Mechanizing Meta-Theory in Beluga

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

Mechanizing formal systems and proofs: How and Why?

Mechanizing formal systems and proofs: How and Why?

- 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.
- Proofs (that a given property is satisfied) are becoming pervasive and an integral part of certified software. (see: CompCert, DeepSpec, RustBelt, Sel4, Cogent)

Mechanizing formal systems and proofs: How and Why?

- 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.

- Proofs (that a given property is satisfied) are becoming pervasive and an integral part of certified software. (see: CompCert, DeepSpec, RustBelt, Sel4, Cogent)

Meta-Theory

Program
(in Assembler, C, ML, Java, Rust, ...)

Properties:

– **Memory/Type Safety**: Prog. doesn't crash

– **Contextual Equivalence**: Two programs are indistinguishable in any valid program context

– **Bisimulation**: Two systems behave the same

Challenges in Establishing Formal Guarantees

Challenges in Establishing Formal Guarantees

- Costly

Challenges in Establishing Formal Guarantees

- Costly
- Large size of formal developments
  - CompCert: 4,400 lines of compiler code vs 28,000 lines of verification
  - A specification of dependent Haskel [ICFP'17]: 17K Coq code + 13K generated code from Ott Spec.; 1.4K Ott Specification;

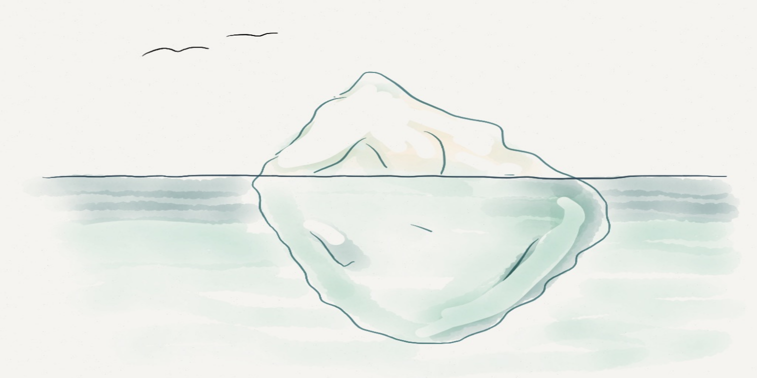
Challenges in Establishing Formal Guarantees

- Costly
- Large size of formal developments
  - CompCert: 4,400 lines of compiler code vs 28,000 lines of verification
  - A specification of dependent Haskel [ICFP'17]: 17K Coq code + 13K generated code from Ott Spec.; 1.4K Ott Specification;
- Low-level representations (variables are modelled via de Bruijn indices)
  - 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)
  - J. Kaiser et. al [FSCD'17]: Relating System F and $\lambda 2$ (PTS) de Bruijn over 1K lines of infrastructural code in Coq; over 500 lines in Abella; about 100 in Beluga

Challenges in Establishing Formal Guarantees

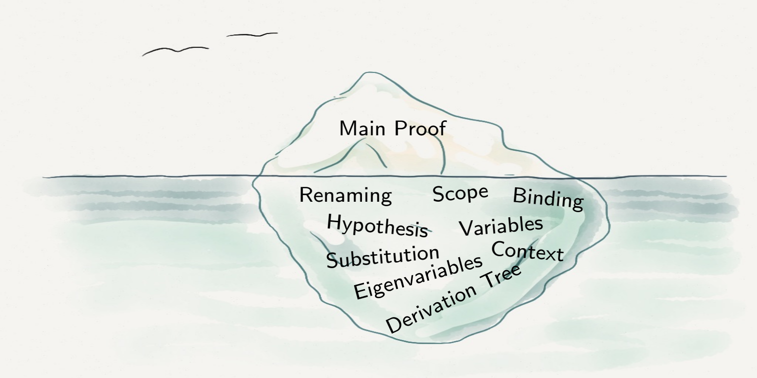
- Costly
- Large size of formal developments
  - CompCert: 4,400 lines of compiler code vs 28,000 lines of verification
  - A specification of dependent Haskel [ICFP'17]: 17K Coq code + 13K generated code from Ott Spec.; 1.4K Ott Specification;
- Low-level representations (variables are modelled via de Bruijn indices)
  - 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)
  - J. Kaiser et. al [FSCD'17]: Relating System F and $\lambda 2$ (PTS) de Bruijn over 1K lines of infrastructural code in Coq; over 500 lines in Abella; about 100 in Beluga
- Scalability, reusability, maintainability, automation

Proofs: The tip of the iceberg



"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]*

Proofs: The tip of the iceberg



"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]*

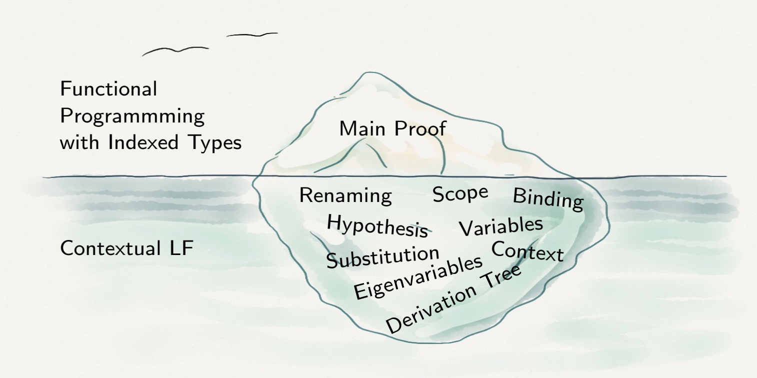
Question

What are good meta-languages to program and reason with formal systems and proofs?

"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]*

Above and Below the Surface

BELUGA: Dependently typed Programming and Proof Environment



- Below the surface: Support for key concepts based on Contextual LF
- Above the surface: (Co)Inductive Proofs = (Co)Recursive Programs using (Co)pattern Matching with built-in index language of Contextual LF objects

Design of Beluga

- Top : 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

- Bottom: 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'13,15]

This Talk

Design and implementation of Beluga

- Introduction
- Example: Proof by logical relation
- Writing a proof in Beluga ...
- Conclusion and curent work

"The limits of my language mean the limits of my world."
- L. Wittgenstein

This Talk

Design and implementation of Beluga

- Introduction
- Example: Proof by logical relations
- Writing a proof in Beluga ...
- Conclusion and curent work

"The limits of my language mean the limits of my world."
- L. Wittgenstein

Simply Typed Lambda-calculus (Gentzen-style)

Types $A, B ::= \mathsf{i} \mid A \Rightarrow B$

Terms M, N $::= x \mid \mathbf{c} \mid \mathsf{lam}\ x.M \mid \mathsf{app}\ M\ N$

Evaluation Judgment: $\boxed{M \longrightarrow M'}$ read as "M steps to M'"

$$\frac{}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ N \longrightarrow [N/x]M}\ \mathsf{s/beta} \qquad \frac{}{M \longrightarrow M}\ \mathsf{s/refl}$$

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \mathsf{s/app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \mathsf{s/trans}$$

Simply Typed Lambda-calculus (Gentzen-style)

Types $A, B ::= \mathrm{i} \mid A \Rightarrow B$

Terms M, N $::= x \mid \mathbf{c} \mid \mathrm{lam}\ x.M \mid \mathrm{app}\ M\ N$

Evaluation Judgment: $\boxed{M \longrightarrow M'}$ read as "M steps to M'"

$$\frac{}{\mathrm{app}\ (\mathrm{lam}\ x.M)\ N \longrightarrow [N/x]M}\ \mathrm{s/beta} \qquad \frac{}{M \longrightarrow M}\ \mathrm{s/refl}$$

$$\frac{M \longrightarrow M'}{\mathrm{app}\ M\ N \longrightarrow \mathrm{app}\ M'\ N}\ \mathrm{s/app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \mathrm{s/trans}$$

Typing Judgment: $\boxed{M : A}$ read as "M has type A" (Gentzen-style)

$$\frac{}{\mathbf{c} : \mathbf{i}}\ \mathrm{const} \qquad \frac{\begin{array}{c}\overline{x : A}\ u \\ \vdots \\ M : B\end{array}}{\mathrm{lam}\, x.M : A \Rightarrow B}\ \mathrm{lam}^{x,u} \qquad \frac{M : A \Rightarrow B \quad N : A}{\mathrm{app}\ M\ N : B}\ \mathrm{app}$$

Simply Typed Lambda-calculus with Contexts

Types and Terms

Types $A, B ::= \mathsf{i} \mid A \Rightarrow B$

Terms M, N $::= x \mid \mathbf{c} \mid \mathsf{lam}\ x.M \mid \mathsf{app}\ M\ N$

Evaluation Judgment: $\boxed{M \longrightarrow M'}$ read as "M steps to M'"

$$\frac{}{\mathsf{app}\ (\mathsf{lam} x.M)\ N \longrightarrow [N/x]M}\ \mathsf{s/beta} \qquad \frac{}{M \longrightarrow M}\ \mathsf{s/refl}$$

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \mathsf{s/app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \mathsf{s/trans}$$

Typing Judgment: $\boxed{\Gamma \vdash M : A}$ read as "M has type A in context Γ"

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \mathsf{lam}\, x.M : A \Rightarrow B}\ \mathsf{lam}^x \qquad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N\ A}{\Gamma \vdash \mathsf{app}\ M\ N : B}\ \mathsf{app}$$

Context $\Gamma ::= \cdot \mid \Gamma, x : A$ We are introducing the variable x together with the assumption $x : A$

Weak Normalization for Simply Typed Lambda-calculus

Weak Normalization for Simply Typed Lambda-calculus

Theorem

If $\vdash M : A$ then M halts, i.e. there exists a value V s.t. $M \longrightarrow^* V$.

Weak Normalization for Simply Typed Lambda-calculus

Theorem

If $\vdash M : A$ then M halts, i.e. there exists a value V s.t. $M \longrightarrow^* V$.

Proof.

1 Define reducibility candidate $\mathcal{R}_A$

$$\begin{array}{lcl} \mathcal{R}_{\mathbf{i}} & = & \{M \mid M \text{ halts}\} \\ \mathcal{R}_{A \Rightarrow B} & = & \{M \mid M \text{ halts and } \forall N \in \mathcal{R}_A, (\text{app } M\ N) \in \mathcal{R}_B\} \end{array}$$

2 If $M \in \mathcal{R}_A$ then M halts.

3 Backwards closed: If $M' \in \mathcal{R}_A$ and $M \longrightarrow M'$ then $M \in \mathcal{R}_A$.

4 Fundamental Lemma: If $\vdash M : A$ then $M \in \mathcal{R}_A$. (Requires a generalization)

□

Generalization of Fundamental Lemma

Lemma (Main lemma)

If $\mathcal{D} : \Gamma \vdash M : A$ *and* $\sigma \in \mathcal{R}_\Gamma$ *then* $[\sigma]M \in \mathcal{R}_A$.

where $\sigma \in \mathcal{R}_\Gamma$ is defined as:

$$\frac{}{\cdot \in \mathcal{R}_{.}} \qquad\qquad \frac{\sigma \in \mathcal{R}_\Gamma \quad N \in \mathcal{R}_A}{(\sigma, N/x) \in \mathcal{R}_{\Gamma, x:A}}$$

Generalization of Fundamental Lemma

Lemma (Main lemma)

If $\mathcal{D} : \Gamma \vdash M : A$ *and* $\sigma \in \mathcal{R}_\Gamma$ *then* $[\sigma]M \in \mathcal{R}_A$.

Proof.

Case $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash \mathrm{lam}\ x.M : A \Rightarrow B}\ \mathit{lam}$

$[\sigma](\mathrm{lam}\ x.M) = \mathrm{lam}\ x.([\sigma, x/x]M)$ by properties of substitution

`halts` $(\mathrm{lam}\ x.[\sigma, x/x]M)$ since it is a value

Suppose $N \in \mathcal{R}_A$.

$[\sigma, N/x]M \in \mathcal{R}_B$ by I.H. on $\mathcal{D}_1$ since $\sigma \in \mathcal{R}_\Gamma$

$[N/x][\sigma, x/x]M \in \mathcal{R}_B$ by properties of substitution

$\mathrm{app}\ (\mathrm{lam}\ x.\ [\sigma, x/x]M)\ N \in \mathcal{R}_B$ by Backwards closure

Hence $[\sigma](\mathrm{lam}\ x.M) \in \mathcal{R}_{A \Rightarrow B}$ by definition

Challenging Benchmark

"I discovered that the core part of the proof (here proving lemmas about CR) is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening." *T. Altenkirch [TLCA'93]*

Challenging Benchmark

"I discovered that the core part of the proof (here proving lemmas about CR) is fairly straightforward and only requires a good understanding of the paper version. However, in completing the proof I observed that in certain places I had to invest much more work than expected, e.g. proving lemmas about substitution and weakening." *T. Altenkirch [TLCA'93]*

This Talk

Design and implementation of Beluga

- Introduction
- Example: Proof by logical relations
- Writing a proof in Beluga . . .
- Conclusion and curent work

Step 1: Represent Types and Lambda-terms in LF

Types $A, B ::= \mathsf{i} \mid A \Rightarrow B$

Terms M, N $::= x \mid \mathbf{c} \mid \mathsf{lam}\ x.M \mid \mathsf{app}\ M\ N$

Typing rules

$$\frac{}{\mathbf{c} : \mathbf{i}}\ \text{const} \qquad \frac{\begin{array}{c}\overline{x : A}\ u \\ \vdots \\ M : B\end{array}}{\mathsf{lam}\, x.M : A \Rightarrow B}\ \mathsf{lam}^x \qquad \frac{M : A \Rightarrow B \quad N : A}{\mathsf{app}\ M\ N : B}\ \mathsf{app}$$

LF representation in Beluga

```
LF tp:type =
| i: tp
| arr: tp → tp  → tp;
```

```
LF tm: tp → type =
| c   : tm i
| lam:(tm A → tm B)  → tm (arr A B)
| app: tm (arr A B) → tm A → tm B;
```

- Higher-order abstract syntax (HOAS) to represent variabe binding (lam x.app (lam $y.y$) x) is represented as `(lam λx. app (lam λy.y) x)`.
- Inheriting α-renaming and single substitutions (β-reduction) from LF

Step 1: Encoding Evaluation in LF

Evaluation Judgment: $\boxed{M \longrightarrow M'}$ read as "M steps to M'"

$$\frac{}{\text{app (lam } x.M)\ N \longrightarrow [N/x]M}\ \text{s/beta} \qquad \frac{}{M \longrightarrow M}\ \text{s/refl}$$

$$\frac{M \longrightarrow M'}{\text{app } M\ N \longrightarrow \text{app } M'\ N}\ \text{s/app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \text{s/trans}$$

Step 1: Encoding Evaluation in LF

Evaluation Judgment: $\boxed{M \longrightarrow M'}$ read as "M steps to M'"

$$\frac{}{\text{app (lam } x.M)\ N \longrightarrow [N/x]M}\ \text{s/beta} \qquad \frac{}{M \longrightarrow M}\ \text{s/refl}$$

$$\frac{M \longrightarrow M'}{\text{app } M\ N \longrightarrow \text{app } M'\ N}\ \text{s/app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \text{s/trans}$$

LF representation in Beluga

```
LF step: tm A → tm A → type =
| s/beta : step (app (lam M) N) (M N)
| s/app  : step M M' → step (app M N) (app M' N)
| s/refl : step M M
| s/trans: step M M' → step M' N → step M N;
```

- Substitution in the `tm` language is modelled via LF application and LF β-reduction

So far . . .

. . . encodings in the logical framework LF

So far . . .

. . . encodings in the logical framework LF

Question: How to reason about LF terms and types?

So far . . .

. . . encodings in the logical framework LF

Question: How to reason about LF terms and types?

Answer: Contextual terms and types [TOCL'08]

What are contextual terms and types?

Recall: `lam λx. app (lam λy.y) x`

What are contextual terms and types?

Recall: `lam` λ`x.` $\boxed{\texttt{app (lam } \lambda\texttt{y.y) x}}$

What are contextual terms and types?

Recall: `lam` λ`x.` $\boxed{\texttt{app (lam } \lambda\texttt{y.y) x}}$

- Subexpression `app (lam` λ`y.y) x` refers to the variable `x`!

What are contextual terms and types?

Recall: `lam` λ`x.` $\boxed{\texttt{app (lam } \lambda\texttt{y.y) x}}$

- Subexpression `app (lam` λ`y.y) x` refers to the variable `x`!
- **The contextual view**:
  Pair up terms and types with their context of variables!

  `[x:tm _` $\vdash$ `app (lam` λ`y.y) x]` has type `[x:tm _` $\vdash$ `tm _ ]`

- Contextual terms and types are closed objects!
  $\Longrightarrow$ there are canonical forms
  $\Longrightarrow$ check for equality by comparing their canonical forms

What are contextual terms and types?

Recall: `lam` λ`x.` $\boxed{\texttt{app (lam } \lambda\texttt{y.y) x}}$

- Subexpression `app (lam` λ`y.y) x` refers to the variable `x`!
- **The contextual view**:
  Pair up terms and types with their context of variables!

  `[x:tm _` $\vdash$ `app (lam` λ`y.y) x]` has type `[x:tm _` $\vdash$ `tm _ ]`

- Contextual terms and types are closed objects!
  $\Longrightarrow$ there are canonical forms
  $\Longrightarrow$ check for equality by comparing their canonical forms
- Reason about contextual terms and types using first-order logic with least and greatest fixed points.
  $\Longrightarrow$ need to abstract over contexts

Reducibility Candidates as Indexed Types

Reducibility candidates for terms $M \in \mathcal{R}_A$:

$$\begin{array}{lcl} \mathcal{R}_{\mathbf{i}} & = & \{M \mid \texttt{halts}\ M\} \\ \mathcal{R}_{A \Rightarrow B} & = & \{M \mid \texttt{halts}\ M \text{ and } \forall N \in \mathcal{R}_A, (\mathsf{app}\ M\ N) \in \mathcal{R}_B\} \end{array}$$

Reducibility Candidates as Indexed Types

Reducibility candidates for terms $M \in \mathcal{R}_A$:

$$\begin{array}{lcl} \mathcal{R}_{\mathbf{i}} & = & \{M \mid \texttt{halts}\ M\} \\ \mathcal{R}_{A \Rightarrow B} & = & \{M \mid \texttt{halts}\ M \text{ and } \forall N \in \mathcal{R}_A, (\text{app } M\ N) \in \mathcal{R}_B\} \end{array}$$

Computation-level data types in Beluga

```
stratified   Reduce : {A:[tp]} {M:[tm A]} type =
| I   : [halts M] → Reduce [i] [M]
| Arr : [halts M] →
        ({N:[tm A]} Reduce [A] [N] → Reduce [B] [app M N])
     → Reduce [arr A B] [M];
```

- `[app M N]` and `[arr A B]` is shorthand for `[ ⊢app M N]` and `[ ⊢arr A B]`; they are contextual types [TOCL'08].
- Note: → is overloaded.
  - → is the LF function space : binders in the object language are modelled by LF functions (used inside `[ ]`)
  - → is a computation-level function (used outside `[ ]`)
- Not strictly positive definition, but stratified.

Reducibility Candidates as Indexed Types

Reducibility candidates for substitutions $\sigma \in \mathcal{R}_\Gamma$:

$$\frac{}{\cdot \in \mathcal{R}_.} \qquad \frac{\sigma \in \mathcal{R}_\Gamma \quad N \in \mathcal{R}_A}{(\sigma, N/x) \in \mathcal{R}_{\Gamma,x:A}}$$

Reducibility Candidates as Indexed Types

Reducibility candidates for substitutions $\sigma \in \mathcal{R}_\Gamma$:

$$\frac{}{\cdot \in \mathcal{R}_\cdot} \qquad \frac{\sigma \in \mathcal{R}_\Gamma \quad N \in \mathcal{R}_A}{(\sigma, N/x) \in \mathcal{R}_{\Gamma, x:A}}$$

Computation-level data types in Beluga

```
inductive  RedSub : (Γ:ctx){σ: ⊢ Γ} type =
| Nil  : RedSub [ ⊢ ^ ]
| Cons : RedSub [ ⊢ σ] → Reduce [ ⊢ A] [ ⊢ M] → RedSub [ ⊢ σ, M ];
```

- Contexts are structured sequences and are classified by context schemas

  ```
  schema ctx =  x:tm A.
  ```

- Substitution σ are first-class and have type $\Psi \vdash \Phi$ providing a mapping from Φ to Ψ.

Theorems as Computation-level Types

Lemma (Backward closed)

If $M \longrightarrow M'$ *and* $M' \in \mathcal{R}_A$ *then* $M \in \mathcal{R}_A$.

```
rec closed : [step M M’] → Reduce [A] [M’] → Reduce [A] [M] = ? ;
```

Lemma (Main lemma)

If $\Gamma \vdash M : A$ *and* $\sigma \in \mathcal{R}_\Gamma$ *then* $[\sigma]M \in \mathcal{R}_A$.

```
rec main : {Γ:ctx}{M:[Γ ⊢ tm A[]]} RedSub [ ⊢ σ] → Reduce [A] [M[σ]] = ? ;
```

Fundamental Lemma

Fundamental Lemma

rec closed : [step M M'] $\rightarrow$ Reduce [A] [M'] $\rightarrow$ Reduce [A] [M] = ? ;

rec main : {Γ:ctx}{M:[$\Gamma \vdash$ tm A[]]} RedSub [$\vdash \sigma$] $\rightarrow$ Reduce [A] [M[σ]] =

Fundamental Lemma

```
rec closed : [step M M'] → Reduce [A] [M'] → Reduce [A] [M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢ tm A[]]} RedSub [ ⊢ σ] → Reduce [A] [M[σ]] =
mlam Γ ⇒ mlam M ⇒ fn rs ⇒ case [Γ ⊢ M ] of
| [Γ ⊢ #p ] ⇒ lookup [Γ] [Γ ⊢ #p ] rs                         % Variable
```

Fundamental Lemma

```
rec closed : [step M M'] → Reduce [A] [M'] → Reduce [A] [M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢ tm A[]]} RedSub [ ⊢ σ] → Reduce [A] [M[σ]] =
mlam Γ ⇒ mlam M ⇒ fn rs ⇒ case [Γ ⊢ M ] of
| [Γ ⊢ #p ] ⇒ lookup [Γ] [Γ ⊢ #p ] rs                    % Variable
| [Γ ⊢ app M1 M2] ⇒                                      % Application
  let Arr ha f = main [Γ] [Γ ⊢ M1] rs in
  f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2] rs)
```

Fundamental Lemma

```
rec closed : [step M M'] → Reduce [A] [M'] → Reduce [A] [M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢ tm A[]]} RedSub [ ⊢ σ] → Reduce [A] [M[σ]] =
mlam Γ⇒mlam M ⇒fn rs ⇒ case [Γ ⊢M ] of
| [Γ ⊢#p ] ⇒lookup [Γ] [Γ ⊢#p ] rs                          % Variable

| [Γ ⊢ app M1 M2] ⇒                                        % Application
  let Arr ha f = main [Γ] [Γ ⊢ M1] rs in
  f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2] rs)

| [Γ ⊢ lam λx. M1] ⇒                                       % Abstraction
  Arr [ ⊢ h/value s/refl v/lam]
   (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
                               (main [Γ,x:tm _] [Γ,x ⊢ M1] (Cons rs rN)))
```

Fundamental Lemma

```
rec closed : [step M M'] → Reduce [A] [M'] → Reduce [A] [M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢ tm A[]]} RedSub [ ⊢ σ] → Reduce [A] [M[σ]] =
mlam Γ⇒mlam M ⇒fn rs ⇒ case [Γ ⊢M ] of
| [Γ ⊢#p ] ⇒lookup [Γ] [Γ ⊢#p ] rs                          % Variable

| [Γ ⊢ app M1 M2] ⇒                                          % Application
  let Arr ha f = main [Γ] [Γ ⊢ M1] rs in
  f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2] rs)

| [Γ ⊢ lam λx. M1] ⇒                                         % Abstraction
  Arr [ ⊢ h/value s/refl v/lam]
   (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
                         (main [Γ,x:tm _] [Γ,x ⊢ M1] (Cons rs rN)))

| [Γ ⊢ c] ⇒ I [ ⊢ h/value s/refl v/c];                        % Constant
```

Fundamental Lemma

```
rec closed : [step M M'] → Reduce [A] [M'] → Reduce [A] [M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢ tm A[]]} RedSub [ ⊢ σ] → Reduce [A] [M[σ]] =
mlam Γ ⇒ mlam M ⇒ fn rs ⇒ case [Γ ⊢ M ] of
| [Γ ⊢#p ] ⇒ lookup [Γ] [Γ ⊢#p ] rs                      % Variable

| [Γ ⊢ app M1 M2] ⇒                                       % Application
  let Arr ha f = main [Γ] [Γ ⊢ M1] rs in
  f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2] rs)

| [Γ ⊢ lam λx. M1] ⇒                                      % Abstraction
  Arr [ ⊢ h/value s/refl v/lam]
   (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
                          (main [Γ,x:tm _] [Γ,x ⊢ M1] (Cons rs rN)))

| [Γ ⊢ c] ⇒ I [ ⊢ h/value s/refl v/c];                     % Constant
```

- Direct encoding of on-paper proof
- Equations about substitution properties automatically discharged (amounts to roughly a dozen lemmas about substitution and weakening)
- Total encoding about 75 lines of Beluga code

Some Alternatives

General Theorem Proving Environments

- Calculus of Construction (Coq) / Martin Löf Type Theory (Agda)
  No special support for variables, assumptions, derivation trees, etc.
  About a dozen extra lemmas
- Isabelle / Nominal
  support for variable names, but not for assumptions, derivation trees, etc.
  based on nominal set theory; about a dozen extra lemmas

Some Alternatives

General Theorem Proving Environments

- Calculus of Construction (Coq) / Martin Löf Type Theory (Agda)
  No special support for variables, assumptions, derivation trees, etc.
  About a dozen extra lemmas
- Isabelle / Nominal
  support for variable names, but not for assumptions, derivation trees, etc.
  based on nominal set theory; about a dozen extra lemmas

Domain-specific Provers (Higher-Order Abstract Syntax (HOAS))

- Abella: encode second-order hereditary Harrop (HH) logic in $\mathcal{G}$, an extension of first-order logic with a new quantifier ∇, and develop inductive proofs in $\mathcal{G}$ by reasoning about the size of HH derivations .
  diverges a bit from on-paper proof; 4 additional lemmas
- Twelf: Too weak for directly encoding such proofs; implement auxiliary logic.

Logical Relations on Open Terms

- Allowing reductions under lambda-abstractions:

$$\frac{M \longrightarrow N}{\mathsf{lam}\, x.M \longrightarrow \mathsf{lam}\, x.N}\ \mathsf{s/lam}^x$$

Logical Relations on Open Terms

- Allowing reductions under lambda-abstractions:

$$\frac{M \longrightarrow N}{\mathsf{lam}\, x.M \longrightarrow \mathsf{lam}\, x.N}\ \mathsf{s/lam}^x$$

- Revisiting the reducibility candidates
  for well-scoped and well-typed open terms $\Gamma \vdash M \in \mathcal{R}_A$:

$$\begin{array}{lcl}
\mathcal{R}_{\mathbf{i}} & = & \{\Psi \vdash M \mid \Psi \vdash M \text{ halts}\} \\
\mathcal{R}_{A \Rightarrow B} & = & \{\Psi \vdash M \mid \Psi \vdash M \text{ halts and } \forall \Phi \geq_\rho \Psi, N \text{ where } \Phi \vdash N : A, \\
 & & \quad \text{if } (\Phi \vdash N) \in \mathcal{R}_A \text{ then } (\Phi \vdash \mathsf{app}\ M[\rho]\ N) \in \mathcal{R}_B\}
\end{array}$$

Encoding Logical Relations on Open Terms

Definition on paper:

$$\begin{aligned}
\mathcal{R}_{\mathbf{i}} &= \{\Psi \vdash M \mid \Psi \vdash M \text{ halts}\} \\
\mathcal{R}_{A \Rightarrow B} &= \{\Psi \vdash M \mid \Psi \vdash M \text{ halts and } \forall \Phi \geq_\rho \Psi, N \text{ where } \Phi \vdash N : A, \\
&\qquad \text{if } (\Phi \vdash N) \in \mathcal{R}_A \text{ then } (\Phi \vdash \text{app } M[\rho]\ N) \in \mathcal{R}_B\}
\end{aligned}$$

Encoding in Beluga

```
stratified   Reduce : (Ψ:nctx) {A:[tp]} {M:[Ψ ⊢  tm A[]]} type =
| Base : Halts [i] [Ψ  ⊢  M] → Reduce [i] [Ψ ⊢  M]
| Arr :  {M:Ψ  ⊢  tm (arr A[] B[])]}
         Halts [arr A B] [Ψ ⊢ M] →
         ({Φ:nctx} {ρ:[Φ ⊢ Ψ]} {N:[Φ ⊢ tm A[]]}
           Reduce [A] [Φ ⊢  N] → Reduce [B] [Φ ⊢  app M[ρ] N])
         → Reduce [arr A B] [Ψ ⊢  M];
```

See our journal paper discussing case studies [MSCS'16]

POPLMark Reloaded!

Strong normalization of a simply-typed lambda-calculus using Kripke-style logical relations.

POPLMark Reloaded: Goal

Benchmark problems that

- Push the state of the art in the area and outline new areas of research
- Compare systems and mechanized proofs qualitatively
- Understand what infrastructural parts (boilerplate) should be generically supported and factored
- Find bugs in existing proof assistants
- Highlight theoretical limitations of existing proof environments
- Highlight practical limitations of existing proof environments
- Polularize and provide a better understanding of logical relations

Question

Why pick strong normalization for simply-typed lambda-calculus using Kripke-style logical relations?

Question

Why pick strong normalization for simply-typed lambda-calculus using Kripke-style logical relations?

Follow up:

We can prove SN without Kripke-style logical relations and we've already done it.

Witness 1: Lego [Altenkirch'93]

... "following Girard's Proofs and Types"

Characteristic Features:

- Terms are not well-scoped or well-typed
- Candidate relation is untyped and does not enforce well-scoped terms
  $\Longrightarrow$ does not scale to typed-directed evaluatation or equivalence
  $\Longrightarrow$ maybe better techniquues to modularize and structure proof

Witness 2: Abella, ATS/HOAS

... "following Girard's Proofs and Types"

Witness 2: Abella, ATS/HOAS

... "following Girard's Proofs and Types"

- Strictly speaking:

  SN for simply-typed λ-calculus plus one constant that has any type.

- Adding a constant significantly simplifies the proof
- Reducibility of terms only defined on closed terms
- Strictly speaking:

  Show that SN for simply-typed λ-calculus plus one constant implies also SN for open simply-typed λ-terms

A Call for Action

- Be part of formulating and tackling the challenge
- Choose your favorite proof assistant and complete the challenge

Status Report

- 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`

Current and Future Directions

- Lincx: A linear logical framework LF with first-class contexts (A.L. Georges, A. Murawska, S. Otis)[ESOP'17]
- Programming with syntax in existing proof and programming environments (F. Ferreira[ESOP'17])
  Translate contextual objects via a deep embedding
- Coinductive Proofs (e.g. Contextual Equivalence)[ICFP'16]
  (joint work with A. Momigliano, D. Thibodeau [Dale's Festschrift'17])
- Full Dependent Type Theory with Contextual Objects and First-class Contexts (joint work with A. Abel, F. Ferreira, D. Thibodeau, R. Zucchini)

The End

Thank you!

Download prototype and examples at

`http://complogic.cs.mcgill.ca/beluga/`

Thanks go to: Andrew Cave, Joshua Dunfield, Olivier Savary Belanger, Matthias Boespflug, Scott Cooper, Francisco Ferreira, Aidan Marchildon, Stefan Monnier, Agata Murawska, Nicolas Jeannerod, David Thibodeau, Shawn Otis, Rohan Jacob Rao, Shanshan Ruan, Tao Xue

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