Programming with Proofs and Explicit Contexts
– Revisited –

Brigitte Pientka
McGill University, Montreal

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...

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

Terms $M ::= x \mid \text{lam } x:A.M \mid \text{app } M N$
Step 1: Representing Types and Terms in LF

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**

<table>
<thead>
<tr>
<th>tp</th>
<th>tm</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>type</td>
</tr>
<tr>
<td>nat: tp</td>
<td>lam: tp (\rightarrow (tm \rightarrow tm) \rightarrow tm)</td>
</tr>
<tr>
<td>arr: tp (\rightarrow tp \rightarrow tp)</td>
<td>app: tm (\rightarrow tm \rightarrow tm)</td>
</tr>
</tbody>
</table>

**On Paper (Object Language)**

| lam $x:$nat.$x$ (Identity) |
| lam $x:$nat. lam $x:$nat $\Rightarrow$ nat.$x$ |
| lam $x:$nat. lam $f:$nat $\Rightarrow$ nat.app $f \ x$ |

**In LF (Meta Language)**

| lam nat $\lambda x.x$ |
| lam nat $\lambda x.$lam (arr nat nat) $\lambda x.x$ |
| lam nat $\lambda x.$lam (arr nat nat) $\lambda f.$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

\[
\begin{align*}
M : A &\Rightarrow B \quad N : A \quad \text{T-APP} \\
\text{app } M \ N &\Rightarrow B
\end{align*}
\]

\[
\frac{x : A}{\vdots}
\]

\[
\frac{M : B}{\text{lam } x : A. M : A \Rightarrow B \quad \text{T-LAM}_{x,u}}
\]

On Paper (Object Language)

In LF (Meta Language)

\[
\begin{align*}
x : \text{nat} &\quad u \\
y : \text{nat} &\quad v \\
D &\quad y : \text{nat} \\
(\text{lam } y : \text{nat}. y) &\quad (\text{nat} \Rightarrow \text{nat})
\end{align*}
\]

\[
\begin{align*}
x : \text{nat} &\quad u \\
y : \text{nat} &\quad v \\
(\text{lam } x : \text{nat}. (\text{lam } y : \text{nat}. y)) &\quad (\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})
\end{align*}
\]

• Hypothetical derivations are represented as LF functions (simple type)
• Parametric derivations are represented as LF functions (dependent type)
Step 2: Representation of Typing Rules in LF

Typing Rules

\[
\begin{align*}
M : A \Rightarrow B & \quad N : A \\
\text{app } M & \quad N : B \quad \text{T-APP} \\
M : B & \quad \text{lam } x : A. M : A \Rightarrow B \quad \text{T-LAM}^{x,u}
\end{align*}
\]

**LF Representation**

- 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)
---|---
\[
\begin{align*}
\dfrac{x : \text{nat} \quad y : \text{nat}}{\text{D}}
\end{align*}
\]  |  \[
\begin{align*}
\text{t_lam}^{y,v} & \quad \text{t_lam}^{x,u} & \quad \text{t_lam} \quad \lambda x. \lambda u. \text{t_lam} \quad \lambda y. \lambda v. D
\end{align*}
\]

\[
\begin{align*}
\dfrac{\text{y : nat}}{(\text{lam } y : \text{nat}. y) : (\text{nat} \Rightarrow \text{nat})}
\end{align*}
\]
How to reason inductively?
- LF definitions are not inductive
- We must handle “open” objects

Preservation: If $M : A$ and $M \rightarrow N$ then $N : A$.

Uniqueness: If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$ then $A = B$. 
Back in the 90s ...

1997  
  (Reason about HOAS indirectly; closed HOAS objects)

1998  
  (No proof witnesses)

1999  
  (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

2008  •  B. Pientka and J. Dunfield: Programming with proofs and explicit contexts, PPDP’08

Key Observation: Characterize LF object together with the LF context

- \text{lam } \text{nat } \lambda x. \text{lam } (\text{arr } \text{nat } \text{nat}) \lambda f. \boxed{\text{app } f \ x}

  \boxed{\text{app } f \ x} \text{ has LF type tm in the LF context } x:tm, f:tm

- \text{t_lam } \lambda x. \lambda u. \boxed{D}

  \boxed{D} \text{ has LF type of } (\text{lam } \text{nat } \lambda y. x) (\text{arr } \text{nat } \text{nat}) \text{ in LF context } x:tm, u:of x \text{ nat.}
Key Observation: Characterize LF object together with the LF context

- \( \text{lam } \text{nat } \lambda x.\text{lam } (\text{arr } \text{nat } \text{nat})\lambda f. \text{app } f \ x \)
  
  \( \text{app } f \ x \) has contextual LF type \([x:tm, f:tm \vdash tm]\)

- \( t_\text{lam} \lambda x.\lambda u. \text{D} \)
  
  \( \text{D} \) has contextual LF type \([x:tm, u:of \ x \text{ nat } \vdash of \ (\text{lam } \text{nat } \lambda y.x) \ (\text{arr } \text{nat } \text{nat})]\).
2008

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

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

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

- \( \lambda x. \lambda u. D \) has contextual LF type \([x:tm, u:of x nat \vdash of (\lambda y.x) (\lambda x.x)]\).

Key Observation: Abstract over LF contexts to enable recursion
“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 $D :: \Gamma \vdash M : A$ and $C :: \Gamma \vdash M : B$ then $E :: A = B$.
Step 2a: Theorem as Type

**Theorem: Type Uniqueness**

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

is represented as

**Computation Level Type for function unique**

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

- Parameterize over and distinguish between contexts
- Contexts are structured sequences
- Contexts are classified by context schemas
  
  ```scheme
ctx = some \{ t : \text{tp} \} \text{ block } x : \text{tm}, u : \text{of } x \ t;
  ```
Step 2a: Theorem as Type

Theorem: Type Uniqueness

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

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 distinguish 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.

$\rightsquigarrow M$ is implicitly associated with the identity substitution

$\rightsquigarrow A$ 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 $x : \text{nat}, y : \text{nat} \Rightarrow \text{nat}$ 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

```plaintext
rec unique: \( \Pi \gamma: \text{ctx.} \Pi A:[\text{tp}]. \Pi B:[\text{tp}]. \Pi M:[\gamma \vdash \text{tm}]. \\
[\gamma \vdash \text{of} M A[]) \rightarrow [\gamma \vdash \text{of} M B[]) \rightarrow [\vdash \text{eq} A B] =
```

Step 2b: Proofs as Programs

```haskell
rec unique: \( \Pi \gamma : \text{ctx}. \Pi A : [\text{tp}]. \Pi B : [\text{tp}]. \Pi M : [\gamma \vdash \text{tm}] . \)

\[ \gamma \vdash \text{of} M A \] \rightarrow \[ \gamma \vdash \text{of} M B \] \rightarrow [ \vdash \text{eq} A B ] =

\text{fn} \ d \ \Rightarrow \ \text{fn} \ c \ \Rightarrow \ \text{case} \ d \ \text{of}
```
rec unique : \( \Gamma : \text{ctx} \). \( A : [\text{tp}] \). \( B : [\text{tp}] \). \( M : [\Gamma \vdash \text{tm}] \). 

\[
[\gamma \vdash \text{of } M A[]] \rightarrow [\gamma \vdash \text{of } M B[]] \rightarrow [\vdash \text{eq } A B] = 
\]

fn d ⇒ fn c ⇒ case d of 
| [\gamma \vdash \text{t_app } D1 D2] ⇒ % Application Case
  let[\gamma \vdash \text{t_app } C1 C2] = c in 
  let[\vdash \text{ref}] = unique [\gamma \vdash D1] [\gamma \vdash C1] in 
  [\vdash \text{ref}]
Step 2b: Proofs as Programs

\[
\text{rec unique:}\Pi_{\gamma:ctx.} \Pi_{A:[tp].} \Pi_{B:[tp].} \Pi_{M:[\gamma \vdash tm].} \\
\quad [\gamma \vdash \text{of } M \ A[\ ]] \rightarrow [\gamma \vdash \text{of } M \ B[\ ]] \rightarrow [\vdash \text{eq } A \ B] = \\
\text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of} \\
\mid [\gamma \vdash \text{t_app } D1 \ D2] \Rightarrow \quad \% \text{Application Case} \\
\quad \text{let}[\gamma \vdash \text{t_app } C1 \ C2] = c \text{ in} \\
\quad \text{let}[\vdash \text{ref}] = \text{unique } [\gamma \vdash D1] [\gamma \vdash C1] \text{ in} \\
\quad [\vdash \text{ref}] \\
\mid [\gamma \vdash \text{t_lam } \lambda x.\lambda u. \ D] \Rightarrow \quad \% \text{Abstraction Case} \\
\quad \text{let}[\gamma \vdash \text{t_lam } \lambda x.\lambda u. C] = c \text{ in} \\
\quad \text{let}[\vdash \text{ref}] = \text{unique } [\gamma, b:\text{block } x:tm; u:\text{of } x \ _ \vdash D[b.x, b.u]] \\
\quad \quad [\gamma, b: _ \vdash C[b.x, b.u]] \text{ in} \\
\quad [\vdash \text{ref}] \
\]

Compact encoding of proofs about derivations as total functions.
Step 2b: Proofs as Programs

\[\text{rec unique:} \Pi \gamma: \text{ctx.}\Pi A:\text{[tp]}\Pi B:\text{[tp]}\Pi M:\underbrace{[\gamma \vdash \text{tm}]}_{\text{[\gamma \vdash \text{of M A}]}} \rightarrow [\gamma \vdash \text{of M B}] \rightarrow [\vdash \text{eq A B}] = \]

\text{fn d} \Rightarrow \text{fn c} \Rightarrow \text{case d of}
| [\gamma \vdash \text{t_app D1 D2}] \Rightarrow \% \text{Application Case}
\quad \text{let}[\gamma \vdash \text{t_app C1 C2}] = c \text{ in}
\quad \text{let}[\vdash \text{ref}] = \text{unique} [\gamma \vdash \text{D1}] [\gamma \vdash \text{C1}] \text{ in}
\quad [\vdash \text{ref}]

| [\gamma \vdash \text{t_lam } \lambda x.\lambda u. D] \Rightarrow \% \text{Abstraction Case}
\quad \text{let}[\gamma \vdash \text{t_lam } \lambda x.\lambda u. C] = c \text{ in}
\quad \text{let}[\vdash \text{ref}] = \text{unique} [\gamma, b: \text{block } x: \text{tm}; u: \text{of } x_\_ \vdash \text{D}[b.x, b.u]] [\gamma, b: _ \vdash \text{C}[b.x, b.u]] \text{ in}
\quad [\vdash \text{ref}]

| [\gamma \vdash \#q.u] \Rightarrow \% d : of \#q.x A \% \text{Assumption Case}
\quad \text{let}[\gamma \vdash \#r.u] = c \text{ in} \% c : of \#r.x B
\quad [\vdash \text{ref}]

Compact encoding of proofs about derivations as total functions.
Step 2b: Proofs as Programs

\textbf{rec unique} : \Pi \gamma : \text{ctx.} \Pi A : [\text{tp}]. \Pi B : [\text{tp}]. \Pi M : [\gamma \vdash \text{tm}].
\[
[\gamma \vdash \text{of M A[]} \rightarrow [\gamma \vdash \text{of M B[]} \rightarrow [\vdash \text{eq A B}]
\]
\text{fn d} \Rightarrow \text{fn c} \Rightarrow \text{case d of}
\begin{align*}
| [\gamma \vdash \text{t_app D1 D2}] & \Rightarrow \quad \% \text{Application Case} \\
\text{let}[\gamma \vdash \text{t_app C1 C2} = c \text{ in} \\
\text{let}[\vdash \text{ref}] = \text{unique} [\gamma \vdash \text{D1}] [\gamma \vdash \text{C1}] \text{ in} \\
& [\vdash \text{ref}] \\
\end{align*}
\begin{align*}
| [\gamma \vdash \text{t_lam} \lambda x. \lambda u. D] & \Rightarrow \quad \% \text{Abstraction Case} \\
\text{let}[\gamma \vdash \text{t_lam} \lambda x. \lambda u. C] = c \text{ in} \\
\text{let}[\vdash \text{ref}] = \text{unique} [\gamma, b:\text{block x:tm; u:of x } _ \vdash \text{D}[b.x, b.u]] \\
& [\gamma, b: _ \vdash \text{C}[b.x, b.u]] \text{ in} \\
& [\vdash \text{ref}] \\
| [\gamma \vdash \text{#q.u}] & \Rightarrow \quad \% d : \text{of #q.x A} \quad \% \text{Assumption Case} \\
\text{let}[\gamma \vdash \text{#r.u} = c \text{ in} \quad \% c : \text{of #r.x B} \\
& [\vdash \text{ref}] ;
\end{align*}

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
  
<table>
<thead>
<tr>
<th>On paper</th>
<th>In Beluga</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case Analysis</td>
<td>Case Analysis using pattern patching</td>
</tr>
<tr>
<td>Inversion</td>
<td>Case Analysis using pattern patching</td>
</tr>
<tr>
<td>IH</td>
<td>Recursive Call</td>
</tr>
</tbody>
</table>

- *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 cocreductive 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 –