# Programming with Proofs and Explicit Contexts – Revisited –

Brigitte Pientka

McGill University, Montreal

beluga

Joint work with J. Dunfield (Queens University, Kingston)

# How to program and reason with formal systems and proofs?

# How to program and reason with formal systems?

- Formal systems (given via axioms and inference rules) play an important role when designing and implementing software.
  Type systems; Evaluation; Program Transformations;Logics; etc.
- Mechanizing properties about formal systems establishes trust and avoids flaws. Type preservation; Compiler correctness; Cut-elimination; Church-Rosser property; etc.

# Underlying Motivation

- Abstract over common operations
- Support common features uniformly

*"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"* *B. Liskov [1974]*

## 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

1989 — *F. Pfenning*: Elf: A language for Logic Definition and Verified Meta-Programming, LICS'89

- Dependently Typed Lambda Calculus ($\lambda^{\Pi}$) 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

**Uniformly handle:**

**– Bound Variables,**

**– Hypothetical and Parametric Assumptions**

# Step 1: Representing Types and Terms in LF

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

# Step 1: Representing Types and Terms in LF

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

## LF Representation

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

| On Paper (Object Language) | In LF (Meta Language) |
|---|---|
| lam $x$:nat.$x$ (Identity) | `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. lamt $f$:nat$\Rightarrow$nat.app $f$ $x$ | `lam nat λx.lam (arr nat nat) λf.app f x` |

- Higher-order Abstract Syntax (HOAS) :
  Uniformly model binding structures in Object Language with (intensional) functions in LF
- Inherit $\alpha$-renaming and single substitutions

# Step 2: Representation of Typing Rules in LF

Typing Rules

$$\frac{M : A \Rightarrow B \quad N : A}{\mathsf{app}\ M\ N : B}\ \textsc{T-App} \qquad \frac{\begin{array}{c}\overline{x : A}^{\,u} \\ \vdots \\ M : B\end{array}}{\mathsf{lam}\, x{:}A.M : A \Rightarrow B}\ \textsc{T-Lam}^{x,u}$$

# Step 2: Representation of Typing Rules in LF

Typing Rules

$$\frac{M : A \Rightarrow B \quad N : A}{\mathsf{app}\ M\ N : B}\ \textsc{T-App} \qquad \frac{\begin{array}{c}\overline{x : A}^{\,u} \\ \vdots \\ M : B\end{array}}{\mathsf{lam}\ x{:}A.M : A \Rightarrow B}\ \textsc{T-Lam}^{x,u}$$

### LF Representation

```
of: tm → tp → type.
t_app: of M (arr A B) → of N A
    → of (app M N) B.
t_lam:(Πx:tm.of x A → of (M x) B)
    → of (lam A M) (arr A B).
```

- Hypothetical derivations are represented as LF functions (simple type)
- Parametric derivations are represented as LF functions (dependent type)

| On Paper (Object Language) | In LF (Meta Language) |
|---|---|
| $$\cfrac{\cfrac{\begin{array}{c}\overline{x : \mathsf{nat}}^{\,u} \quad \overline{y : \mathsf{nat}}^{\,v} \\ \mathcal{D} \\ y : \mathsf{nat}\end{array}}{(\mathsf{lam}\ y{:}\mathsf{nat}.y) : (\mathsf{nat} \Rightarrow \mathsf{nat})}\ \mathsf{t\_lam}^{y,v}}{(\mathsf{lam}\ x{:}\mathsf{nat}.\mathsf{lam}\ y{:}\mathsf{nat}.y) : (\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat})}\ \mathsf{t\_lam}^{x,u}$$ | `t_lam λx.λu.t_lam λy.λv.D` |

**How to reason inductively?**

– **LF definitions are not inductive**

– **We must handle "open" objects**

**Preservation: If** $M : A$ **and** $M \longrightarrow N$ **then** $N : A$.

**Uniqueness: If** $\Gamma \vdash M : A$ **and** $\Gamma \vdash M : B$ **then** $A{=}B$.

# Back in the 90s ...

1997 *R. McDowell and D. Miller*:A Logic for Reasoning with Higher-Order Abstract Syntax. LICS 1997
(Reason about HOAS indirectly; closed HOAS objects)

1998 *C. Schürmann and F. Pfenning*: Automated Theorem Proving in a Simple Meta-Logic for LF, CADE'98
(No proof witnesses)

1999 *F. Pfenning and C. Schürmann:* Twelf — A Meta-Logical Framework for Deductive Systems, CADE'99 (Regular worlds; proofs as relations with LF.)

*"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

2008 — *A. Nanevski, F. Pfenning, B. Pientka*: Contextual Modal Type Theory, ACM TOCL 2008

2008 — *B. Pientka*: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions, POPL'08 — simply-typed

2008 — *B. Pientka and J. Dunfield:* Programming with proofs and explicit contexts, PPDP'08 — dependently-typed

Key Observation: Characterize LF object together with the LF context

- `lam nat` $\lambda$`x.lam (arr nat nat)`$\lambda$`f.` $\boxed{\texttt{app f x}}$

  $\boxed{\texttt{app f x}}$ has LF type `tm` in the LF context `x:tm, f:tm`
- `t_lam` $\lambda$`x.`$\lambda$`u.` $\boxed{\texttt{D}}$

  $\boxed{\texttt{D}}$ has LF type `of (lam nat` $\lambda$`y.x) (arr nat nat)` in LF context `x:tm,u:of x nat`.

# Back in 2008

2008 — *A. Nanevski, F. Pfenning, B. Pientka*: Contextual Modal Type Theory, ACM TOCL 2008

2008 — *B. Pientka*: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions, POPL'08 — simply-typed

2008 — *B. Pientka and J. Dunfield:* Programming with proofs and explicit contexts, PPDP'08 — dependently-typed

**Key Observation:** Characterize LF object together with the LF context

- `lam nat` $\lambda$`x.lam (arr nat nat)`$\lambda$`f.` $\boxed{\texttt{app f x}}$

  $\boxed{\texttt{app f x}}$ has contextual LF type `[x:tm, f:tm` $\vdash$ `tm]`

- `t_lam`$\lambda$`x.`$\lambda$`u.`$\boxed{\texttt{D}}$

  $\boxed{\texttt{D}}$ has contextual LF type `[x:tm,u:of x nat` $\vdash$ `of (lam nat` $\lambda$`y.x) (arr nat nat)]`.

# Back in 2008

2008 — *A. Nanevski, F. Pfenning, B. Pientka*: Contextual Modal Type Theory, ACM TOCL 2008

2008 — *B. Pientka*: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions, POPL'08 — simply-typed

2008 — *B. Pientka and J. Dunfield:* Programming with proofs and explicit contexts, PPDP'08 — dependently-typed

**Key Observation:** Characterize LF object together with the LF context

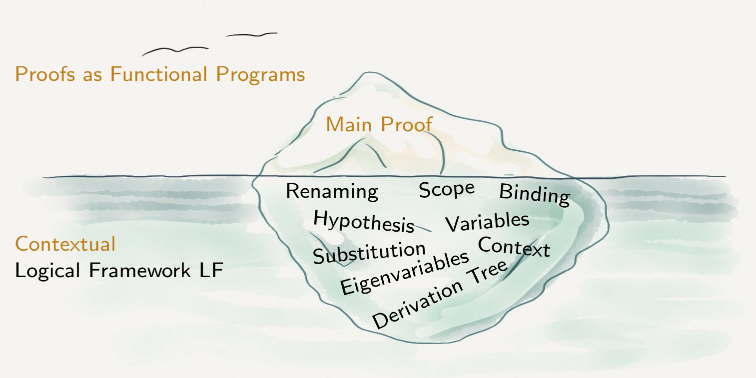
- `lam nat` $\lambda$`x.lam (arr nat nat)`$\lambda$`f.` $\boxed{\texttt{app f x}}$

  $\boxed{\texttt{app f x}}$ has contextual LF type `[x:tm, f:tm` $\vdash$ `tm]`
- `t_lam`$\lambda$`x.`$\lambda$`u.` $\boxed{\texttt{D}}$

  $\boxed{\texttt{D}}$ has contextual LF type `[x:tm,u:of x nat` $\vdash$ `of (lam nat` $\lambda$`y.x) (arr nat nat)]`.

**Key Observation:** Abstract over LF contexts to enable recursion

# The tip of the iceberg: Beluga




*"We may think of [the] proof as an iceberg. In the top of it, we find what we usually consider the real proof; underwater, the most of the matter, consisting of all mathematical preliminaries a reader must know in order to understand what is going on."* *S. Berardi [1990]*

# Step 2a: Theorem as Type

**Theorem: Type Uniqueness**

If $\mathcal{D} :: \Gamma \vdash M : A$ and $\mathcal{C} :: \Gamma \vdash M : B$ then $\mathcal{E} :: A = B$.

## Step 2a: Theorem as Type

**Theorem: Type Uniqueness**

If $\mathcal{D} :: \Gamma \vdash M : A$ and $\mathcal{C} :: \Gamma \vdash M : B$ then $\mathcal{E} :: A = B$.

is represented as

**Computation Level Type for function `unique`**

$\Pi\gamma$`:ctx.[`$\gamma \vdash$ `of M A ]` $\rightarrow$ `[`$\gamma \vdash$ `of M B ]` $\rightarrow$ `[` $\vdash$ `eq A B]`

- Parameterize over and distinsuigh between contexts
- Contexts are structured sequences
- Contexts are classified by context schemas

  ```
  schema ctx =  some [t:tp] block x:tm, u:of x t;
  ```

# Step 2a: Theorem as Type

**Theorem: Type Uniqueness**

If $\mathcal{D} :: \Gamma \vdash M : A$ and $\mathcal{C} :: \Gamma \vdash M : B$ then $\mathcal{E} :: A = B$.

is represented as

**Computation Level Type for function `unique`**

$\Pi\gamma$`:ctx.[`$\gamma \vdash$ `of M A[]]` $\to$ `[`$\gamma \vdash$ `of M B[]]` $\to$ `[` $\vdash$ `eq A B]`

- Parameterize over and distinsuigh between contexts
- Contexts are structured sequences
- Contexts are classified by context schemas
  `schema ctx = some [t:tp] block x:tm, u:of x t;`
- `M` is a term that depends on $\gamma$; it has type `[`$\gamma \vdash$ `tm]`

  `A` and `B` are types that are closed; they have type `[` $\vdash$ `tp]`

  **Fact**: All meta-variables are associated with a substitution.

  $\leadsto$ `M` is implicitely associated with the identity substitution

  $\leadsto$ `A` and `B` are associated with a weakening substitution

# Intrinsic Support for Contexts

```
schema ctx =  some [t:tp] block x:tm, u:of x t;
```

- The context $x$ : nat, $y$ : nat $\Rightarrow$ nat   is represented as
  `b1:block(x:tm,u:of x nat),`
  `b2:block(y:tm,v:of y (arr nat nat))`
- Well-formedness: `b1:block (x:tm,u:of y nat)` is ill-formed.
  `x:tm, y:tm, u:of x nat` is ill-formed.
- Projections (`b1.1` or `b1.x`) to access components of a block
- Declarations are unique: `b1` is different from `b2`
  `b1.x` is different from `b2.x`
- Later declarations overshadow earlier ones
- Support Weakening and Substitution lemmas

# Step 2b: Proofs as Programs

```
rec unique:Πγ:ctx. Π A:[tp].Π B:[tp].Π M:[γ ⊢ tm].
          [γ ⊢ of M A[]] → [γ ⊢ of M B[]] → [ ⊢ eq A B] =
```

# Step 2b: Proofs as Programs

```
rec unique:Πγ:ctx. Π A:[tp].Π B:[tp].Π M:[γ ⊢ tm].
          [γ ⊢ of M A[]] → [γ ⊢ of M B[]] → [ ⊢ eq A B] =
fn d ⇒ fn c ⇒ case d of
```

# Step 2b: Proofs as Programs

```
rec unique:Πγ:ctx. Π A:[tp].Π B:[tp].Π M:[γ ⊢ tm].
          [γ ⊢ of M A[]] → [γ ⊢ of M B[]] → [ ⊢ eq A B] =
fn d ⇒ fn c ⇒ case d of
| [γ ⊢ t_app D1 D2] ⇒                          % Application Case
  let[γ ⊢ t_app C1 C2] = c in
  let[ ⊢ ref] = unique [γ ⊢ D1] [γ ⊢ C1] in
     [ ⊢ ref]
```

# Step 2b: Proofs as Programs

```
rec unique:Πγ:ctx. Π A:[tp].Π B:[tp].Π M:[γ ⊢ tm].
          [γ ⊢ of M A[]] → [γ ⊢ of M B[]] → [ ⊢ eq A B] =
fn d ⇒ fn c ⇒ case d of
| [γ ⊢ t_app D1 D2] ⇒                       % Application Case
  let[γ ⊢ t_app C1 C2] = c in
  let[ ⊢ ref] = unique [γ ⊢ D1] [γ ⊢ C1] in
     [ ⊢ ref]

| [γ ⊢ t_lam λx.λu. D] ⇒                     % Abstraction Case
  let[γ ⊢ t_lam λx.λu.C] = c in
  let[ ⊢ ref] = unique [γ,b:block x:tm;u:of x _ ⊢ D[b.x, b.u]]
                       [γ,b: _ ⊢ C[b.x, b.u]] in
     [ ⊢ ref]
```

# Step 2b: Proofs as Programs

```
rec unique:Πγ:ctx. Π A:[tp].Π B:[tp].Π M:[γ ⊢ tm].
          [γ ⊢ of M A[]] → [γ ⊢ of M B[]] → [ ⊢ eq A B] =
fn d ⇒ fn c ⇒ case d of
| [γ ⊢ t_app D1 D2] ⇒                      % Application Case
  let[γ ⊢ t_app C1 C2] = c in
  let[ ⊢ ref] = unique [γ ⊢ D1] [γ ⊢ C1] in
     [ ⊢ ref]

| [γ ⊢ t_lam λx.λu. D] ⇒                   % Abstraction Case
  let[γ ⊢ t_lam λx.λu.C] = c in
  let[ ⊢ ref] = unique [γ,b:block x:tm;u:of x _ ⊢ D[b.x, b.u]]
                       [γ,b: _ ⊢ C[b.x, b.u]] in
     [ ⊢ ref]
| [γ ⊢ #q.u] ⇒            % d : of #q.x A    % Assumption Case
  let[γ ⊢ #r.u] = c  in   % c : of #r.x B
     [ ⊢ ref] ;
```

Compact encoding of proofs about derivations as total functions.

# Step 2b: Proofs as Programs

```
rec unique:Πγ:ctx. Π A:[tp].Π B:[tp].Π M:[γ ⊢ tm].
          [γ ⊢ of M A[]] → [γ ⊢ of M B[]] → [ ⊢ eq A B] =
fn d ⇒ fn c ⇒ case d of
| [γ ⊢ t_app D1 D2] ⇒                         % Application Case
  let[γ ⊢ t_app C1 C2] = c in
  let[ ⊢ ref] = unique [γ ⊢ D1] [γ ⊢ C1] in
     [ ⊢ ref]

| [γ ⊢ t_lam λx.λu. D] ⇒                      % Abstraction Case
  let[γ ⊢ t_lam λx.λu.C] = c in
  let[ ⊢ ref] = unique [γ,b:block x:tm;u:of x _ ⊢ D[b.x, b.u]]
                       [γ,b: _ ⊢ C[b.x, b.u]] in
     [ ⊢ ref]
| [γ ⊢ #q.u] ⇒              % d : of #q.x A     % Assumption Case
  let[γ ⊢ #r.u] = c  in     % c : of #r.x B
     [ ⊢ ref] ;
```

Compact encoding of proofs about derivations as total functions.

## Contribution of PPDP'08

- Lays the foundation for viewing inductive proofs about derivations as recursive programs

| On paper | In Beluga |
|---|---|
| Case Analysis | Case Analysis using pattern patching |
| Inversion | Case Analysis using pattern patching |
| IH | Recursive Call |

- *Contextual LF*: Extends LF with meta-variables, parameter variables, variable projections, and first-class context variables.
- Bi-directional type system for contextual LF
- Bi-directional type system for Beluga(computations)
  Dependently type pattern matching using refinements
- Type safety: Preservation and progress

## Since 2008: Beluga has grown up

Theory:

- Normalization proof for BELUGA[TLCA'15,FSCD'18]
- Extension to indexed recursive and stratified types [POPL'12,FSCD'18]
- Extensions to indexed cocrecursive types [ICFP'16]

Implementation:

- First prototype [IJCAR'10]
- Total BELUGA[CADE'15]
- Interactive BELUGA[ongoing, Tutorial at ICFP'18]

Case studies: Certified compiler, Howe's method (coinductive proof), Logical relations proofs (see POPLMark Reloaded [CPP'18])

# What's to come?

COCON: Type theory with contextual types and first-class contexts
– Martin Löf Style –