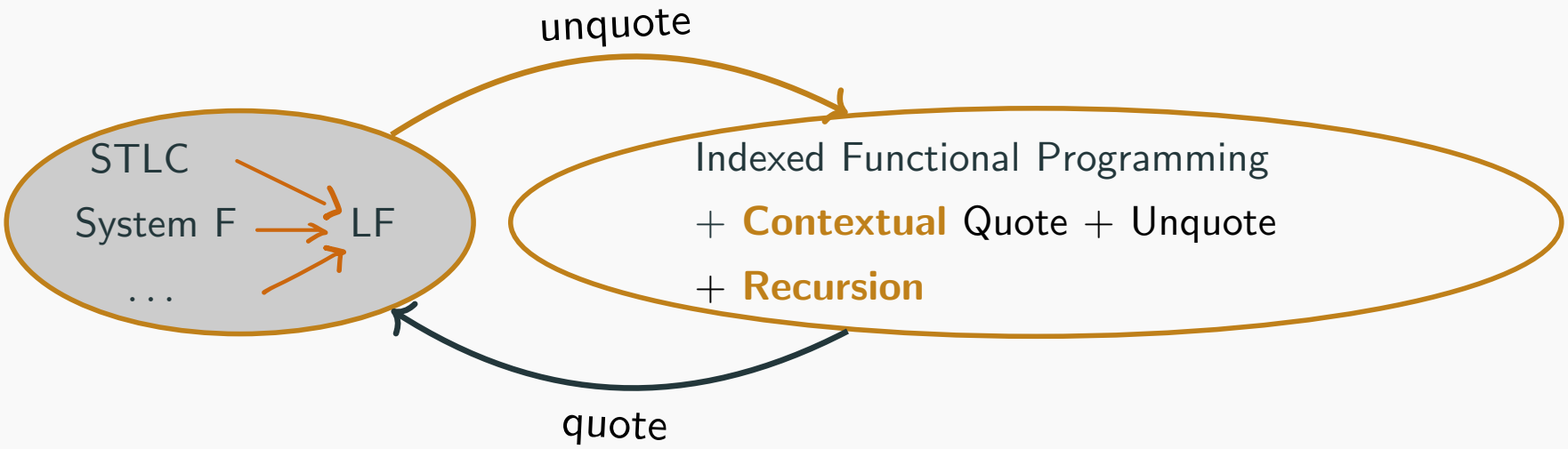
[POPL'08, POPL'12, ICFP'16,...]



- **Code = Contextual HOAS tree**
- **Quote** : Accessing contextual HOAS tree defined in LF
- **Unquote/Splice**: Composing contextual HOAS trees
- **Recursion over HOAS trees**: Analyze contextual HOAS trees recursively

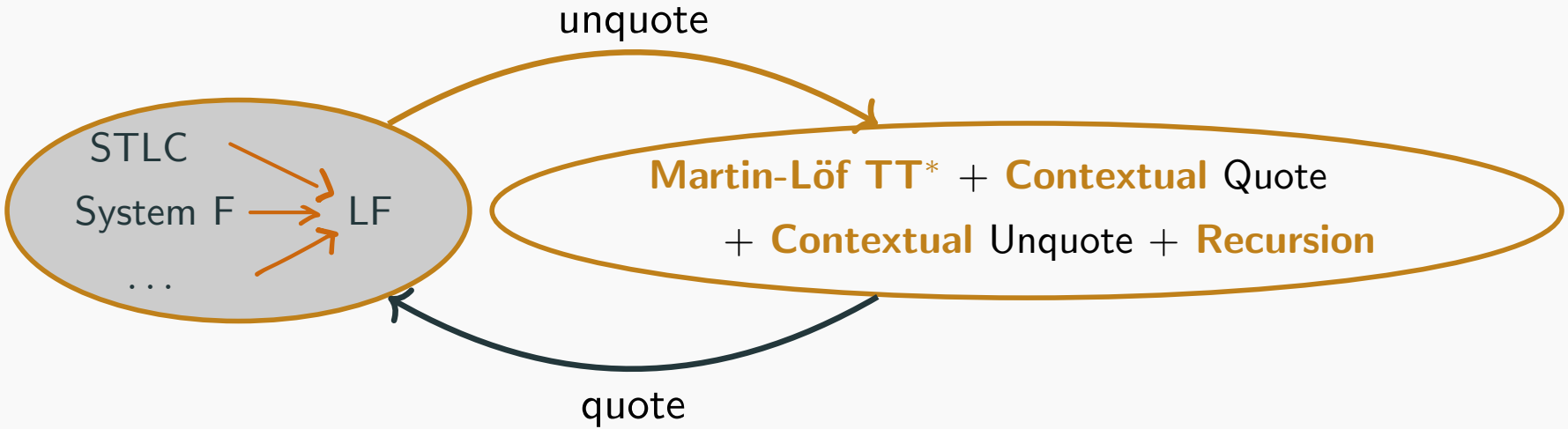
Beluga* as 2-Layered Simply Typed Meta-Programming Framework

[POPL'08, POPL'12, ICFP'16,...]



- **Code = Contextual HOAS tree**
- **Quote** : Accessing contextual HOAS tree defined in LF
- **Unquote/Splice**: Composing contextual HOAS trees
- **Recursion over HOAS trees**: Analyze contextual HOAS trees recursively

Cocon: 2-Layered Dependently Typed Meta-Programming Framework [LICS'19]



- **Code = Contextual HOAS tree**
- **Quote** : Accessing contextual HOAS tree defined in LF
- **Unquote/Splice**: Composing contextual HOAS trees
- **Recursion over HOAS trees**: Analyze contextual HOAS trees recursively
- **Evaluation by Reflection**

Do we have now a type theory with quote?

Let's look at other encodings - adding type-level computation

Assume `nil : (A : tp) → tm (list A)`

Let's look at other encodings - adding type-level computation

Assume $\texttt{nil} : (\texttt{A} : \texttt{tp}) \to \texttt{tm}\ (\texttt{list}\ \texttt{A})$

Code Object $\ulcorner\ \texttt{nil}\ (\texttt{tapp}\ (\texttt{tlam}\ (\lambda\ \texttt{x.x}))\ \texttt{nat})\ \urcorner \not\equiv \ulcorner\ \texttt{nil}\ \texttt{nat}\ \urcorner$

Let's look at other encodings - adding type-level computation

Assume `nil : (A : tp) → tm (list A)`

Code Object $\underbrace{\ulcorner \texttt{nil (tapp (tlam (}\lambda\ \texttt{x.x)) nat)} \urcorner}$ $\not\equiv$ $\underbrace{\ulcorner \texttt{nil nat} \urcorner}$

Code Type $\ulcorner \vdash \texttt{tm (list (tapp (tlam (}\lambda\texttt{x.x)) nat))} \urcorner$ $\ulcorner \vdash \texttt{tm (list nat)} \urcorner$

Let's look at other encodings - adding type-level computation

Assume `nil : (A : tp) → tm (list A)`

Code Object $\lceil$ `nil (tapp (tlam (λ x.x)) nat)` $\rceil$ $\not\equiv$ $\lceil$ `nil nat` $\rceil$

Code Type $\lceil \vdash$ `tm (list (tapp (tlam (λx.x)) nat))` $\rceil$ $\lceil \vdash$ `tm (list nat)` $\rceil$

When we encode systems in LF,

we do not get their equational theory "for free" during type checking.

Let's look at other encodings - adding type-level computation

Assume `nil : (A : tp) → tm (list A)`

Code Object $\lceil$ `nil (tapp (tlam (λ x.x)) nat)` $\rceil$ $\not\equiv$ $\lceil$ `nil nat` $\rceil$

Code Type $\lceil \vdash$ `tm (list (tapp (tlam (λx.x)) nat))` $\rceil$ $\lceil \vdash$ `tm (list nat)` $\rceil$

When we encode systems in LF,
we do not get their equational theory "for free" during type checking.

$\Longrightarrow$ **Design a modal type theory where we control equivalences.**

Key Idea: Layers in Modal Type Theory (TT) for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **(no computation)**
  Assume `nil : (A : Ty) → List A`.

$$\text{Code Object} \quad \ulcorner \texttt{nil ((}\lambda\texttt{x. x) Nat)} \urcorner \quad \not\equiv \quad \ulcorner \texttt{(nil Nat} \urcorner$$

Layers in Modal Type Theory for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **(no computation)**

  Assume `nil : (A : Ty) → List A`.

Code Object	$\ulcorner$ `nil ((λx. x) Nat)` $\urcorner$	$\not\equiv$	$\ulcorner$ `nil Nat` $\urcorner$
Code Type	$\ulcorner \vdash$ `List ((λx. x) Nat)` $\urcorner$		$\ulcorner \vdash$ `List Nat` $\urcorner$

Three Layers in Modal Type Theory for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **(no computation)**

  Assume `nil : (A : Ty) → List A`.

$$\text{Code Object} \quad \underbrace{\textbf{box}\ (\ \texttt{nil}\ ((\lambda\ \texttt{x.}\ \texttt{x})\ \texttt{Nat})\ )}\ \not\equiv\ \underbrace{\textbf{box}\ (\texttt{nil Nat})}$$

$$\text{Code Type} \quad \lceil \vdash \texttt{List}\ ((\lambda\texttt{x.}\ \texttt{x})\ \texttt{Nat}) \rceil \quad \equiv \quad \lceil \vdash \texttt{List Nat} \rceil$$

Allow β-equivalences on code types:

Three Layers in Modal Type Theory for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **(no computation)**

  Assume `nil : (A : Ty) → List A`.

$$\underbrace{\textbf{box}\ (\ \texttt{nil}\ ((\lambda\ \texttt{x.}\ \texttt{x})\ \texttt{Nat})\ )}_{\text{Code Object}} \not\equiv \underbrace{\textbf{box}\ (\texttt{nil}\ \texttt{Nat})}_{\text{Code Object}}$$

$$\lceil \vdash \texttt{List}\ ((\lambda\texttt{x.}\ \texttt{x})\ \texttt{Nat}) \rceil \equiv \lceil \vdash \texttt{List}\ \texttt{Nat} \rceil$$

Code Type Code Type

Allow β-equivalences on code types:

$$((\lambda\texttt{x.}\ \texttt{x})\ \texttt{Nat}) \equiv \texttt{Nat}$$

Three Layers in Modal Type Theory for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **no computation**

  Assume `nil : (A : Ty) → List A`.

  Code Object **box** `( nil ((λx. x) Nat) )` $\not\equiv$ **box** `(nil Nat)`

- **Objects in Type Theory** (D) **includes computation**

  `List ((λx. x) Nat)` $\equiv$ `List Nat`

Three Layers in Modal Type Theory for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **no computation**
  Assume `nil : (A : Ty) → List A`.

  Code Object **box** `( nil ((λx. x) Nat) )` $\not\equiv$ **box** `(nil Nat)`

- **Objects in Type Theory** (D) **includes computation**

  `List ((λx. x) Nat)` $\equiv$ `List Nat`

- **Meta-programming** (M) includes quotation of code objects, composition of code objects, recursion and pattern matching of code objects

Three Layers in Modal Type Theory for Meta-Programming

- A **Code Object** (C) describes the AST of a TT object **no computation**

  Assume `nil : (A : Ty) → List A`.

  Code Object **box** `( nil ((λx. x) Nat) )` $\not\equiv$ **box** `(nil Nat)`

- **Objects in Type Theory** (D) **includes computation**

  `List ((λx. x) Nat)` $\equiv$ `List Nat`

- **Meta-programming** (M) includes quotation of code objects, composition of code objects, recursion and pattern matching of code objects

DeLaM: A Layered Type Theory for Meta-Programming with Intensional Analysis
(joint work with J. Z. Hu [ESOP'24,TOPLAS'25,POPL'25]):
Uniform syntax, BUT syntactic judgments are parameterized by layers : $\mathrm{C} < \mathrm{D} < \mathrm{M}$

Contribution: DeLaM - A Type Theory for Meta-Programming

- **Syntactic type theory** with a non-cumulative, Tarski-style universe hierarchy that supports **quotation, composition, execution of and recursion on TT code**.
- **Uniform syntax**, layers $\{\mathrm{C},\ \mathrm{D},\ \mathrm{M}\}$ control what equivalences hold!
- **Lifting Lemma**: Let layer $i < j$.
  A well-typed object at layer i remains well-typed at layer j.
- **Weak normalization and the decidability of convertibility** based on Kripke logical relations following Abel et al. [2017] (based on PERs)
- Illustrate the power of DeLaM sketching different tactics

Further Research on Homogeneous Meta-Programming

Dependently-Typed Meta-Programming for Code Generation and Staging

- MINT: Kripke-style Modal Intuitionistic Type Theory
  J. Z Hu, J. Jang, and B. Pientka[JFP 2023]
  – no pattern matching on code; code is treated as a black-box; as a consequence both code objects and code types are normalized; useful for code generation; nested boxes allowed

Meta-Programming with Intensional Code Analysis

- MOEBIUS: Where System F can pattern match on itself (J. Jang et al [POPL'22])
  – pattern matching on System F code where code is represented as **(contextual) HOAS objects**; nested boxes allowed; no normalization proof

Further Research on Heterogeneous Meta-Programming

Meta-Programming for Generating and Analyzing Session-Typed Processes

- Fusing Session-typed Concurrent Programming into Functional Programming
  C. Sano et al [ICFP'25]
  – Two Layers: Session Types for processes and Simple Types for functional programs; support for **pattern matching on processes using contextual types** and for **observing process behaviour**

Further Research on Heterogeneous Meta-Programming

Meta-Programming for Generating and Analyzing Session-Typed Processes

- Fusing Session-typed Concurrent Programming into Functional Programming C. Sano et al [ICFP'25]
  – Two Layers: Session Types for processes and Simple Types for functional programs; support for **pattern matching on processes using contextual types** and for **observing process behaviour**

Meta-Programming for Hybrid Quantum Programming

- Deconstructed Proto-Quipper: A Rational Reconstruction C. Sano et al [PPDP'26]
  – Functional Meta-programming layer that generates quantum circuits described using a linear lambda-calculus with gate constants;simplifies the formal foundation of Proto-Quipper
- Staged Hybrid Quantum Programming C. Sano et al [submitted]
  Extend the previous system with a classical runtime layer to support dynamic lifting

We have (mostly) achieved a type-theoretic foundation for meta-programming.

The Surprising Effectiveness of Contextual Types Beyond Meta-Programming

The Surprising Effectiveness of Contextual Types Beyond Meta-Programming

- Modelling unification alg. (Abel, Pientka, et al [IJCAR'06,TLCA'11])

The Surprising Effectiveness of Contextual Types Beyond Meta-Programming

- Modelling unification alg. (Abel, Pientka, et al [IJCAR'06,TLCA'11])
- Interactive programming (C. Omar et al [POPL'19, PLDI'21, OOPSLA'24])

The Surprising Effectiveness of Contextual Types Beyond Meta-Programming

- Modelling unification alg. (Abel, Pientka, et al [IJCAR'06,TLCA'11])
- Interactive programming (C. Omar et al [POPL'19, PLDI'21, OOPSLA'24])
- Programming with algebraic effects and capabilities (Nanevski et al [ICFP'21], Xu et al OOPSLA'25])

The Surprising Effectiveness of Contextual Types Beyond Meta-Programming

- Modelling unification alg. (Abel, Pientka, et al [IJCAR'06,TLCA'11])
- Interactive programming (C. Omar et al [POPL'19, PLDI'21, OOPSLA'24])
- Programming with algebraic effects and capabilities (Nanevski et al [ICFP'21], Xu et al OOPSLA'25])
- Reasoning about universes in homotopy type (Licata et al [FSCD'2018])
- ...

What I Learned

Lesson 1: Contextual HOAS provides a high-level representation of code in canonical form that is stable under α-renaming and suitable for pattern matching.

Lesson 2: Layering allows us to systematically grow a meta-programming system.

Lesson 3: Layers allow us to syntactically characterize the code and control equivalences (reductions)

Lesson 4: Modelling code via contextual HOAS provides the right abstraction for a range of meta-programming systems – from dependently-typed homogeneous meta-programming to heterogeneous systems.