A Case-Study in Programming Coinductive Proofs:
Mechanizing Proofs using Howe's Method

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

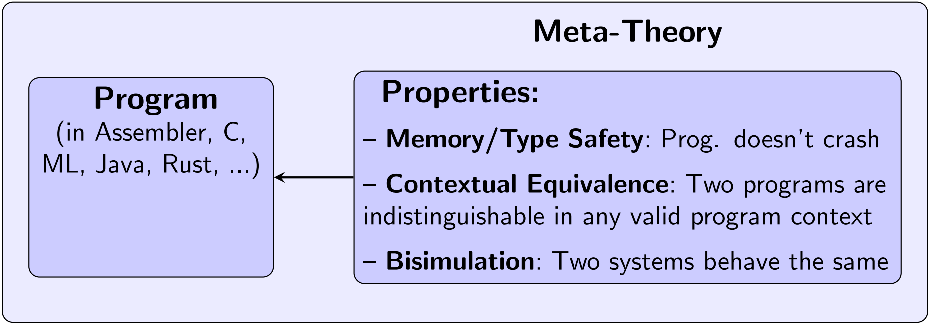
Currently at: Ludwig Maximilian University Munich, Germany

Joint work with D. Thibodeau (McGill) and A. Momigliano (Milan)

Mechanizing formal systems and proofs: How?

Mechanizing formal systems and proofs: How?

Formal systems (given via axioms and inference rules) play an important role when designing languages and more generally ensure that software are reliable, safe, and trustworthy.



See also: CompCert, DeepSpec, RustBelt, Sel4, Cogent, etc.

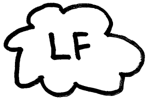
Challenges in Establishing Formal Guarantees

- Costly
- Large size of formal developments
  (CompCert: 4,400 lines of compiler code vs 28,000 lines of verification)
- Low-level representations
  For example: variables are modelled via de Bruijn indices, substitution, etc.
  - D. Hirschkoff [TPHOLs'97]: Bisimulation Proofs for the π-calculus in Coq (600 out of 800 lemmas are infrastructural)
  - Ambler and Crole [TPHOLs'99] Precongruence of bisimulation for PCFL ($\approx$ 160 infrastructural lemmas about de Brujn representation;main lemmas $\approx$ 34)
- Complex deep properties beyond type safety
- Scalability, reusability, maintainability, automation

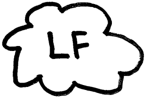
Main Question

Can we develop very high-level proof languages that make it easier to develop and maintain formal guarantees by providing the right primitives and abstractions to bring down the cost of verification?

Back in the eighties . . .

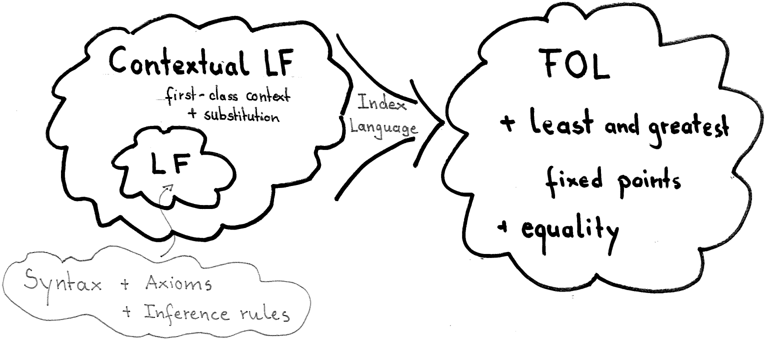


Back in the eighties . . .



How to reason (co)inductively?

Dawn of the 21. Century: (HOA)Syntax in Context



Indexed Functional Programs
defined by recursion and
(co) pattern matching

BELUGA: Programming (Co)inductive Proofs

- Functional programming with indexed (co)data types [POPL'08,POPL'12,POPL'13,ICFP'16]

On paper proof	In Beluga [IJCAR'10,CADE'15]
Case analysis of inputs	Case analysis via pattern matching
Inversion	Pattern matching using let-expression
Observations on output	Case analysis via copattern matching
(Co)Induction hypothesis	(Co)Recursive call

- Contextual LF

Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF
Well-scoped derivation	Contextual types and objects [TOCL'08]
Context	Context schemas
Properties of contexts (weakening, uniqueness)	Typing for schemas
Simultaneous Substitutions (composition, identity)	Substitution type [LFMTP'13]

This Talk: Mechanizing Meta-Theory

A Case Study of Proving Contextual Equivalence using Howe's Method

Contextual Equivalence = Bisimilarity

M and N are bisimilar iff M and N are contextual equivalent.

(a) **Open bisimilarity is a pre-congruence.**
$\Longrightarrow M$ and N are contextual equivalent.

(b) If M and N are contextual equivalent then M and N are bisimilar.

Contextual Equivalence = Bisimilarity

M and N are bisimilar iff M and N are contextual equivalent.

(a) **Open bisimilarity is a pre-congruence for PCFL (Mini-ML with lazy lists and recursion)[Pitts'97]**
$\Longrightarrow M$ and N are contextual equivalent.

(b) If M and N are contextual equivalent then M and N are bisimilar.

Step 1: Represent Types and Lambda-terms in LF

$$\begin{aligned}\text{Types } A, B ::= & \ \text{unit} \\ & \mid \text{list } A \\ & \mid A \Rightarrow B\end{aligned}$$

$$\begin{aligned}\text{Terms M,N} ::= & \ () \\ & \mid \text{nil} \mid M :: N \mid \text{case } M \text{ of } \{\text{nil} \Rightarrow N_1 \mid h :: t \Rightarrow N_2\} \\ & \mid x \mid \text{lam } x.M \mid \text{app } M\ N \mid \text{fix } x.M\end{aligned}$$

$$\text{Value } V ::= () \mid \text{lam } x.M \mid \text{nil} \mid M :: N$$

Step 1: Represent Types and Lambda-terms in LF

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

Terms $M, N ::= () \mid \text{nil} \mid M :: N \mid \text{case } M \text{ of } \{\text{nil} \Rightarrow N_1 \mid h :: t \Rightarrow N_2\} \mid x \mid \text{lam } x.M \mid \text{app } M\ N \mid \text{fix } x.M$

Value $V ::= () \mid \text{lam } x.M \mid \text{nil} \mid M :: N$

LF representation in Beluga (intrinsically typed terms)

```
LF tp:type =
| unit: tp
| arr : tp → tp  → tp
| list: tp → tp;
```

```
LF tm: tp → type =
| top  : tm unit
| lam  : (tm A → tm B) → tm (arr A B)
| app  : tm (arr A B) → tm A → tm B
| fix  : (tm A → tm A) → tm A
| nil  : tm (list A)
| cons : tm A → tm (list A) → tm (list A)
| lcase: tm (list A) → tm B →
        (tm A → tm (list A) → tm B) → tm B;
```

- Higher-order abstract syntax (HOAS) to represent variabe binding
- Inheriting α-renaming and single substitutions (β-reduction) from LF
- Warning: Negative occurrences!

Step 1: Representing Evaluations in LF

Evaluation Judgment: $\boxed{M \Downarrow V}$ read as "M evaluates V"

$$\frac{M \Downarrow \text{nil} \quad N_1 \Downarrow V}{\text{case } M \text{ of } \{\text{nil} \Rightarrow N_1 \mid h :: t \Rightarrow N_2\} \Downarrow V} \qquad \frac{M \Downarrow M_1 :: M_2 \quad [M_1/h, M_2/t]N_2 \Downarrow V}{\text{case } M \text{ of } \{\text{nil} \Rightarrow N_1 \mid h :: t \Rightarrow N_2\} \Downarrow V}$$

$$\frac{}{V \Downarrow V} \qquad \frac{M \Downarrow \text{lam } x.M' \quad [N/x]M' \Downarrow V}{\text{app } M \; N \Downarrow V} \qquad \frac{[\text{fix } x.M/x]M \Downarrow V}{\text{fix } x.M \Downarrow V}$$

Step 1: Representing Evaluations in LF

Evaluation Judgment: $\boxed{M \Downarrow V}$ read as "M evaluates V"

$$\frac{M \Downarrow \text{nil} \quad N_1 \Downarrow V}{\text{case } M \text{ of } \{\text{nil} \Rightarrow N_1 \mid h :: t \Rightarrow N_2\} \Downarrow V} \qquad \frac{M \Downarrow M_1 :: M_2 \quad [M_1/h, M_2/t]N_2 \Downarrow V}{\text{case } M \text{ of } \{\text{nil} \Rightarrow N_1 \mid h :: t \Rightarrow N_2\} \Downarrow V}$$

$$\frac{}{V \Downarrow V} \qquad \frac{M \Downarrow \text{lam } x.M' \quad [N/x]M' \Downarrow V}{\text{app } M \; N \Downarrow V} \qquad \frac{[\text{fix } x.M/x]M \Downarrow V}{\text{fix } x.M \Downarrow V}$$

LF representation in Beluga (intrinsically typed evaluations)

```
LF eval : tm A → tm A → type =
 | ev-app        : eval M (lam M’) → eval (M’ N) V → eval (app M N) V
 | ev-v          : value V → eval V V
 | ev-fix        : eval (M (fix M)) V → eval (fix M) V
 | ev-lcase-nil  : eval M nil → eval N1 V → eval (lcase M N1 N2) V.
 | ev-lcase-cons: eval M (cons H T) → eval (N2 H T) V
                 → eval (lcase M N1 N2) V.
```

Object-level substitution = LF application

Step 2: Similarity $M \preccurlyeq_A N$ (greatest fixed point)

$M \preccurlyeq_{\text{list } A} N$: $M \Downarrow \text{nil}$ entails $N \Downarrow \text{nil}$

$M \preccurlyeq_{\text{list } A} N$: $M \Downarrow H :: T$ entails that there is $N \Downarrow H' :: T'$ and $H \preccurlyeq_A H'$ and $T \preccurlyeq_{\text{list } A} T'$.

$M \preccurlyeq_{A \to B} N$: $M \Downarrow \text{lam } x.M'$ entails that there is $N \Downarrow \text{lam } y.N'$ and for every $R{:}A$, we have $M'[R/x] \preccurlyeq_B N'[R/y]$;

Step 2: Similarity $M \preccurlyeq_A N$ (greatest fixed point)

$M \preccurlyeq_{\text{list } A} N$: $M \Downarrow \text{nil}$ entails $N \Downarrow \text{nil}$

$M \preccurlyeq_{\text{list } A} N$: $M \Downarrow H :: T$ entails that there is $N \Downarrow H' :: T'$ and $H \preccurlyeq_A H'$ and $T \preccurlyeq_{\text{list } A} T'$.

$M \preccurlyeq_{A \to B} N$: $M \Downarrow \text{lam } x.M'$ entails that there is $N \Downarrow \text{lam } y.N'$ and for every $R{:}A$, we have $M'[R/x] \preccurlyeq_B N'[R/y]$;

Computation-level codata types in Beluga using records

```
coinductive   Sim: Π A:[tp].[tm A] → [tm A] → type =
{ (Sim_nil   : Sim [list A] [M] [N])::
                [eval M nil] → [eval N nil]
; (Sim_cons : Sim [list A] [M] [N])::
               [eval M (cons H L)] →  ExSimCons [H] [L] [N]
; (Sim_lam  : Sim [arr A B] [M] [N])::
               [eval M (lam λx.M')] → ExSimLam [x:tm A ⊢ M'] [N] }
 and inductive   ExSimLam: [x:tm A ⊢ tm B[]] → [tm (arr A B)] → type =
| ExSimlam: [eval N (lam λy.N')] → (Π R:[tm A]} Sim [B] [ M'[R] ] [ N'[R] ])
              → ExSimLam [x:tm A ⊢ M'] [N]
```

[POPL'13,ICFP'16]

A Simple Coinductive Proof: Similarity is reflexive

Proofs as Computation in Beluga

```
rec sim_refl : Π A:[tp].Π M:[tm A]Sim [A] [M] [M] =
fun  [unit]    [M] .Sim_unit                  d ⇒ d
  |  [list A]  [M] .Sim_nil                   d ⇒ d
  |  [list A]  [M] .Sim_cons [H] [T]          d ⇒
     ExSimcons  _ _ _ _ _ d ( sim_refl  [A] [H]) ( sim_refl [list A] [T])
  |  [arr A B] [M] .Sim_lam [x:tm A ⊢ M'] d ⇒
     ExSimlam  _ _ _ _ _ d (fun  [R] ⇒ sim_refl [B] [M'[R]] )
```

- Coinductive Proof = Recursive Program via copattern matching [POPL'13,ICFP'16]
- Implicit arguments that are reconstructed

Step 3: Defining Open Simulation as Inductive Data Type

Open Bisimulation: $\boxed{\Gamma \vdash M \preccurlyeq^{\circ}_{A} N \text{ iff } M[\sigma] \preccurlyeq_{A} N[\sigma], \text{ for any } \cdot \vdash \sigma : \Gamma.}$

- First-class contexts are classified by context schemas.

```
schema ctx = tm A.
```

- First-class substitutions σ have type $\Psi \vdash \Phi$ and provide a mapping from the context Φ to the context Ψ.

```
inductive  OSim:Π Γ:ctx.Π A:[tp]. [Γ ⊢tm A[]] → [Γ ⊢tm A[]] → type =
  | OSimC : (Π σ:[ ⊢Γ].Sim [A] [  ⊢ M[σ]] [  ⊢ N[σ]])
                          → OSim [A] [Γ ⊢ M   ] [Γ ⊢ N   ]
```

- Open similarity is closed under substitutions (exploits built-in composition of first-class substitutions).

```
rec osim_cus: Π Γ:ctx.Π Ψ:ctx.Π ρ:[Ψ ⊢Γ].OSim [A] [Γ ⊢M] [Γ ⊢N]
                → OSim [A] [Ψ ⊢M[ρ]] [Ψ ⊢N[ρ]] =
fun [Ψ ⊢ρ] (OSimC f) = OSimC (fun [σ] ⇒ f [ρ[σ]])
```

Howe-Relations, Substitutivity, etc.

- Indexed inductive types precisely characterize Howe-relations.

Howe-relation on open terms: $\Gamma \vdash M \preccurlyeq^{\mathcal{H}}_{A} N$

Howe-relation on substitutions: $\Gamma \vdash \sigma_1 \preccurlyeq^{\mathcal{H}}_{\Psi} \sigma_2$

Computation-level data types in Beluga

inductive Howe-Terms:ΠΓ:ctx.ΠA:[tp].ΠM:[Γ ⊢tm A[]].ΠN:[Γ ⊢tm A[]] **type**

inductive Howe-Subst:ΠΓ:ctx.ΠΨ:ctx.Πσ_1:[Γ ⊢Ψ].Πσ_2:[Γ ⊢Ψ] **type**

- Direct translation of the theorem as computation-level types
- Substitutivity for Howe-related terms is straightforward.
- Additional proofs (downward closed additional lemmas, etc.) are straightforward.
- No infrastructural lemmas needed

What did we learn from this case study?

- Higher-order abstract syntax (HOAS) encodings are convenient to model binding structures in syntax trees
- Contextual LF extends the spirit of HOAS to also support bindings with respect to a context of assumptions; this allows us to state and prove properties about open terms.
- First-class contexts and substitutions and their equational theory are a big win

  Substitution lemma, composition, decomposition, associativity, identity, etc.

$$\begin{array}{lcl} M[\cdot] & = & M \\ M[\sigma, N/x] & = & M[N/x][\sigma, x/x] \\ M[\sigma_1][\sigma_2] & = & M[[\sigma_1]\sigma_2] \end{array}$$

  a dozen such properties are needed

More Lessons

- Bisimilarity is a pre-congruence takes 35 theorems in Beluga
  No infrastructural theorems needed; all definitions and lemmas can be directly encoded included the notoriously difficult substitutivity
- Prototype of working with coinductive definitions (still needs work)
- Mechanization for STLC (not PCFL!) in Abella using HOAS style [Momigliano'12]:
  $\approx$ 45 theorems total
  $\approx$ 10 lemmas to maintain typing invariants;
  $\approx$ 6 lemmas to reason about the scope of variables;
  substitutivity was hard

Status Update on Beluga

- Prototype in OCaml (ongoing - last release March 2015)
  providing an interactive programming mode, totality checker [CADE'15]

  `https://github.com/Beluga-lang/Beluga`

- Mechanizing Types and Programming Languages - A companion:

  `https://github.com/Beluga-lang/Meta`

Status Update on Beluga

- Prototype in OCaml (ongoing - last release March 2015)
  providing an interactive programming mode, totality checker [CADE'15]

  `https://github.com/Beluga-lang/Beluga`

- Mechanizing Types and Programming Languages - A companion:

  `https://github.com/Beluga-lang/Meta`

Thank you!

"A language that doesn't affect the way you think about programming, is not worth knowing." - *Alan Perlis*