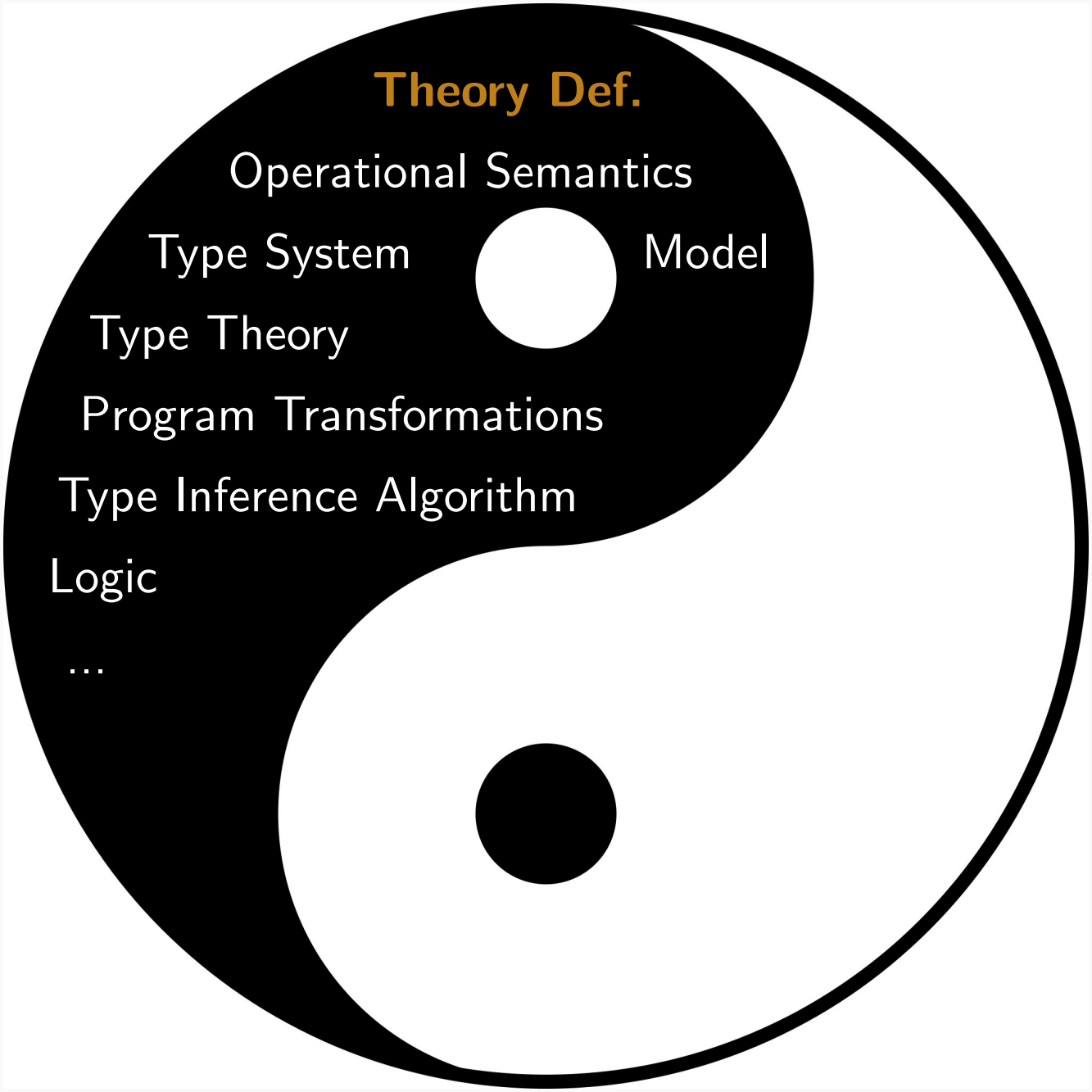
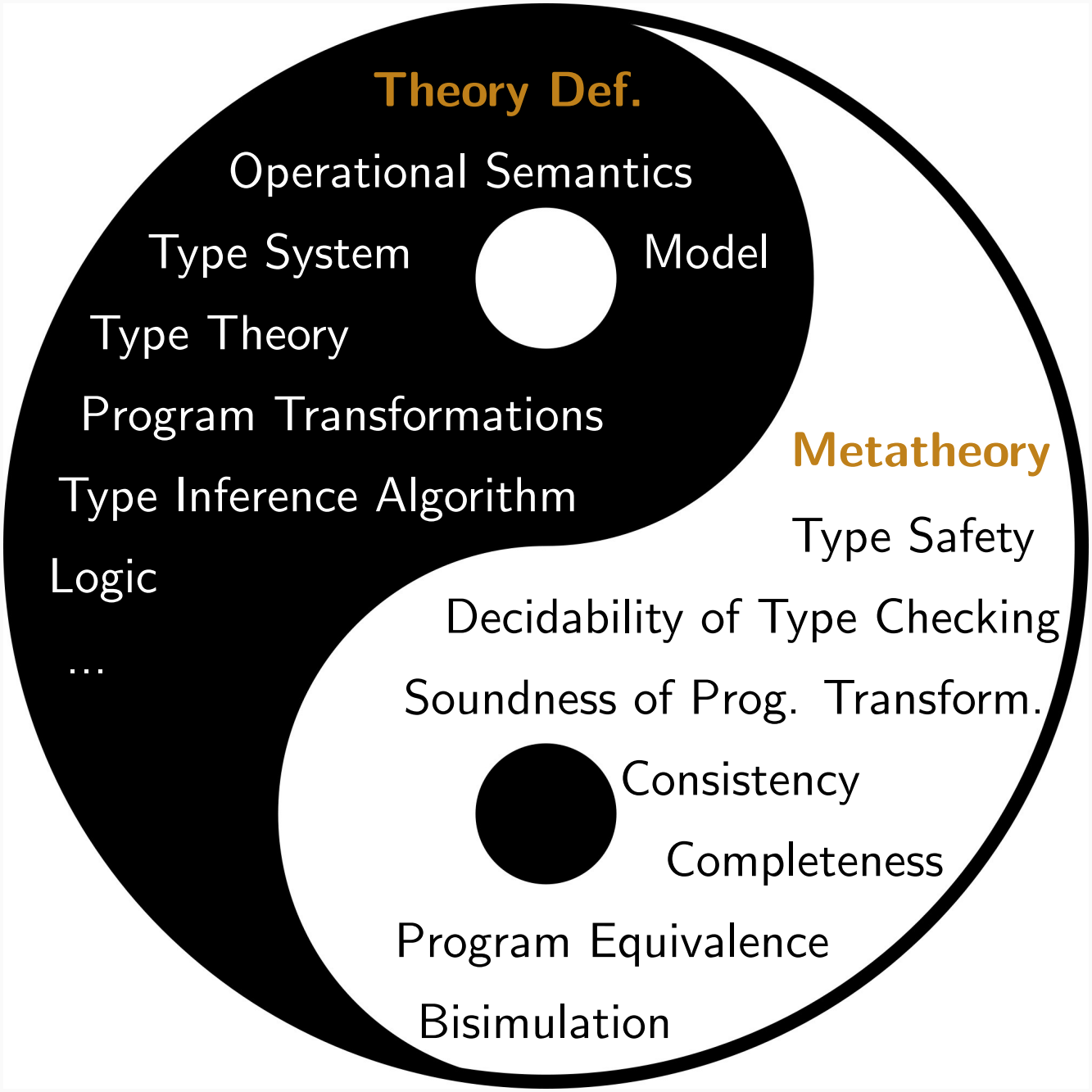
Mechanizing Metatheory – The Next Chapter

Brigitte Pientka

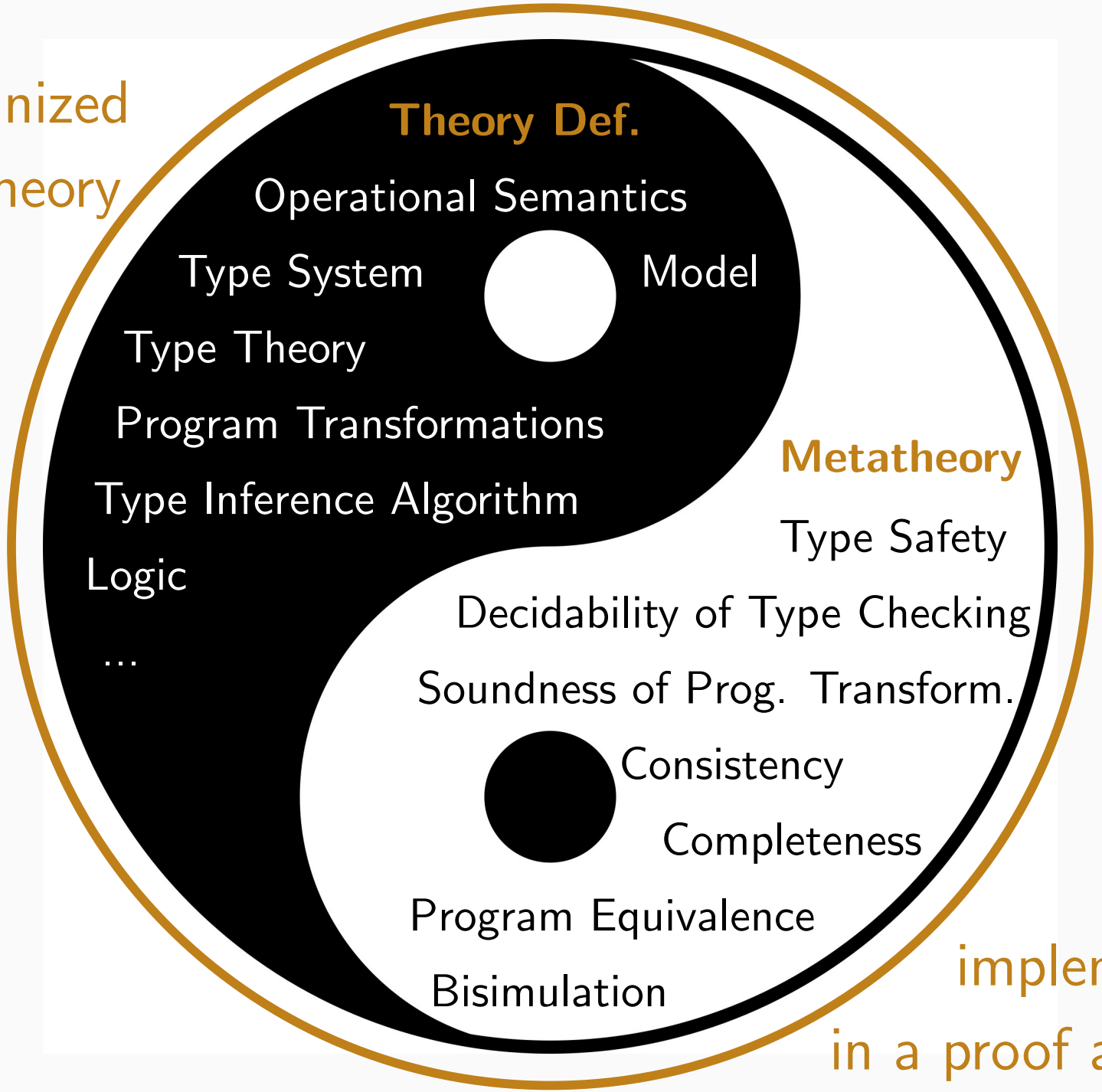
McGill University
Montreal, Canada

What is mechanized 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



implemented in a proof assistant

1. Mechanizing formal systems together with proofs about them is the talk of the town.

Everybody talks about it.

Mechanized Metatheory for the Masses: The PoplMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

POPLMark Reloaded: Mechanizing Proofs by Logical Relations

ANDREAS ABEL
Department of Computer Science and Engineering, Gothenburg University, Sweden

GUILLAUME ALLAIS
iCIS, Radboud University, Nijmegen, Netherlands

ALIYA HAMEER and BRIGITTE PIENTKA
School of Computer Science, McGill University, Canada

ALBERTO MOMIGLIANO
Department of Computer Science, University of Milan, Italy

STEVEN SCHÄFER and KATHRIN STARK

SOFTWARE FOUNDATIONS

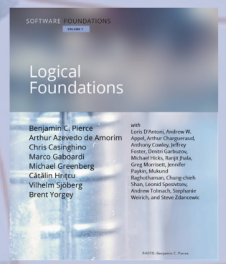
The Software Foundations series is a broad introduction to the mathematical underpinnings of reliable software.

The principal novelty of the series is that every detail is one hundred percent formalized and machine-checked: the entire text of each volume, including the exercises, is literally a "proof script" for the Coq proof assistant.

The exposition is intended for a broad range of readers, from advanced undergraduates to PhD students and researchers. No specific background in logic or programming languages is assumed, though a degree of mathematical maturity is helpful. A one-semester course can expect to cover *Logical Foundations* plus most of *Programming Language Foundations* or *Verified Functional Algorithms*, or selections from both.

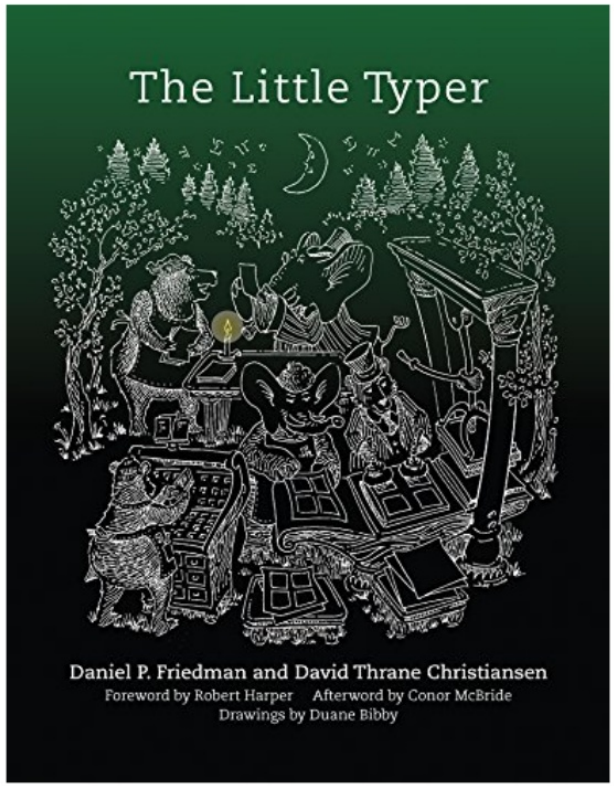
Volume 1

Logical Foundations **is the entry-point to the series. It covers functional programming, basic concepts of logic, computer-assisted theorem proving, and Coq.**



Volume 2

Programming Language Foundations **surveys the theory of programming languages, including operational semantics, Hoare logic, and static type systems.**



From Simple Types To Dependent Types

2. Mechanizing formal systems together with proofs about them establishes trust.

2. Mechanizing formal systems together with proofs about them establishes trust... and avoids flaws.

Programs go wrong.

Programs go wrong.

Testing correctness of C compilers [Vu et al. PLDI'14]:

- GCC and LLVM had over 195 bugs
- Compcert the only compiler where no bugs were found

"This is a strong testimony to the promise and quality of verified compilers."

[Vu et al. PLDI'14]

Programming lang. designs and implementations go wrong.

Type Safety of Java (20 years ago)

Java is Type Safe — Probably

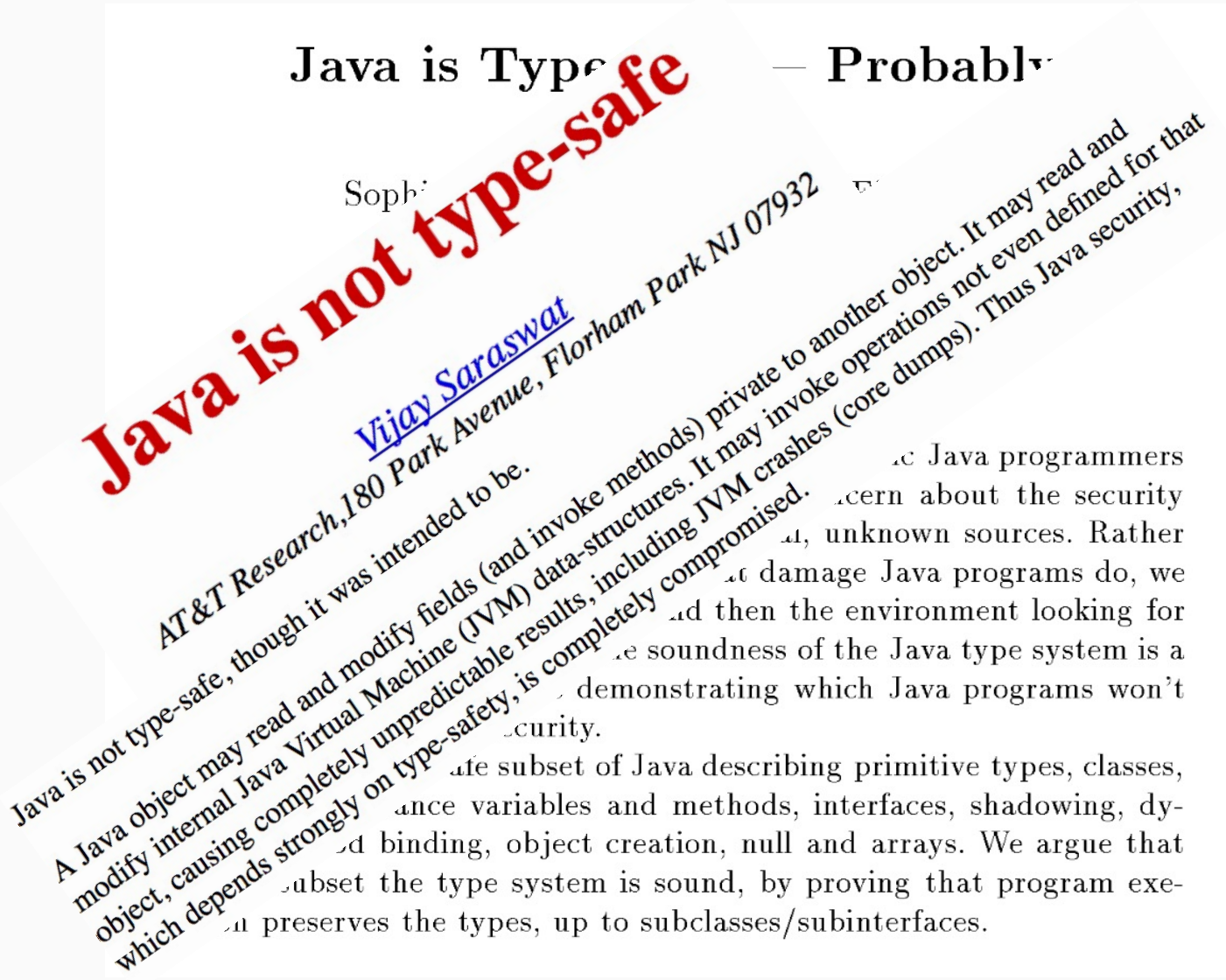
Sophia Drossopoulou and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
email: sd and se @doc.ic.ac.uk

Abstract. Amidst rocketing numbers of enthusiastic Java programmers and internet applet users, there is growing concern about the security of executing Java code produced by external, unknown sources. Rather than waiting to find out empirically what damage Java programs do, we aim to examine first the language and then the environment looking for points of weakness. A proof of the soundness of the Java type system is a first, necessary step towards demonstrating which Java programs won't compromise computer security.
We consider a type safe subset of Java describing primitive types, classes, inheritance, instance variables and methods, interfaces, shadowing, dynamic method binding, object creation, null and arrays. We argue that for this subset the type system is sound, by proving that program execution preserves the types, up to subclasses/subinterfaces.

Programming lang. designs and implementations go wrong.

Type Safety of Java (20 years ago)



Java is Type — Probably

Java is not type-safe

Soph[illegible]

Vijay Saraswat

AT&T Research, 180 Park Avenue, Florham Park NJ 07932

Java is not type-safe, though it was intended to be.

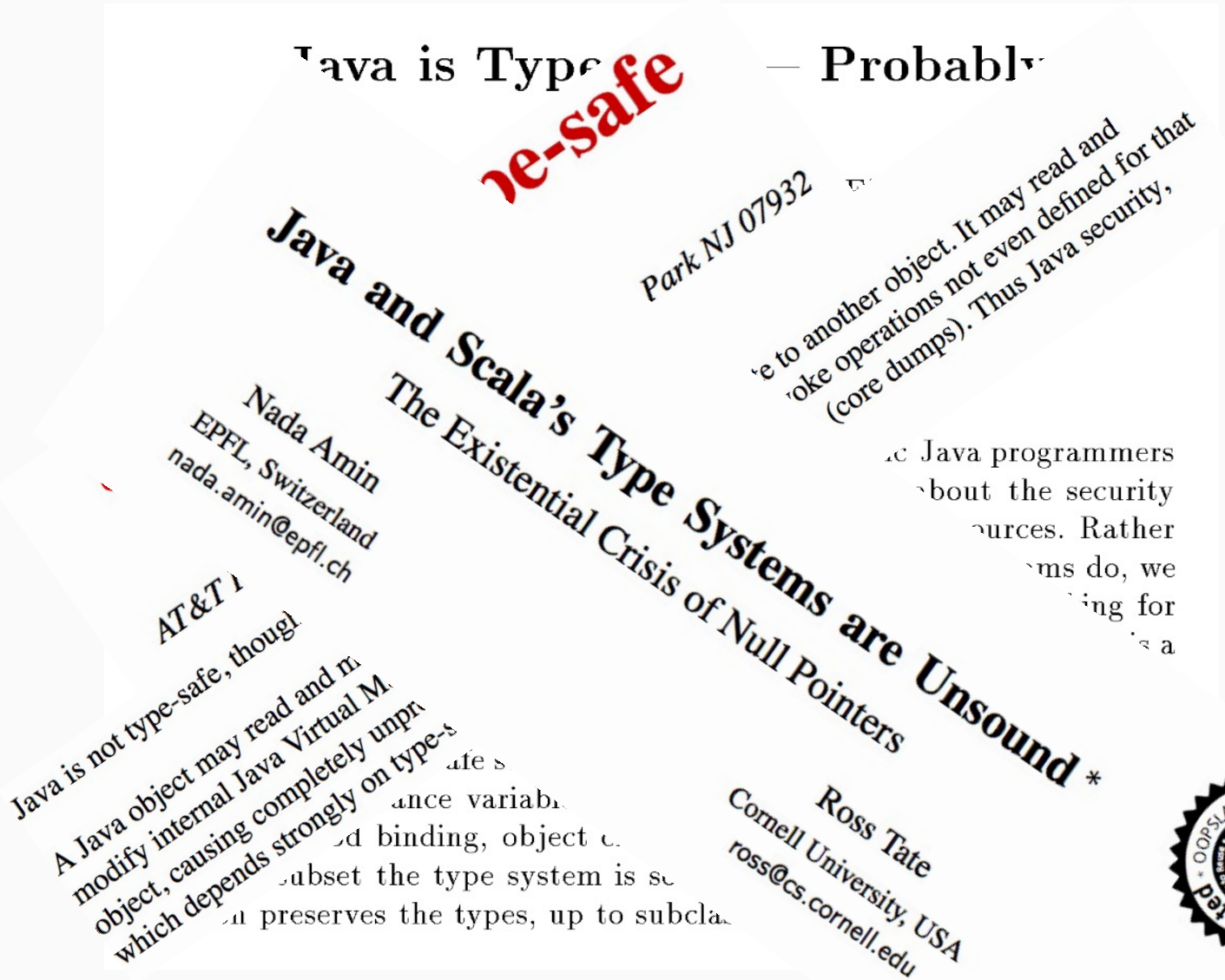
A Java object may read and modify fields (and invoke methods) private to another object. It may read and modify internal Java Virtual Machine (JVM) data-structures. It may invoke operations not even defined for that object, causing completely unpredictable results, including JVM crashes (core dumps). Thus Java security, which depends strongly on type-safety, is completely compromised.

[illegible] Java programmers [illegible]cern about the security [illegible], unknown sources. Rather [illegible] damage Java programs do, we [illegible]d then the environment looking for [illegible]e soundness of the Java type system is a [illegible] demonstrating which Java programs won't [illegible]curity.

[illegible]afe subset of Java describing primitive types, classes, [illegible]ance variables and methods, interfaces, shadowing, dy- [illegible]d binding, object creation, null and arrays. We argue that [illegible]ubset the type system is sound, by proving that program exe- [illegible] preserves the types, up to subclasses/subinterfaces.

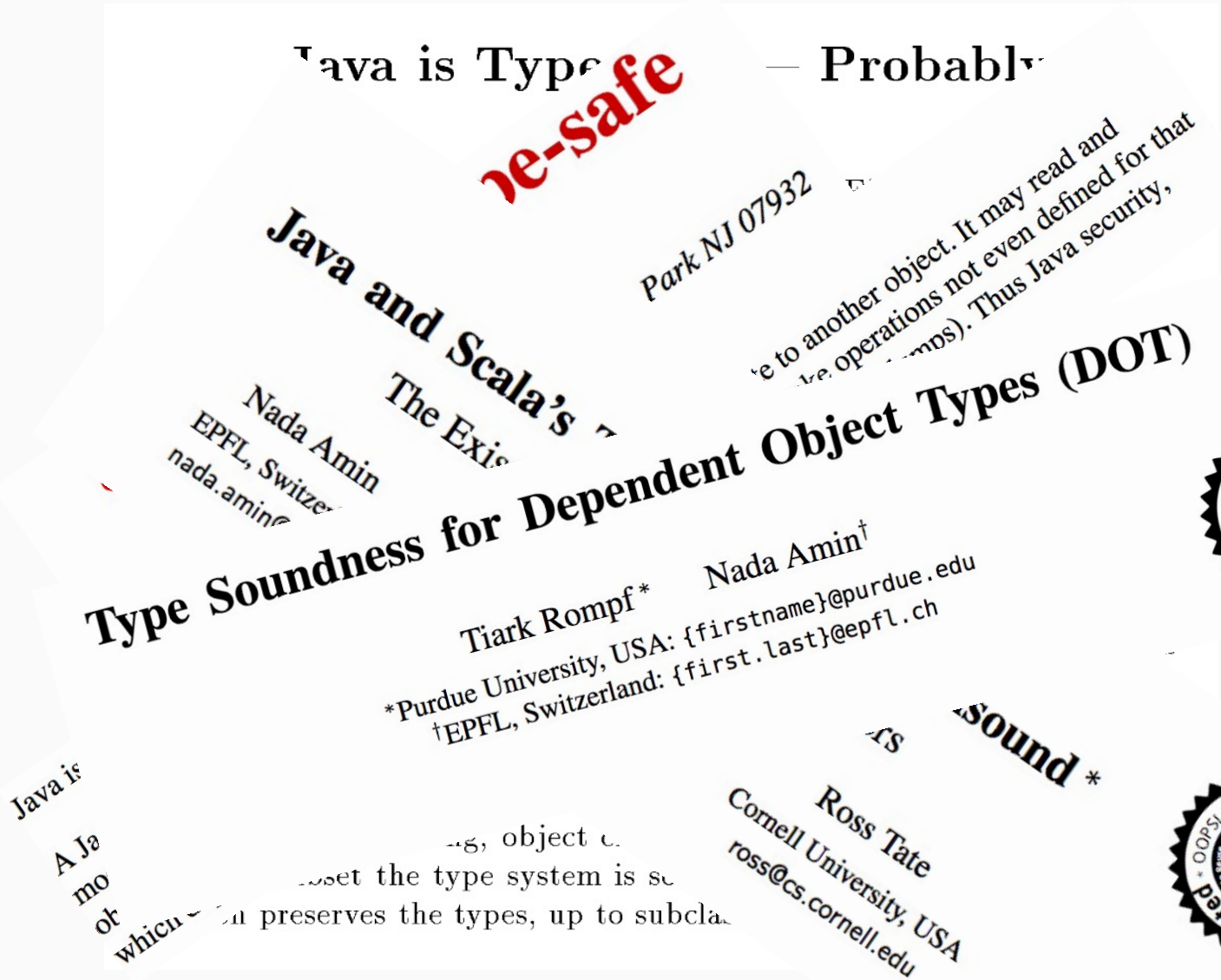
Programming lang. designs and implementations go wrong.

Type Safety of Java and Scala (20 years later)



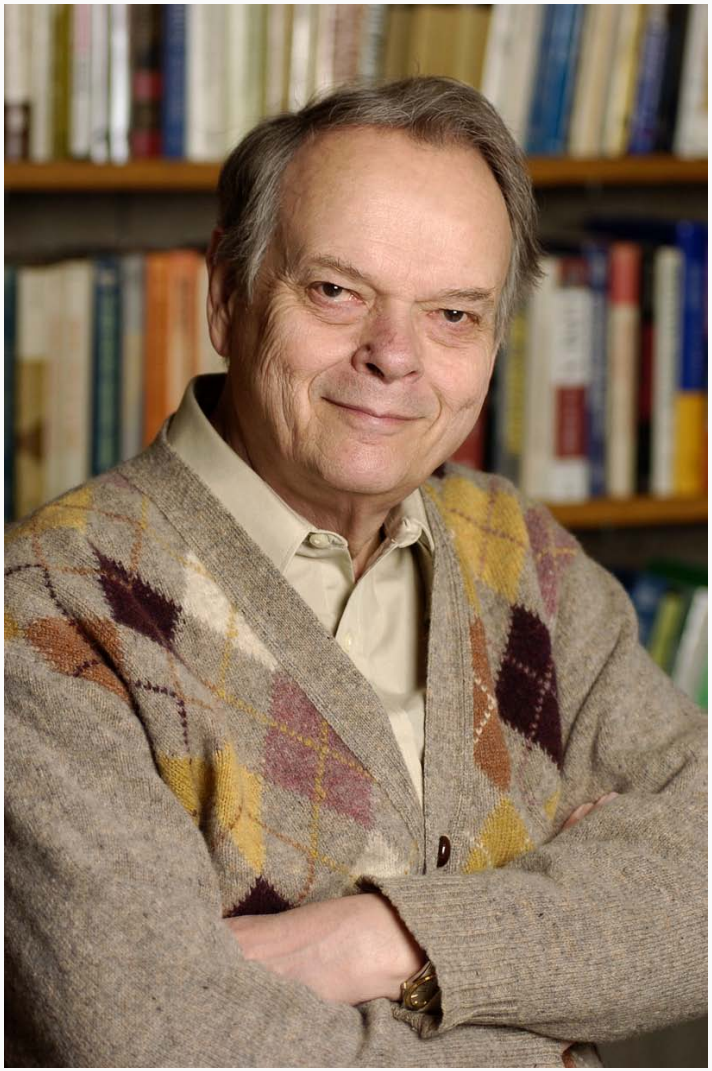
Programming lang. designs and implementations go wrong.

Type Safety of Java and Scala (20 years later)



Why is it hard to get theories and implementations right?

It's a tricky business.



"The truth of the matter is that putting languages together is a very tricky business. When one attempts to combine language concepts, unexpected and counterintuitive interactions arise."

- J. Reynolds

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

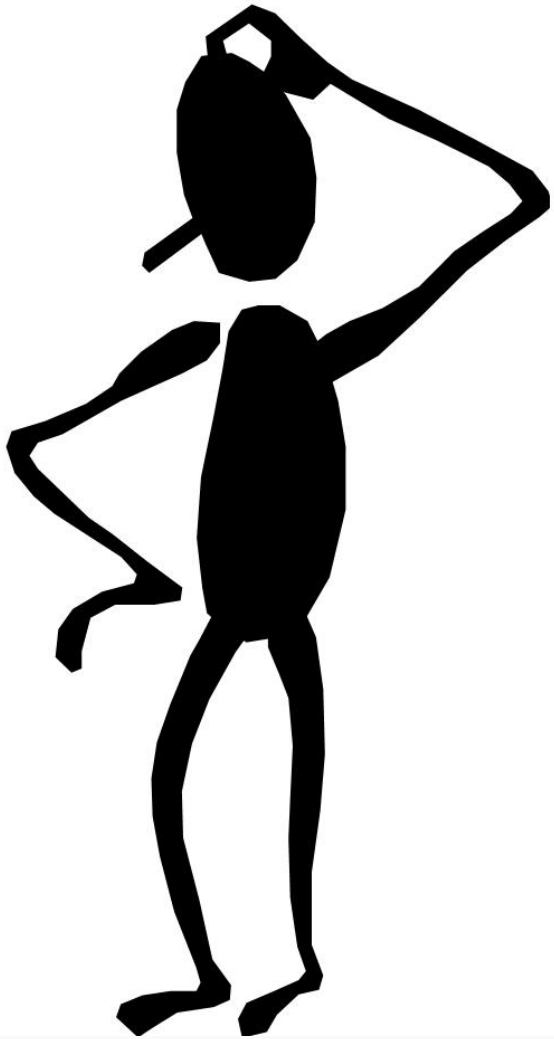
In a proof assistant:

- 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`

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

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 Binding Structures using LF Functions

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

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

C…

tl…

$\forall\alpha$…

```
…λx.x)
…a.all λb.arr a b
```

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

An Attempt in OCaml

OCaml

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

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

An Attempt in OCaml

OCaml

```
1 type tm = Lam of (tm -> tm)
2 let apply = function (Lam f) -> f
3 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

```
1 type tm = Lam of (tm -> tm)
2 let apply = function (Lam f) -> f
3 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

```
1 type tm = Lam of (tm -> tm)
2 let apply = function (Lam f) -> f
3 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    m : 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

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

"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]*

Back in 2008...

LF and Holes in HOAS trees – Revisited

In LF (Meta Lang.)

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

`lam` λ`x. lam` λ`f.` $\boxed{\texttt{app f x}}$

LF Typing Judgment:

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

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:

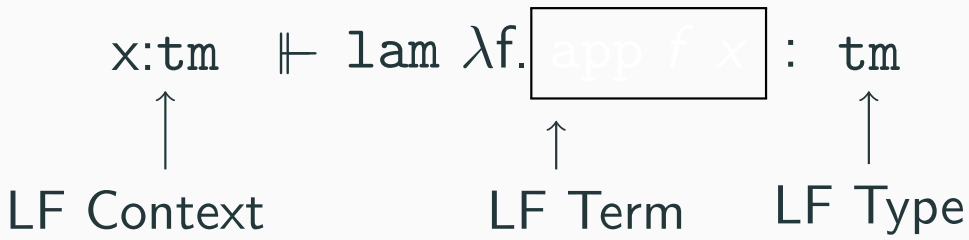
$$\texttt{x}{:}\texttt{tm} \ \vdash\ \texttt{lam}\,\lambda\texttt{f}.\texttt{app}\ \texttt{f}\ \texttt{x} : \texttt{tm}$$

LF Context — LF Term — LF Type

LF and Holes in HOAS trees – Revisited

In LF (Meta Lang.)
`lam` λ`x.` `lam` λ`f.app f x`
`lam` λ`x.` `lam` λ`f.` `app f x`

LF Typing Judgment:



LF and Holes in HOAS trees – Revisited

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

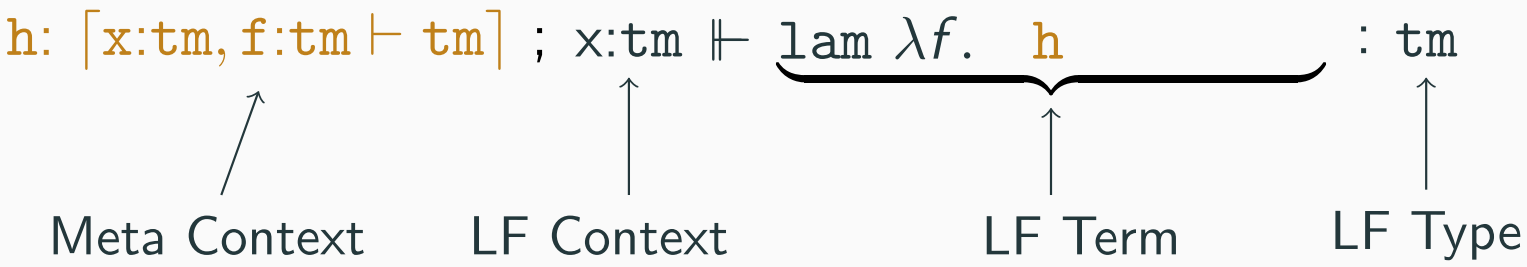
LF Typing Judgment:

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

LF Context — LF Term — LF Type

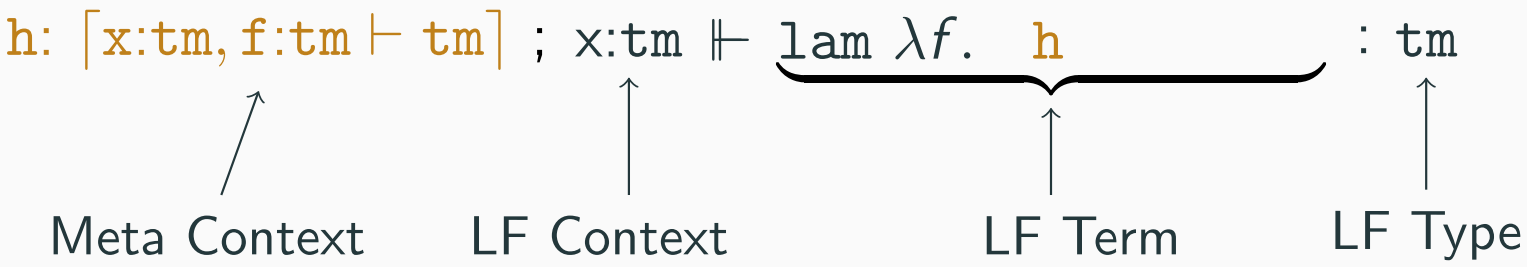
What is the type of $\boxed{\texttt{app f x}}$? – Its type is $\lceil \texttt{x:tm}, \texttt{f:tm} \vdash \texttt{tm} \rceil$.

Contextual Types [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 types (⊢) reify LF typing derivations (⊩)

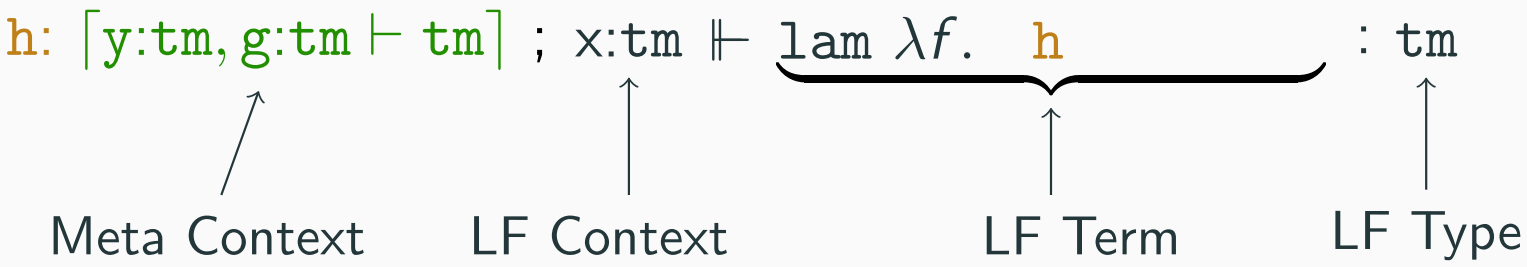
Contextual Types [Nanevski, Pfenning, Pientka'08]



- 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 types (⊢) reify LF typing derivations (⊩)

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

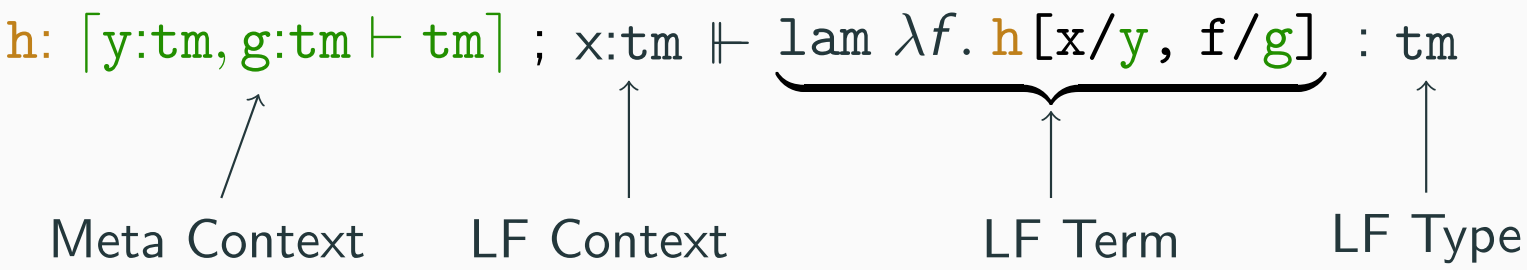
Contextual Types [Nanevski, Pfenning, Pientka'08]



- 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⌉
- Contextual types (⊢) reify LF typing derivations (⊩)

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

Contextual Types [Nanevski, Pfenning, Pientka'08]

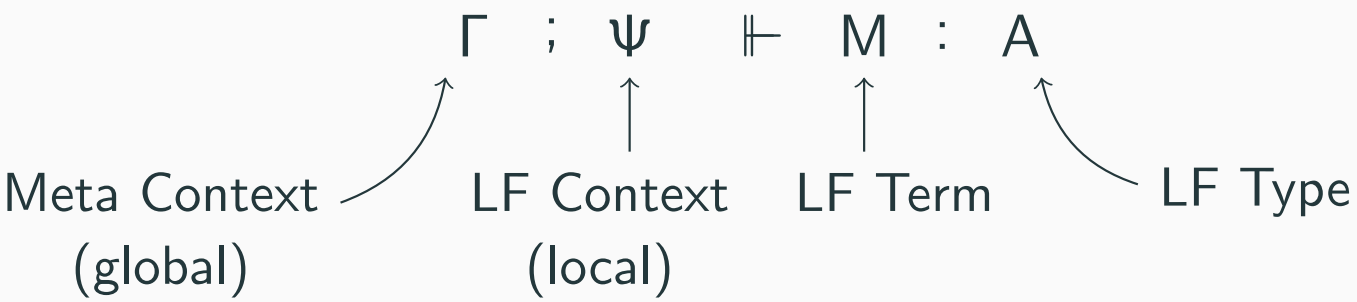


- h is a contextual variable
- It has the contextual type ⌈y:tm, g:tm ⊢ tm⌉
- It can be instantiated with a contextual term ⌈y,g⊢ app g y⌉
- Contextual types (⊢) reify LF typing derivations (⊩)

WAIT! . . . whatever we plug in for 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

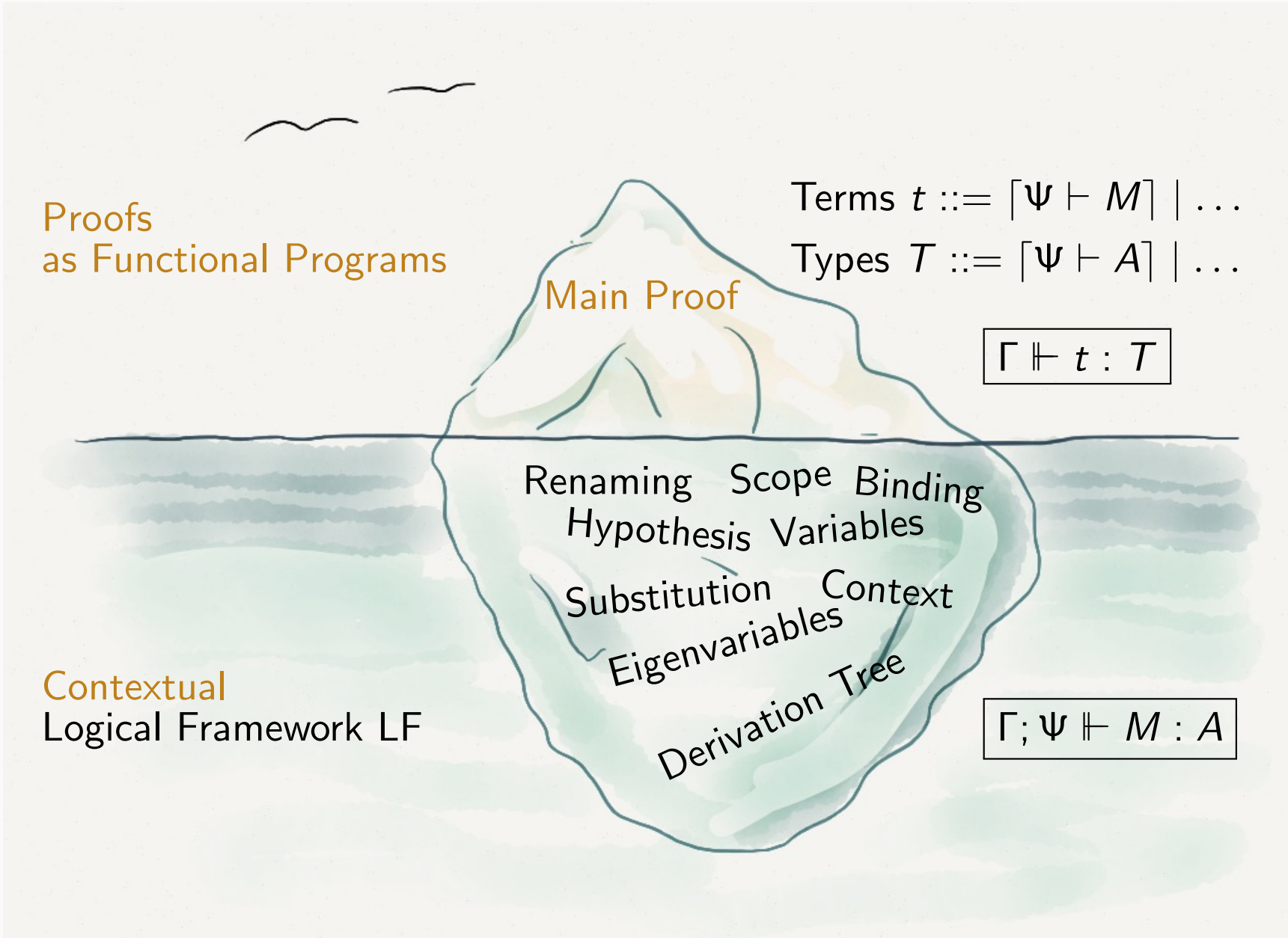
$$\frac{x{:}A \in \Psi}{\Gamma; \Psi \Vdash x : A}$$

Contextual Variable

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

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

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



Revisiting the program `size`

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

Revisiting the program `size`

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

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

- **Coinductive proofs:**

  Bisimulation proof using Howe's Method [MSCS'18]

  Joint work with D. Thibodeau and A. Momigliano

Remember the PhD student who mechanized strong normalization for STLC in Coq using de Bruijn?

POPLMark Reloaded

4 Beyond the Challenge

The PoplMark Challenge is not meant to be exhaustive: other aspects of programming language theory raise formalization difficulties that are interestingly different from the problems we have proposed—to name a few: more complex binding constructs such as mutually recursive definitions, logical relations proofs, coinductive simulation arguments, undecidability results, and linear handling of

Strong Normalization for STLC using Kripke-style Logical Relations Joint work with A. Abel, G. Allais, A. Hameer, A. Momigliano, S. Schäfer, K. Stark

- Easily accessible problem, while still being worthwhile
- Survey the state of the art
- Compare proof assistants
- Encourage development to make them more robust

What I Learned

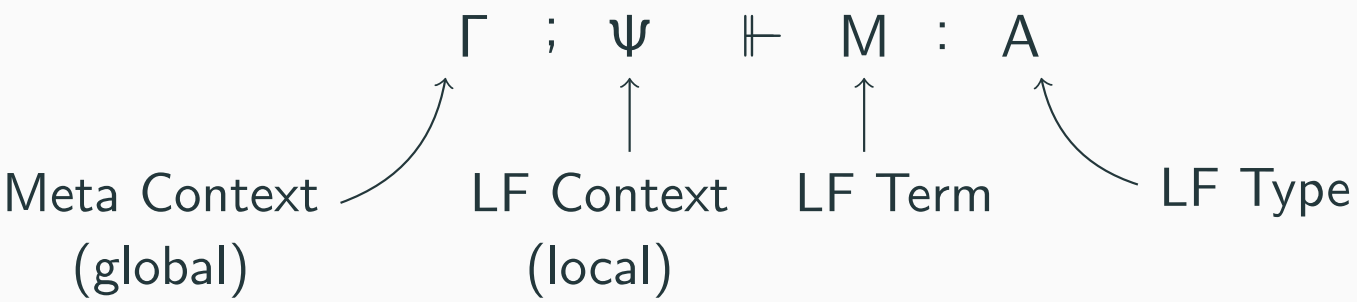
Lesson 1: Choosing an inductive definition for SN makes proofs modular and simpler – on paper and in mechanizations.

Lesson 2: Beluga exploits high-level abstractions and primitives (HOAS, contexts, substitutions, renamings, etc.) leading to a compact implementation (416 LOC).

Lesson 3: Contextual types provide an abstract, conceptual view of syntax trees within a context of assumptions.

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

A Type Theory for Defining Logics and Proofs



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

Contextual Variable Rule

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

A Type Theory for Defining Logics and Proofs

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

Meta Context (global) — Γ

LF Context (local) — Ψ

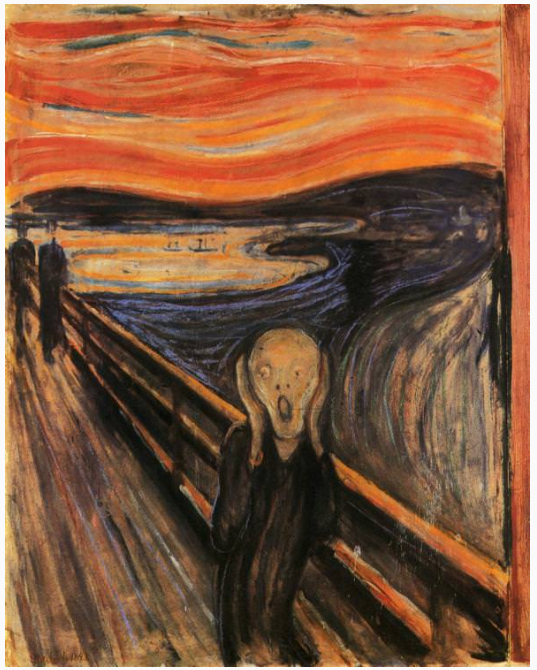
LF Term — M

LF Type — A

What if we did?

Rule for Embedding Computations

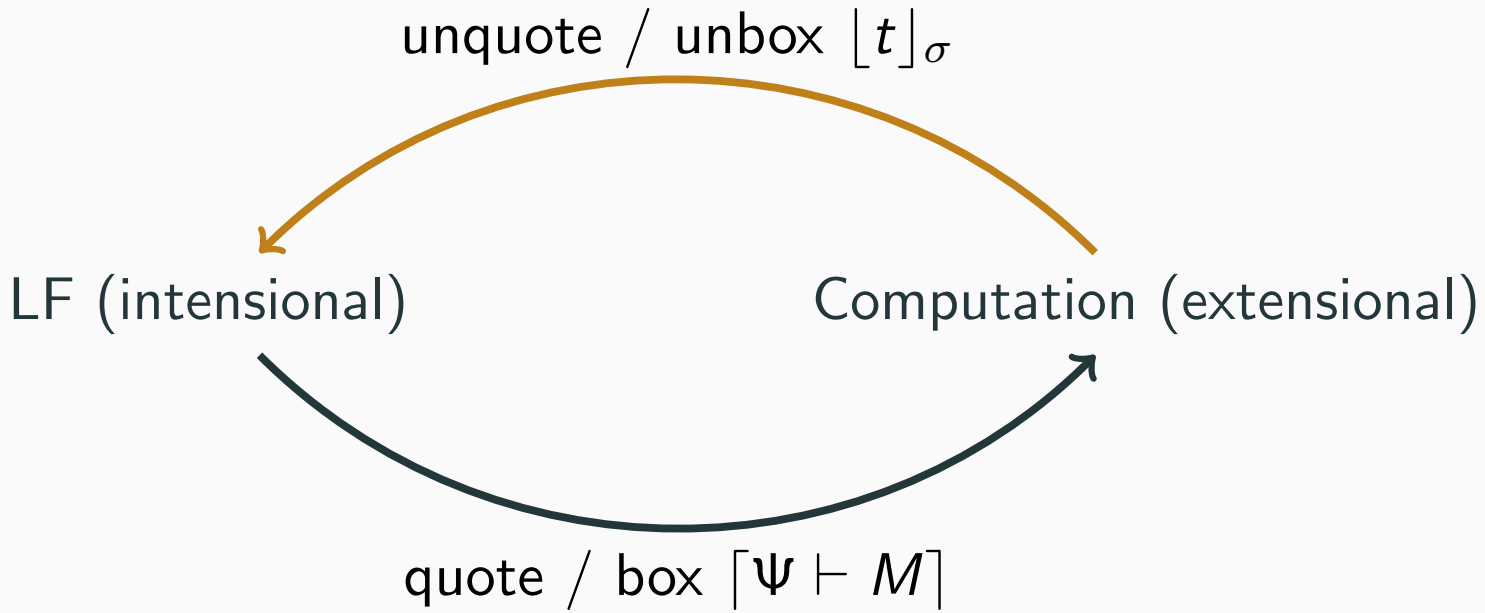
$$\frac{\Gamma \Vdash t : \lceil \Phi \vdash A \rceil \qquad \Gamma; \Psi \Vdash \sigma : \Phi}{\Gamma; \Psi \Vdash \underbrace{\lfloor t \rfloor_\sigma}_{\text{Closure}} \;:\; \underbrace{[\sigma]A}_{\text{Apply subst. } \sigma}}$$



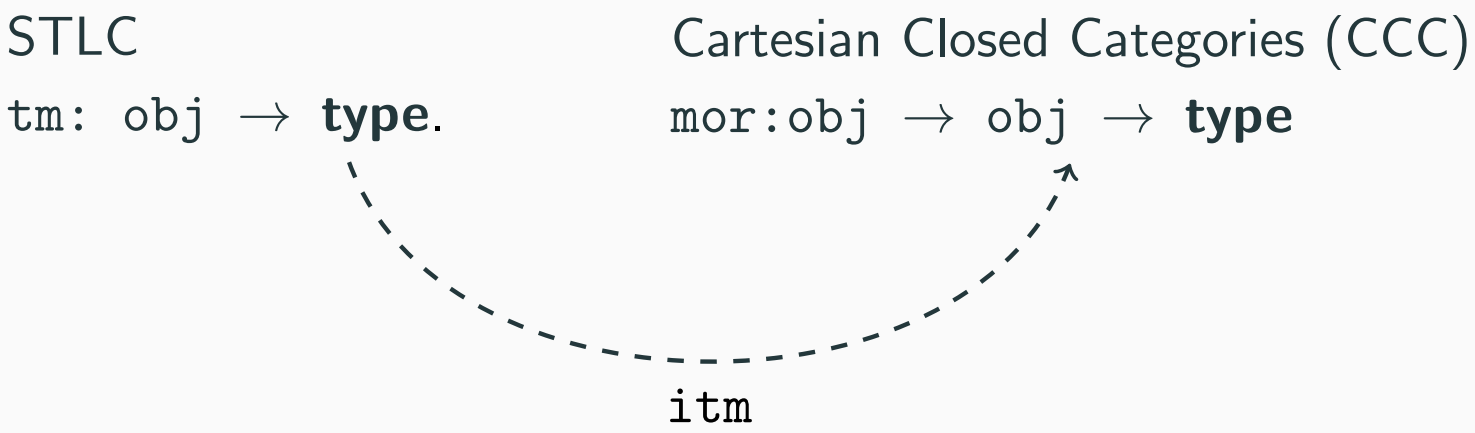
Cocon: A Type Theory for Defining Logics

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

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



Sketch: Translation Between STLC and CCC



Translate an LF context γ to cross product: `ictx:Πγ:ctx.⌈ ⊢ obj⌉`

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

Translate STLC to CCC

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

Bridging the Gap between LF and Martin Löf Type Theory

What's Next?

Theory

- Decidable equality
- Categorical semantics
- ...

Implementation and Case Studies

- Build an extension to Coq/Agda/Beluga
- Case studies: Equivalence of STLC and CCC
- Meta-Programming (Tactics)
- Compilation
- Proof Search
- ...

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: Abstractions for variable binding, contexts, and substitution, etc. are useful when we mechanize metatheory.

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: Abstractions for variable binding, contexts, and substitution, etc. are useful when we mechanize metatheory.

Last but not least: There are many other abstractions and primitives we should explore: heaps, linearity, resources, ...