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?

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.

Programs (in Assembler, C, ML, Java, Rust, ...):

- **Memory/Type Safety**: Program doesn’t crash
- **Contextual Equivalence**: Two programs are indistinguishable in any valid program context
- **Bisimulation**: Two systems behave the same

Meta-Theory

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

Programs (in Assembler, C, ML, Java, Rust, ...) have the following properties:

- **Memory/Type Safety**: The program doesn't crash.
- **Contextual Equivalence**: Two programs are indistinguishable in any valid program context.
- **Bisimulation**: Two systems behave the same.

Meta-Theory

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 $\pi$-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 Brujin 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?
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]

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>In Beluga [IJCAR’10, CADE’15]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case analysis of inputs</td>
<td>Case analysis via pattern matching</td>
</tr>
<tr>
<td>Inversion</td>
<td>Pattern matching using let-expression</td>
</tr>
<tr>
<td>Observations on output</td>
<td>Case analysis via copattern matching</td>
</tr>
<tr>
<td>(Co)Induction hypothesis</td>
<td>(Co)Recursive call</td>
</tr>
</tbody>
</table>

- **Contextual LF**
  - Well-formed derivations
  - Renaming, Substitution
  - Well-scoped derivation
  - Context
  - Properties of contexts (weakening, uniqueness)
  - Simultaneous Substitutions (composition, identity)
  - Dependent types
  - $\alpha$-renaming, $\beta$-reduction in LF
  - Contextual types and objects [TOCL’08]
  - Context schemas
  - Typing for schemas
  - Substitution type [LFMTP’13]
This Talk: Mechanizing Meta-Theory

A Case Study of Proving Contextual Equivalence using Howe’s Method
Contextual Equivalence $\equiv$ Bisimilarity

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

(a) **Open bisimilarity is a pre-congruence.**

$$\Rightarrow M \text{ and } N \text{ are contextual equivalent.}$$

(b) If $M$ and $N$ are contextual equivalent then $M$ and $N$ are bisimilar.
Contextual Equivalence = Bisimilarity

\[ M \text{ and } N \text{ are bisimilar iff } M \text{ and } N \text{ are contextual equivalent.} \]

(a) **Open bisimilarity is a pre-congruence** for PCFL (Mini-ML with lazy lists and recursion)[Pitts’97]
\[ \Rightarrow M \text{ and } N \text{ are contextual equivalent.} \]

(b) If \( M \) and \( N \) are contextual equivalent then \( M \) and \( N \) are bisimilar.
Step 1: Represent Types and Lambda-terms in LF

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 $V ::= () \mid \text{lam } x. M \mid \text{nil} \mid M :: N$
Step 1: Represent Types and Lambda-terms in LF

Types $A, B ::= unit$

Terms $M, N ::= ()$

| list $A$
| $A \Rightarrow B$
| nil $M :: N$
| case $M$ of \{nil $\Rightarrow N_1$ | $h :: t \Rightarrow N_2$\}
| $x \mid \text{lam } x.M \mid \text{app } M N \mid \text{fix } x.M$

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

LF representation in Beluga (intrinsically typed terms)

\[
\text{LF tp: type} = \\
| \text{unit: tp} \\
| \text{arr : tp} \rightarrow \text{tp} \rightarrow \text{tp} \\
| \text{list: tp} \rightarrow \text{tp};
\]

\[
\text{LF tm: tp} \rightarrow \text{type} = \\
| \text{top : tm unit} \\
| \text{lam : (tm A} \rightarrow \text{tm B}) \rightarrow \text{tm (arr A B)} \\
| \text{app : tm (arr A B)} \rightarrow \text{tm A} \rightarrow \text{tm B} \\
| \text{fix : (tm A} \rightarrow \text{tm A}) \rightarrow \text{tm A} \\
| \text{nil : tm (list A)} \\
| \text{cons : tm A} \rightarrow \text{tm (list A)} \rightarrow \text{tm (list A)} \\
| \text{lcase: tm (list A} \rightarrow \text{tm B} \rightarrow \text{(tm A} \rightarrow \text{tm (list A)} \rightarrow \text{tm B}) \rightarrow \text{tm B};
\]

- Higher-order abstract syntax (HOAS) to represent variable binding
- Inheriting $\alpha$-renaming and single substitutions ($\beta$-reduction) from LF
- Warning: Negative occurrences!
Step 1: Representing Evaluations in LF

Evaluation Judgment: \( M \Downarrow V \) read as “\( M \) evaluates \( V \)”

\[
\begin{align*}
M \Downarrow \text{nil} & \quad N_1 \Downarrow V \\
\text{case } M \text{ of } \{ \text{nil } \Rightarrow N_1 \mid h :: t \Rightarrow N_2 \} \Downarrow V \\
\hline
V \Downarrow V \\
M \Downarrow \text{lam } x. M' & \quad [N/x]M' \Downarrow V & \quad [\text{fix } x. M/x]M \Downarrow V \\
\text{app } M N \Downarrow V & \\
\text{fix } x. M \Downarrow V
\end{align*}
\]
Step 1: Representing Evaluations in LF

Evaluation Judgment: \[ M \downarrow V \]
read as “\( M \) evaluates \( V \)”

\[
\begin{align*}
M & \downarrow \text{nil} & N_1 & \downarrow V \\
\text{case } M & \text{ of } \{ \text{nil } \Rightarrow N_1 \mid h :: t \Rightarrow N_2 \} & \downarrow V \\
V & \downarrow V
\end{align*}
\]

\[
\begin{align*}
M & \downarrow M_1 :: M_2 & [M_1/h, M_2/t]N_2 & \downarrow V \\
\text{case } M & \text{ of } \{ \text{nil } \Rightarrow N_1 \mid h :: t \Rightarrow N_2 \} & \downarrow V \\
N_1 & \downarrow V \\
N_2 & \downarrow V
\end{align*}
\]

\[
\begin{align*}
M & \downarrow \text{lam } x.M' & [N/x]M' & \downarrow V \\
\text{app } M & N & \downarrow V \\
\text{fix } x.M & \downarrow V
\end{align*}
\]

LF representation in Beluga (intrinsically typed evaluations)

\[
\text{LF eval : tm A } \rightarrow \text{ tm A } \rightarrow \text{ type } =
\]

| ev-app | eval M (lam M’) \rightarrow eval (M’ N) V \rightarrow eval (app M N) V |
| ev-v | value V \rightarrow eval V V |
| ev-fix | eval (M (fix M)) V \rightarrow eval (fix M) V |
| ev-lcase-nil | eval M nil \rightarrow eval N1 V \rightarrow eval (lcase M N1 N2) V. |
| ev-lcase-cons | eval M (cons H T) \rightarrow eval (N2 H T) V \rightarrow eval (lcase M N1 N2) V. |

Object-level substitution = LF application
Step 2: Similarity $M \preceq_A N$ (greatest fixed point)

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

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

$M \preceq_{A \rightarrow B} N : M \Downarrow \text{lam } x. M'$ entails that there is $N \Downarrow \text{lam } y. N'$ and for every $R : A$, we have $M'[R/x] \preceq_B N'[R/y]$;
Step 2: Similarity $M \lesssim_A N$ (greatest fixed point)

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

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

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

Computation-level codata types in Beluga using records

```plaintext
coinductive Sim: \(\Pi A: [\text{tp}]. [\text{tm } A] \rightarrow [\text{tm } A] \rightarrow \text{type} = \) 
{ (Sim_nil : Sim [list A] [M] [N]):: 
  [eval M \text{nil}] \rightarrow [eval N \text{nil}] 
; (Sim_cons : Sim [list A] [M] [N]):: 
  [eval M (\text{cons } H \text{L})] \rightarrow \text{ExSimCons [H] [L] [N]} 
; (Sim_lam : Sim [arr A B] [M] [N]):: 
  [eval M (\text{lam } \lambda x. M')] \rightarrow \text{ExSimLam [x:tm } A \vdash M' [N] \}

and inductive ExSimLam: [x:tm } A \vdash \text{tm } B[]] \rightarrow [\text{tm } (\text{arr } A \text{B})] \rightarrow \text{type} = 
| ExSimlam: [eval N (\text{lam } \lambda y. N')] \rightarrow (\Pi R: [\text{tm } A}] \text{Sim [B] [ M' [R] ] [ N' [R] ]}) 
\rightarrow \text{ExSimLam [x:tm } A \vdash M' [N]}
```

[POPL’13,ICFP’16]
A Simple Coinductive Proof: Similarity is reflexive

Proofs as Computation in Beluga

```
rec sim_refl : \ A:[tp].\ 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: \[ \Gamma \vdash M \sim^\circ_A N \iff M[\sigma] \sim_A N[\sigma], \text{ for any } \cdot \vdash \sigma : \Gamma. \]

- First-class contexts are classified by context schemas.
  
  \textbf{schema} \ ctx = \ tm \ A.

- First-class substitutions \( \sigma \) have type \( \Psi \vdash \Phi \) and provide a mapping from the context \( \Phi \) to the context \( \Psi \).

\[
\text{inductive} \quad \text{OSim} : \Pi \; \Gamma : \text{ctx}. \; \Pi \; A : [\text{tp}] . \; [\Gamma \vdash \text{tm} \ A []] \rightarrow [\Gamma \vdash \text{tm} \ A []] \rightarrow \text{type} =
\]

\[
\text{OSimC} : (\Pi \; \sigma : [ \vdash \Gamma]. \text{Sim} \ [A] \ [ \vdash M[\sigma]] [ \vdash N[\sigma]])
\rightarrow \text{OSim} \ [A] \ [\Gamma \vdash M \ ] [\Gamma \vdash N \ ]
\]

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

\[
\text{rec osim_cus: } \Pi \; \Gamma : \text{ctx}. \Pi \; \Psi : \text{ctx}. \Pi \; \rho : [\Psi \vdash \Gamma]. \text{OSim} \ [A] \ [\Gamma \vdash M] [\Gamma \vdash N]
\rightarrow \text{OSim} \ [A] \ [\Psi \vdash M[\rho]] [\Psi \vdash N[\rho]] =
\text{fun} \ [\Psi \vdash \rho] \ (\text{OSimC} \ f) = \text{OSimC} \ (\text{fun} \ [\sigma] \Rightarrow f \ [\rho[\sigma]])
\]
Indexed inductive types precisely characterize Howe-relations.

Howe-relation on open terms: $\Gamma \vdash M \leq^H_A N$

Howe-relation on substitutions: $\Gamma \vdash \sigma_1 \leq^H_\Psi \sigma_2$

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.

\[
M[\cdot] = M
\]
\[
M[\sigma, N/x] = M[N/x][\sigma, x/x]
\]
\[
M[\sigma_1][\sigma_2] = M[[\sigma_1]\sigma_2]
\]

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]:
  \[ \approx 45 \text{ theorems total} \]
  \[ \approx 10 \text{ lemmas to maintain typing invariants;} \]
  \[ \approx 6 \text{ lemmas to reason about the scope of variables;} \]

  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