Cocon:
A Type Theory for Defining Logics and Proofs

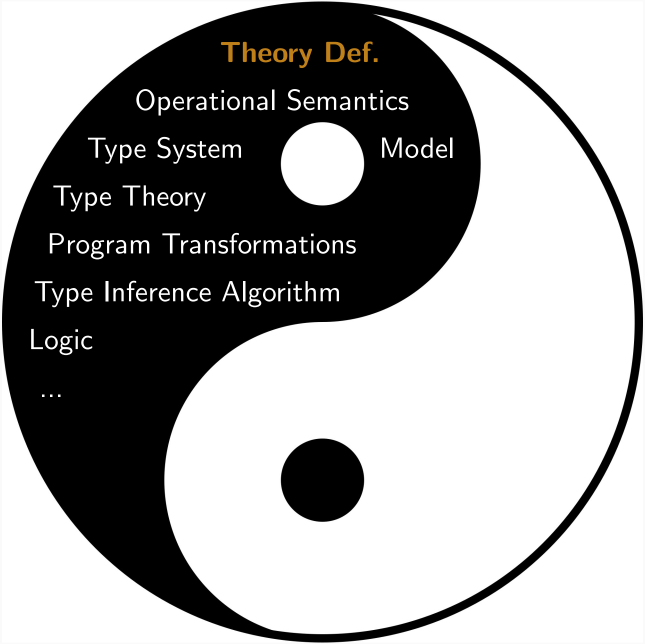
Brigitte Pientka

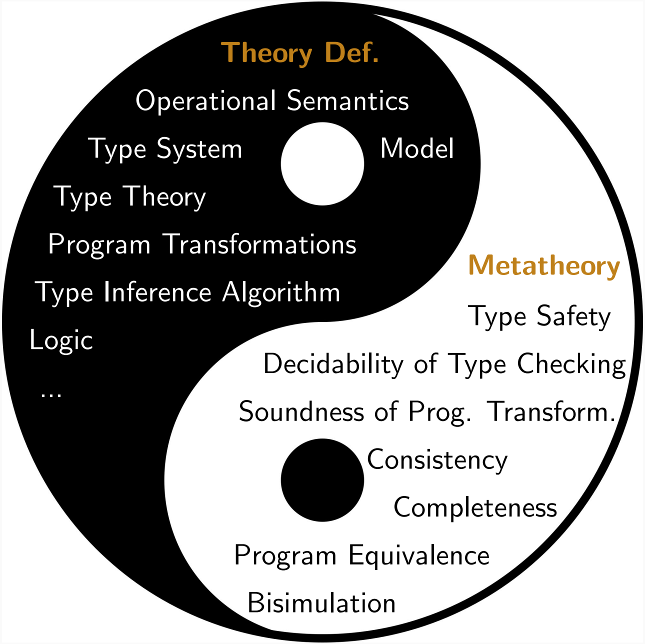
McGill University
Montreal, Canada

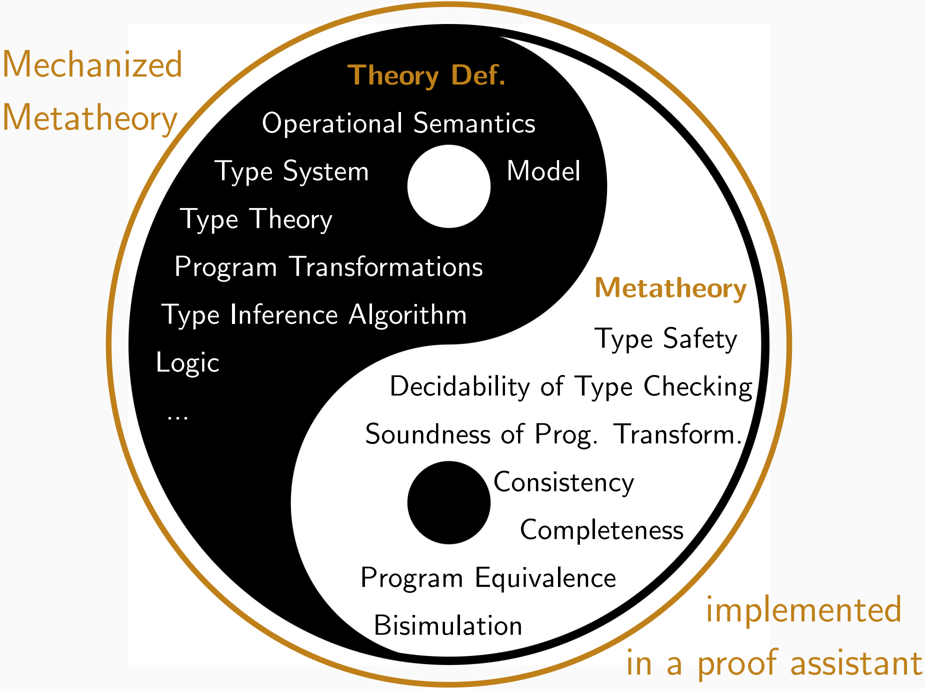
beluga

What are good high-level proof languages that make it easier to mechanize metatheory?

What are good high-level proof languages that make it easier to mechanize metatheory?


Theory Def.
Operational Semantics
Type System
Model
Type Theory
Program Transformations
Type Inference Algorithm
Logic
...


Theory Def.
Operational Semantics
Type System
Model
Type Theory
Program Transformations
Type Inference Algorithm
Logic
...
Metatheory
Type Safety
Decidability of Type Checking
Soundness of Prog. Transform.
Consistency
Completeness
Program Equivalence
Bisimulation


Mechanized
Metatheory
Theory Def.
Operational Semantics
Type System
Model
Type Theory
Program Transformations
Type Inference Algorithm
Logic
...
Metatheory
Type Safety
Decidability of Type Checking
Soundness of Prog. Transform.
Consistency
Completeness
Program Equivalence
Bisimulation
implemented
in a proof assistant

Correct proofs are tricky to write.

On paper:

- Challenging to keep track of all the details
- Easy to skip over details
- Difficult to understand interaction between different features
- Difficulties increase with size

In a proof assistant:

- A lot of overhead in building basic infrastructure
- May get lost in the technical, low-level details
- Time consuming
- Experience, experience, experience

Mechanizing Normalization for STLC

"To those that doubted de Bruijn, I wished to prove them wrong, or discover why they were right. Now, after some years and many hundred hours of labor, I can say with some authority: they were right. De Bruijn indices are foolishly difficult for this kind of proof. [...] The full proof runs to 3500 lines, although that relies on a further library of 1900 lines of basic facts about lists and sets. [...] the cost of de Bruijn is partly reflected in the painful 1600 lines that are used to prove facts about "shifting" and "substitution"."

Ezra Cooper (PhD Student)

https://github.com/ezrakilty/sn-stlc-de-bruijn-coq

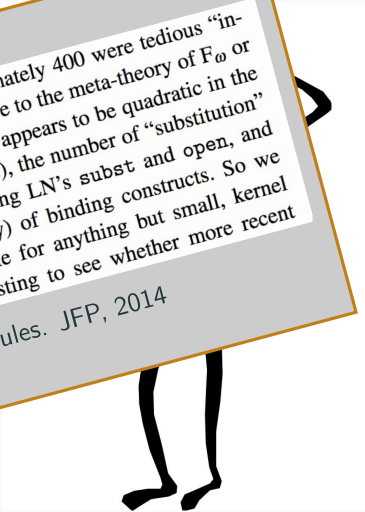
Mechanizing Normalization for STLC

“To those that doubted de Bruijn, I wished to prove them wrong, or discover why … were right. Now, after … many hundr…

…ooper (PhD Student)

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



https://github.com/ezrakilty/sn-stlc-de-bruijn-coq

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]

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 abstraction[…] his problem area. The programmer is th[…] effort in the right place; h[…] and the resulti[…]

Goal:

A dependent type theory (similar to Coq or Agda) that provides the right abstractions for compactly and elegantly defining logics and proofs.

"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$\Rightarrow$nat.x)	`lam nat λx.(lam (arr nat nat) λx.x)`
lam x:nat. (lam f:nat$\Rightarrow$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 $A, B ::= \mathsf{nat} \mid A \Rightarrow B \mid \alpha \mid \forall \alpha.A$

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

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)
$\mathsf{tlam}\ \alpha.\ (\mathsf{lam}\ x{:}\alpha.x)$	`tlam λa.(lam a λx.x)`
$\forall\alpha.\forall\beta.\alpha \Rightarrow \beta$	`all λa.all λb.arr a b`

Uniformly Model Binding Structures using LF Functions

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

… | …

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**
- **Benefit:** α-renaming and substitution principles
- **Scales:** Model derivation trees
  - Hypothetical derivations as LF functions
  - Parametric derivations as LF functions

tl… λx.x)

$\forall\alpha$ … a.all λb.arr a b

Sounds cool… can I do this in OCaml or Agda?

An Attempt in OCaml

OCaml

```ocaml
type tm = Lam of (tm -> tm)
let apply = function (Lam f) -> f
let omega =  Lam (function x -> apply x x)
```

What happens, when we try to evaluate `apply omega omega`?

An Attempt in OCaml

OCaml

```
type tm = Lam of (tm -> tm)
let apply = function (Lam f) -> f
let omega =  Lam (function x -> apply x x)
```

What happens, when we try to evaluate `apply omega omega`?

It will loop.

An Attempt in OCaml and Agda

OCaml

```
type tm = Lam of (tm -> tm)
let apply = function (Lam f) -> f
let omega =  Lam (function x -> apply x x)
```

What happens, when we try to evaluate `apply omega omega`?

It will loop.

Agda

```
data tm : type = lam : (tm → tm) → tm
```

Violates positivity restriction

An Attempt in OCaml and Agda

OCaml

```
type tm = Lam of (tm -> tm)
let apply = function (Lam f) -> f
let omega =  Lam (fun
```

Functions in OCaml and Agda are opaque (black box).

- We **can observe** the result that a function computes
- We **cannot pattern match** to inspect the function body

Ag

```
d    tm : type = lam : (tm → tm) → tm
```

Violates positivity restriction

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

$$
\begin{array}{lllll}
 & \texttt{size} & (\texttt{lam}\,\lambda\texttt{x}.\texttt{lam}\,\lambda\texttt{f}.\ \texttt{app f x}) & & \\
\Longrightarrow & \texttt{size} & (\texttt{lam}\,\lambda\texttt{f}.\ \texttt{app f } {\color{red}\texttt{x}}) + 1 & & \\
\Longrightarrow & \texttt{size} & (\texttt{app } {\color{red}\texttt{f}}\ {\color{red}\texttt{x}}) + 1 + 1 & & \\
\Longrightarrow & & \texttt{size } {\color{red}\texttt{f}} \quad + \quad \texttt{size } {\color{red}\texttt{x}} + 1 + 1 + 1 & & \\
\Longrightarrow & & 0 \quad + \quad 0 \quad + 1 + 1 + 1 & &
\end{array}
$$

"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]*

Back in 2008. . .

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:

$$\underset{\substack{\uparrow \\ \text{LF Context}}}{\Psi} \Vdash \underset{\substack{\uparrow \\ \text{LF Term}}}{M} : \underset{\substack{\uparrow \\ \text{LF Type}}}{A}$$

LF and Holes in HOAS trees – Revisited

In LF (Meta Lang.)

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

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

LF Typing Judgment:

$$\underset{\text{LF Context}}{\underset{\uparrow}{\texttt{x}{:}\texttt{tm}}} \Vdash \underset{\text{LF Term}}{\underset{\uparrow}{\texttt{lam}\,\lambda\texttt{f}.\texttt{app f x}}} : \underset{\text{LF Type}}{\underset{\uparrow}{\texttt{tm}}}$$

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:

$$\underbrace{\texttt{x}{:}\texttt{tm}}_{\text{LF Context}} \ \Vdash\ \texttt{lam}\ \lambda \mathsf{f}.\underbrace{\boxed{\texttt{app}\ f\ x}}_{\text{LF Term}}\ :\ \underbrace{\texttt{tm}}_{\text{LF Type}}$$

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

LF Typing Judgment:

$$\underset{\substack{\uparrow \\ \text{LF Context}}}{x{:}\texttt{tm}} \Vdash \texttt{lam}\ \lambda f.\underset{\substack{\uparrow \\ \text{LF Term}}}{\boxed{\texttt{app}\ f\ x}} : \underset{\substack{\uparrow \\ \text{LF Type}}}{\texttt{tm}}$$

What is the type of $\boxed{\texttt{app}\ f\ x}$? – Its type is $\lceil x{:}\texttt{tm}, f{:}\texttt{tm} \vdash \texttt{tm} \rceil$.

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

$$\underbrace{\mathtt{h}{:}\ \lceil \mathtt{x}{:}\mathtt{tm}, \mathtt{f}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil}_{\text{Meta Context}}\ ;\ \underbrace{\mathtt{x}{:}\mathtt{tm}}_{\text{LF Context}} \Vdash \underbrace{\mathtt{lam}\ \lambda f.\ \ \mathtt{h}\qquad\qquad}_{\text{LF Term}} : \underbrace{\mathtt{tm}}_{\text{LF Type}}$$

- h is a contextual variable
- It has the contextual type $\lceil \mathtt{x}{:}\mathtt{tm}, \mathtt{f}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil$
- It can be instantiated with a contextual term $\lceil \mathtt{x,f} \vdash \mathtt{app\ f\ x} \rceil$
- Contextual types ($\vdash$) reify LF typing derivations ($\Vdash$)

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

$$\mathtt{h}{:}\ \lceil \mathtt{x}{:}\mathtt{tm}, \mathtt{f}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil \;;\; \mathtt{x}{:}\mathtt{tm} \Vdash \underbrace{\mathtt{lam}\ \lambda f.\ \ \mathtt{h}\qquad\quad}_{\text{LF Term}} : \mathtt{tm}$$

Meta Context LF Context LF Type

- $\mathtt{h}$ is a contextual variable
- It has the contextual type $\lceil \mathtt{x}{:}\mathtt{tm}, \mathtt{f}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil$
- It can be instantiated with a contextual term $\lceil \mathtt{x,f} \vdash \mathtt{app\ f\ x} \rceil$
- Contextual types ($\vdash$) reify LF typing derivations ($\Vdash$)

WAIT! ... whatever we plug in for $\mathtt{h}$ may contain free LF variables?

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

$$\mathtt{h}{:}\ \lceil \mathtt{y}{:}\mathtt{tm}, \mathtt{g}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil\ ;\ \mathtt{x}{:}\mathtt{tm} \Vdash \underbrace{\mathtt{lam}\ \lambda f.\ \ \mathtt{h}\qquad\qquad}\ :\ \mathtt{tm}$$

Meta Context LF Context LF Term LF Type

- h is a contextual variable
- It has the contextual type $\lceil \mathtt{y}{:}\mathtt{tm}, \mathtt{g}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil$
- It can be instantiated with a contextual term $\lceil \mathtt{y}, \mathtt{g} \vdash \mathtt{app\ g\ y} \rceil$
- Contextual types ($\vdash$) reify LF typing derivations ($\Vdash$)

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]

$$\mathtt{h}{:}\ \lceil \mathtt{y}{:}\mathtt{tm}, \mathtt{g}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil \underset{\text{Meta Context}}{\uparrow}\ ;\ \underset{\text{LF Context}}{\mathtt{x}{:}\mathtt{tm}} \Vdash \underbrace{\mathtt{lam}\ \lambda f.\ \mathtt{h}[\mathtt{x}/\mathtt{y},\ \mathtt{f}/\mathtt{g}]}_{\text{LF Term}}\ :\ \underset{\text{LF Type}}{\mathtt{tm}}$$

- $\mathtt{h}$ is a contextual variable
- It has the contextual type $\lceil \mathtt{y}{:}\mathtt{tm}, \mathtt{g}{:}\mathtt{tm} \vdash \mathtt{tm} \rceil$
- It can be instantiated with a contextual term $\lceil \mathtt{y,g} \vdash \mathtt{app\ g\ y} \rceil$
- Contextual types ($\vdash$) reify LF typing derivations ($\Vdash$)

WAIT! ... whatever we plug in for $\mathtt{h}$ 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]

$$\underbrace{\Gamma}_{\text{Meta Context (global)}} \; ; \; \underbrace{\Psi}_{\text{LF Context (local)}} \;\Vdash\; \underbrace{M}_{\text{LF Term}} : \underbrace{A}_{\text{LF Type}}$$

LF Variable

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

Contextual Variable

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

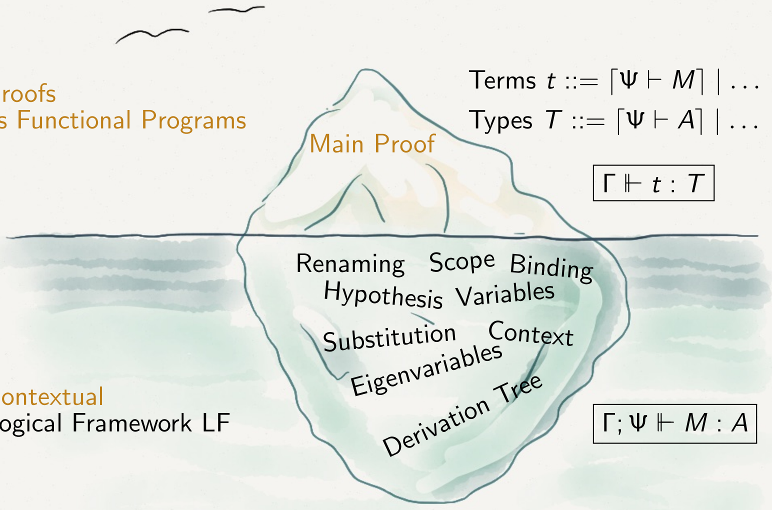
[1]Footnote for nerds: CTT is a generalization of modal S4.

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

Proofs
as Functional Programs

Terms $t ::= \lceil \Psi \vdash M \rceil \mid \ldots$

Types $T ::= \lceil \Psi \vdash A \rceil \mid \ldots$



$\boxed{\Gamma \Vdash t : T}$

Contextual
Logical Framework LF

$\boxed{\Gamma; \Psi \Vdash M : A}$

Revisiting the program size

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

Revisiting the program size

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

Corresponding program:

```
size : Πγ:ctx. ⌈γ ⊢ tm⌉ → int
size ⌈γ ⊢ #p⌉ = 0
size ⌈γ ⊢ lam λx. M⌉ = size ⌈γ,x ⊢ M⌉ + 1
size ⌈γ ⊢ app M N⌉ = size ⌈γ ⊢ M⌉ + size ⌈γ ⊢ N⌉ + 1;
```

- Abstract over context γ and introduce special variable pattern #p
- Higher-order pattern matching [Miller'91]

What Programs / Proofs Can We Write?

- **Certified programs**:
  Type-preserving closure conversion and hoisting [CPP'13]
  Joint work with O. Savary-Bélanger, S. Monnier
- **Inductive proofs:**
  Logical relations proofs (Kripke-style) [MSCS'18]
  Joint work with A. Cave

  POPLMark Reloaded: Strong Normalization for STLC using Kripke-style Logical Relations
  Joint work with A. Abel, G. Allais, A. Hameer, A. Momigliano, S. Schäfer, K. Stark
- **Coinductive proofs:**
  Bisimulation proof using Howe's Method [MSCS'18]
  Joint work with D. Thibodeau and A. Momigliano

Sounds cool. . . but how can we get this into type theories (like Agda)?

The Essence of the Problem

$$\Gamma \; ; \; \Psi \;\; \Vdash \;\; M \; : \; A$$

Meta Context (global)

LF Context (local)

LF Term

LF Type

The strict separation between contextual LF and computations means we cannot embed computation terms directly.

Contextual Variable Rule

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

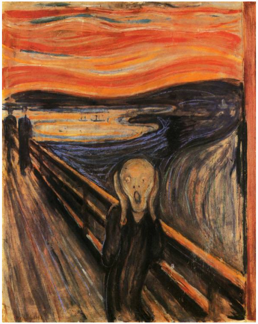
The Essence of the Problem and its Solution?

$$\underset{\substack{\text{Meta Context}\\\text{(global)}}}{\Gamma}\;;\;\underset{\substack{\text{LF Context}\\\text{(local)}}}{\Psi}\;\Vdash\;\underset{\text{LF Term}}{M}\;:\;\underset{\text{LF Type}}{A}$$

What if we did?

Rule for Embedding Computations

$$\frac{\Gamma \Vdash t : \lceil \Phi \vdash A \rceil \qquad \Gamma ; \Psi \Vdash \sigma : \Phi}{\Gamma ; \Psi \Vdash \underbrace{\lfloor t \rfloor_{\sigma}}_{\text{Closure}} \;:\; \underbrace{[\sigma]A}_{\text{Apply subst. } \sigma}}$$



A Type Theory for Defining Logics and Proofs [LICS'19]

Joint work with A. Abel, F. Ferreira, D. Thibodeau, R. Zucchini

- Hierarchy of universes and type-level computation
- Writing proofs about functions (such as `size`)

$$\begin{tikzcd}
\begin{array}{c}\text{LF (intensional)}\\ \Gamma;\Psi \Vdash M : A\end{array} \arrow[r, bend left, "\text{unquote / unbox } \lfloor t \rfloor_{\sigma}"] & \begin{array}{c}\text{Computation (extensional)}\\ \Gamma \Vdash t : \tau\end{array} \arrow[l, bend left, "\text{quote / box } \lceil \Psi \vdash M \rceil"]
\end{tikzcd}$$

see our LICS'19 paper and the extended report for the technical development of the normalization proof.

Sketch: Translation Between STLC and CCC

STLC

```
tm: obj → type
tUnit: tm one.
tPair: tm A → tm B
    → tm (cross A B).
tFst : tm (cross A B)
    → tm A.
tSnd : tm (cross A B)
    → tm B.
tLam : (tm A → tm B)
    → tm (arrow A B).
tApp : tm (arrow A B) → tm A
    → tm B.
```

Cartesian Closed Categories (CCC)

```
mor : obj → obj → type.
id  : mor A A.
@   : mor B C → mor A B
    → mor A C.
drop: mor A one.
fst : mor (cross A B) A.
snd : mor (cross A B) B.
pair: mor A B → mor A C
    → mor A (cross B C).
app : mor (cross (arrow B C) B) C.
cur : mor (cross A B) C
    → mor A (arrow B C).
```

itm

Sketch: Translation Between STLC and CCC

STLC

$\texttt{tm: obj} \to \textbf{type}.$

Cartesian Closed Categories (CCC)

$\texttt{mor:obj} \to \texttt{obj} \to \textbf{type}$

itm

A concrete example: $\texttt{itm}\ \lceil \vdash \texttt{tLam}\ \lambda\texttt{x. tLam}\ \lambda\texttt{f. tApp f x}\rceil$

$\Longrightarrow^{*} \texttt{itm}\ \lceil \texttt{x:tm A,f:tm (arrow A B)} \vdash \texttt{tApp f x}\rceil$

Sketch: Translation Between STLC and CCC

STLC

`tm: obj` $\rightarrow$ **type**.

Cartesian Closed Categories (CCC)

`mor:obj` $\rightarrow$ `obj` $\rightarrow$ **type**

`itm`

A concrete example: `itm` $\lceil \vdash$ `tLam` λ`x.` `tLam` λ`f.` `tApp f x`$\rceil$

$\Longrightarrow^*$ `itm` $\lceil$`x:tm A,f:tm (arrow A B)` $\vdash$ `tApp f x`$\rceil$

Translate an LF context γ to cross product: `ictx:`$\Pi\gamma$`:ctx.`$\lceil \vdash$ `obj`$\rceil$

Example: `ictx` $(x_1{:}\mathtt{tm}\ A_1,\ x_2{:}\mathtt{tm}\ A_2) \Longrightarrow (\mathtt{cross}\ (\mathtt{cross}\ \mathtt{one}\ \ A_1)\ A_2)$

Sketch: Translation Between STLC and CCC

STLC	Cartesian Closed Categories (CCC)
`tm: obj` $\rightarrow$ **type**.	`mor:obj` $\rightarrow$ `obj` $\rightarrow$ **type**

`itm`

A concrete example: `itm` $\lceil\ \vdash$ `tLam` λ`x. tLam` λ`f. tApp f x`$\rceil$

$\Longrightarrow^{*}$ `itm` $\lceil$`x:tm A,f:tm (arrow A B)` $\vdash$ `tApp f x`$\rceil$

Translate an LF context γ to cross product: `ictx:`$\Pi\gamma$`:ctx.`$\lceil\ \vdash$ `obj`$\rceil$

Example: `ictx` $(\mathtt{x}_1\mathtt{:tm\ A}_1,\ \mathtt{x}_2\mathtt{:tm\ A}_2) \Longrightarrow$ `(cross (cross one` $\mathtt{A}_1$`)` $\mathtt{A}_2$`)`

Translate STLC to CCC

`itm:`$\Pi\gamma$`:ctx.`Π`A:`$\lceil\vdash$ `obj`$\rceil$`.`$\lceil\gamma\vdash$ `tm` $\lfloor$`A`$\rfloor\rceil \rightarrow \lceil\vdash$ `mor` $\lfloor$`ictx` $\gamma\rfloor$ $\lfloor$`A`$\rfloor\rceil$

Translate an LF context γ to cross product

```
ictx:Πγ:ctx.⌈ ⊢ obj⌉
```

```
fn  ·                  = ⌈ ⊢ one⌉
 | γ, x:tm ⌊A⌋         = ⌈ ⊢ cross ⌊ictx γ⌋ ⌊A⌋⌉;
```

Example: `ictx (x₁:tm A₁, x₂:tm A₂)` $\Longrightarrow$ `(cross (cross one A₁) A₂)`

Translate an LF context γ to cross product

```
ictx:Πγ:ctx.⌈ ⊢ obj⌉
```

```
fn ·                         = ⌈ ⊢ one⌉
 | γ, x:tm (⌊A⌋ with ·) = ⌈ ⊢ cross ⌊ictx γ⌋ ⌊A⌋⌉;
```

Example: `ictx (x₁:tm A₁, x₂:tm A₂)` $\Longrightarrow$ `(cross (cross one A₁) A₂)`

Translate STLC to CCC

```
itm:Πγ:ctx.ΠA:⌈ ⊢ obj⌉.⌈γ ⊢ tm ⌊A⌋⌉ → ⌈ ⊢ mor ⌊ictx γ⌋ ⌊A⌋⌉
```

Translate STLC to CCC

$\texttt{itm}:\Pi\gamma:\texttt{ctx}.\Pi\texttt{A}:\lceil\vdash \texttt{obj}\rceil.\lceil\gamma\vdash \texttt{tm}\ (\lfloor\texttt{A}\rfloor\ \textbf{with}\cdot)\rceil \to \lceil\vdash \texttt{mor}\ \lfloor\texttt{ictx}\ \gamma\rfloor\ \lfloor\texttt{A}\rfloor\rceil$

Translate STLC to CCC

$\texttt{itm}:\Pi\gamma:\texttt{ctx}.\Pi\texttt{A}:\lceil\vdash\texttt{obj}\rceil.\lceil\gamma\vdash\texttt{tm}\ (\lfloor\texttt{A}\rfloor\ \textbf{with}\cdot)\rceil \to \lceil\vdash\texttt{mor}\ \lfloor\texttt{ictx}\ \gamma\rfloor\ \lfloor\texttt{A}\rfloor\rceil$

Idea: Write a recursive function pattern matching on m

$$
\begin{array}{lll}
\textbf{fn} & \lceil\gamma\vdash_{\#}\ \texttt{p}\rceil & = \texttt{ivar}\ \gamma\ \texttt{p} \\
\mid & \lceil\gamma\vdash\ \texttt{tUnit}\rceil & = \lceil\ \vdash\ \texttt{drop}\rceil \\
\mid & \lceil\gamma\vdash\ \texttt{tFst}\ \lfloor\texttt{e}\rfloor\rceil & = \lceil\ \vdash\ \texttt{fst}\ @\ \lfloor\texttt{itm e}\rfloor\rceil \\
\mid & \lceil\gamma\vdash\ \texttt{tSnd}\ \lfloor\texttt{e}\rfloor\rceil & = \lceil\ \vdash\ \texttt{snd}\ @\ \lfloor\texttt{itm e}\rfloor\rceil \\
\mid & \lceil\gamma\vdash\ \texttt{tPair}\ \lfloor\texttt{e1}\rfloor\ \lfloor\texttt{e2}\rfloor\rceil & = \lceil\ \vdash\ \texttt{pair}\ \lfloor\texttt{itm e1}\rfloor\ \lfloor\texttt{itm e2}\rfloor\rceil \\
\mid & \lceil\gamma\vdash\ \texttt{tLam}\ \lambda\texttt{x}.\lfloor\texttt{e}\rfloor\rceil & = \lceil\ \vdash\ \texttt{cur}\ \lfloor\texttt{itm e}\rfloor\rceil \\
\mid & \lceil\gamma\vdash\ \texttt{tApp}\ \lfloor\texttt{e1}\rfloor\ \lfloor\texttt{e2}\rfloor\rceil & = \lceil\ \vdash\ \texttt{app}\ @\ \texttt{pair}\ \lfloor\texttt{itm e1}\rfloor\ \lfloor\texttt{itm e2}\rfloor\rceil;
\end{array}
$$

Translation of CCC to STLC

Given a morphism between `A` and `B`, we build a term of type `B` with one variable of type `A`.

```
imorph:Π A:⌈ ⊢ obj⌉.Π B:⌈ ⊢ obj⌉.
        ⌈ ⊢ mor ⌊A⌋ ⌊B⌋⌉ ⇒ ⌈x:tm ⌊A⌋ ⊢ tm ⌊B⌋⌉
```

Translation of CCC to STLC

Given a morphism between `A` and `B`, we build a term of type `B` with one variable of type `A`.

```
imorph:Π A:⌈ ⊢ obj⌉.Π B:⌈ ⊢ obj⌉.
        ⌈ ⊢ mor ⌊A⌋ ⌊B⌋⌉ ⇒ ⌈x:tm ⌊A⌋ ⊢ tm (⌊B⌋ with·)⌉
```

Translation of CCC to STLC

Given a morphism between A and B, we build a term of type B with one variable of type A.

```
imorph:Π A:⌈ ⊢ obj⌉.Π B:⌈ ⊢ obj⌉.
        ⌈ ⊢ mor ⌊A⌋ ⌊B⌋⌉ ⇒ ⌈x:tm ⌊A⌋ ⊢ tm (⌊B⌋ with·)⌉

fn ⌈ ⊢ id⌉            = ⌈x:tm _ ⊢ x⌉
 | ⌈ ⊢ drop⌉          = ⌈x:tm _ ⊢ tUnit⌉
 | ⌈ ⊢ fst⌉           = ⌈x:tm _ ⊢ tFst x⌉
 | ⌈ ⊢ snd⌉           = ⌈x:tm _ ⊢ tSnd x⌉
 | ⌈ ⊢ pair ⌊f⌋ ⌊g⌋⌉ = ⌈x:tm _ ⊢ tPair ⌊imorph f⌋ ⌊imorph g⌋⌉
 | ⌈ ⊢ cur ⌊f⌋⌉       = ⌈x:tm _ ⊢ tLam λy.(⌊imorph f⌋ with tPair x y)⌉
 | ⌈ ⊢ ⌊f⌋ @ ⌊g⌋⌉     = ⌈x:tm _ ⊢ ⌊imorph f⌋ with ⌊imorph g⌋⌉
 | ⌈ ⊢ app⌉           = ⌈x:tm _ ⊢ tApp (tFst x) (tSnd x)⌉;
```

Bridging the Gap between LF and Martin Löf Type Theory

What we've already done – What's Next

Theory

- ✓ Normalization
- ✓ Decidable equality
- Categorical semantics
- ...

Implementation and Case Studies

- Build an extension to Coq/Agda/Beluga
- Case studies:
  - Equivalence of STLC and CCC
  - Homotopy Type Theory (see relations to Crisp Type Theory)
- Meta-Programming (Tactics)
- Compilation
- ...

Towards More Civilized High-Level Proof Languages

Lesson 1: Contextual types provide a type-theoretic framework to think about syntax trees within a context of assumptions.

Lesson 2: Contextual types allow us to mediate and mix between strong (computation-level) function types and weak (HOAS) function types.

Lesson 3: Existing proof technique of defining a model for well-typed terms based on their semantic type scales.

Taken Together: This is a first step towards bridging the long-standing gap between LF and Martin Löf type theories.