Mechanizing Meta-Theory in Beluga

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

Joint work with Andrew Cave

How to mechanize formal systems and proofs?

- Formal systems (given via axioms and inference rules) play an important role when designing languages and more generally software.
- Proofs (that a given property is satisfied) are an integral part of the software (see: certified code, proof-carrying architectures).

How to mechanize formal systems and proofs?

- Formal systems (given via axioms and inference rules) play an important role when designing languages and more generally software.
- Proofs (that a given property is satisfied) are an integral part of the software (see: certified code, proof-carrying architectures).

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

Meta-Theory

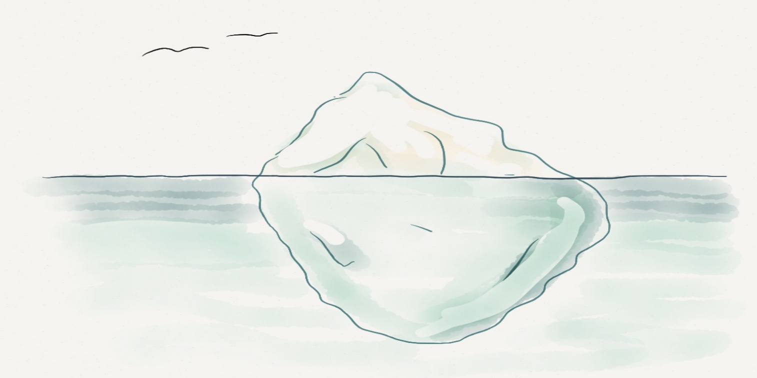
Properties

– Memory Safety:
Program does not crash

– Authenticity:
Communicates only within domain mcgill.ca

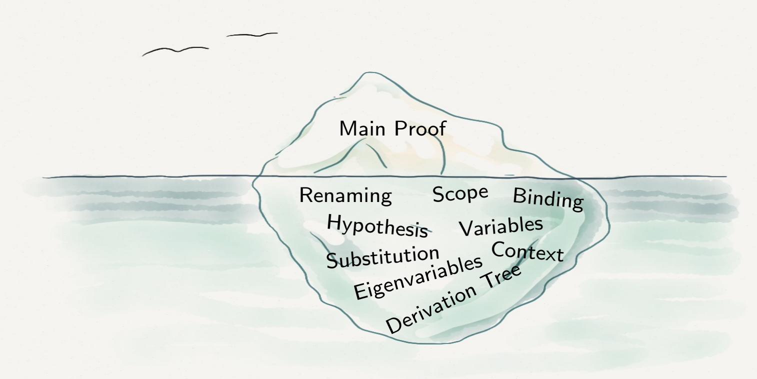
– Type Safety:
Execution of program does not go wrong

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]*

Beluga: Programming Proofs in Context

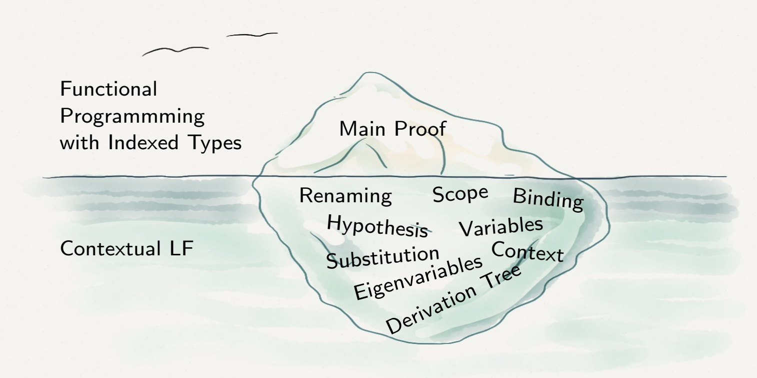
"The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result. Clearly, this is a worthwhile goal." B. Liskov [1974]

BELUGA: Programming Proofs in Context

"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: Proofs by structural Induction = Recursive Programs
  First-order Logic over Contextual LF objects (i.e. Contexts, Derivation trees, Substitutions, ...) together with inductive definitions and induction principles

This Talk

Design and implementation of Beluga

- Introduction
- Basics - Intermediate: Mechanizing Languages and Proofs
  - Type Preservation
  - Uniqueness of Evaluation
  - Type Uniqueness
  - Translating between Lambda-terms to de Bruijn terms
- Advanced: Proofs using logical relations
- 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 $::= x \mid \mathsf{app}\ M\ N \mid \mathsf{lam}\ x{:}A.M$

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

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Simply Typed Lambda-calculus (Gentzen-style)

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

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

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

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \text{E-APP1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \text{E-APP2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \text{E-APP-ABS}$$

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

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

Simply Typed Lambda-calculus with Contexts

Types and Terms

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

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

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

$$\frac{M \longrightarrow M'}{\text{app } M\ N \longrightarrow \text{app } M'\ N}\ \text{E-APP1} \qquad \frac{N \longrightarrow N' \quad V \text{ value}}{\text{app } V\ N \longrightarrow \text{app } V\ N'}\ \text{E-APP2}$$

$$\frac{V \text{ value}}{\text{app } (\text{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \text{E-APP-ABS}$$

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

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x{:}A.M : A \Rightarrow B}\ \text{T-ABS}^x \qquad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N\ A}{\Gamma \vdash \text{app } M\ N : B}\ \text{T-APP}$$

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

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

- What kinds of variables are used?

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \text{E-APP-ABS}$$

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables
- What operations on variables are needed?

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables
- What operations on variables are needed? Substitution and Renaming for bound variable, Substitution for schematic variables, Substitution for hypothesis and eigenvariables

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V \ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \text{E-APP-ABS}$$

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables
- What operations on variables are needed? Substitution and Renaming for bound variable, Substitution for schematic variables, Substitution for hypothesis and eigenvariables
- How should we represent contexts? What properties do contexts have?

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textrm{E-APP-ABS}$$

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables
- What operations on variables are needed? Substitution and Renaming for bound variable, Substitution for schematic variables, Substitution for hypothesis and eigenvariables
- How should we represent contexts? What properties do contexts have? (Structured) sequences, Uniqueness of declaration, Weakening, Substitution lemma, etc.

Derivations Under the Magnifying Glass

Typing rules

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

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables
- What operations on variables are needed? Substitution and Renaming for bound variable, Substitution for schematic variables, Substitution for hypothesis and eigenvariables
- How should we represent contexts? What properties do contexts have? (Structured) sequences, Uniqueness of declaration, Weakening, Substitution lemma, etc.

Any mechanization of proofs must deal with these issues.

Derivations Under the Magnifying Glass

Typing rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \mathsf{lam}\, x{:}A.M : A \Rightarrow B}\ \textsc{T-Abs}^x \qquad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{app}\ M\ N : B}\ \mathit{TApp}$$

Evaluation rules

$$\frac{V \ \text{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Derivations Under the Magnifying Glass

Typing rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \mathsf{lam}\, x{:}A.M : A \Rightarrow B}\ \textsc{T-Abs}^x \qquad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \mathsf{app}\ M\ N : B}\ \mathit{TApp}$$

Evaluation rules

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

In Beluga$^\mu$: Model formal systems and derivation trees in the contextual logical framework LF

- Compact representation of formal systems and derivations
- Higher-order abstract syntax trees and dependent types
  $\rightsquigarrow$ support for α-renaming, substitution, adequate representations

} LF [HHP'93]

- Well-scoped derivation trees
- First-class contexts and substitutions
  + equational theory about substitutions

} Contextual LF [TOCL08,POPL08, PPDP08,LFMTP13]

Step 1: Representing Types and Terms in LF

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

Step 1: Representing Types and Terms in LF

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

LF representation in Beluga

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

```
LF tm: type =
| lam: tp →  (tm → tm)  → tm
| app: tm → tm  → tm;
```

Step 1: Representing Types and Terms in LF

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

LF representation in Beluga

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

```
LF tm: type =
| lam: tp →  (tm → tm)  → tm
| app: tm → tm  → tm;
```

Examples: lam x:nat.x (Identity),
lam x:nat. lam x:nat $\Rightarrow$ nat).x,
lam f:nat $\Rightarrow$ nat. lam g:nat $\Rightarrow$ nat. lam x:nat. app f (app g x)

```
lam nat (λx.x)
lam nat (λx. lam (arr nat nat) (λx.x))
lam (arr nat nat) (λf. lam (arr nat nat) (λg. lam nat (λx. app f (app g x))))
```

- Binding in the object language are modelled using LF functions.
- Inherit α-renaming and single substitutions

Step 2a: Representation of Semantics in LF

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

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \text{E-APP1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \text{E-APP2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x.M)\ V \longrightarrow [V/x]M}\ \text{E-APP-ABS}$$

Step 2a: Representation of Semantics in LF

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

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \text{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \text{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x.M)\ V \longrightarrow [V/x]M}\ \text{E-App-Abs}$$

- Judgments are represented as type families
- Rules are represented as (dependent) types
- Substitution on terms is represented as application in LF

LF representation in Beluga

```
LF step: term → term → type =
| e_app_1   : step M1 M1'
             → step (app M1 M2) (app M1' M2)
| e_app_2   : step M2 M2' → value M1
              → step (app M1 M2) (app M1 M2')
| e_app_abs : value M2
              → step (app (lam M) M2)  (M M2) ;
```

Step 2b: Representation of Typing in LF

Typing Rules

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

Step 2b: Representation of Typing in LF

Typing Rules

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

```
LF hastype: tm → tp → type =
| t_app: hastype M (arr A B)     | t_abs: (Π x:tm.hastype x A → hastype (M x) B)
      → hastype N A                     → hastype (lam A M) (arr A B);
      → hastype (app M N) B
```

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

$$\frac{\begin{array}{c}\overline{x : \text{nat}}^{\,u} \\ \mathcal{D} \\ (\text{lam } y{:}\text{nat}.y) : (\text{nat} \Rightarrow \text{nat})\end{array}}{(\text{lam } x{:}\text{nat}.\text{lam } y{:}\text{nat}.y) : (\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat})}\ \text{t\_lam}^{x,u}$$ is represented as `[⊢ t_abs x.u.D]`

Step 2b: Representation of Typing in LF

Typing Rules

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

```
LF hastype: tm → tp → type =
| t_app: hastype M (arr A B)
       → hastype N A
       → hastype (app M N) B
| t_abs: (Π x:tm.hastype x A → hastype (M x) B)
       → hastype (lam A M) (arr A B);
```

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

$$\frac{\begin{array}{c}\overline{x : \mathsf{nat}}\ u \\ \mathcal{D} \\ (\mathsf{lam}\ y{:}\mathsf{nat}.y) : (\mathsf{nat} \Rightarrow \mathsf{nat})\end{array}}{(\mathsf{lam}\ x{:}\mathsf{nat}.\mathsf{lam}\ y{:}\mathsf{nat}.y) : (\mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat})}\ \mathsf{t\_lam}^{x,u}$$ is represented as `[⊢ t_abs x.u.D[x,u]]`

Proofs by Induction: Type Preservation

Theorem

If $\mathcal{D} :: \ \vdash M : B$ and $\mathcal{S} :: M \longrightarrow N$ then $\vdash N : B$.

Proof.

By structural induction on the derivation $\mathcal{S} :: M \longrightarrow N$.

Case: $\mathcal{S} = \dfrac{\begin{array}{c}\mathcal{V}\\ V \text{ value}\end{array}}{\text{app }(\text{lam } x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$

$\mathcal{D} \ :: \ \vdash \text{app }(\text{lam } x{:}A.M)\ V : B$ — by assumption

$\mathcal{D}_1 :: \ \vdash \text{lam } x{:}A.M : A \Rightarrow B$ and $\mathcal{D}_2 :: \ \vdash V : A$ — by inversion using rule $\textsc{T-App}$

$\mathcal{D}' :: x : A \vdash M : B$ — by inversion on $\mathcal{D}_1$ using rule $\textsc{T-Abs}$

$:: \ \vdash [V/x]M : B$ — by substitution lemma using V and $\mathcal{D}_2$ in $\mathcal{D}$.

□

Beluga$^\mu$: Proofs as Programs

Functional programming with indexed types [POPL'08,POPL'12]

Proof term language for first-order logic over a specifc domain (= contextual LF) together inductive definitions (= relations) about domain objects and domain-specific induction principle [TLCA'15]

Beluga$^{\mu}$: Proofs as Programs

Functional programming with indexed types [POPL'08,POPL'12]

Proof term language for first-order logic over a specifc domain (= contextual LF) together inductive definitions (= relations) about domain objects and domain-specific induction principle [TLCA'15]

On paper proof	Proofs as functions in Beluga
Theorem	Type
Case analysis	Case analysis and pattern matching
Inversion	Pattern matching using let-expression
Induction Hypothesis	Recursive call

Beluga programs manipulate directly derivation trees and contexts

Type preservation: Theorems as Types

Theorem

If $\mathcal{D} :: \ \vdash M : B$ and $\mathcal{S} :: M \longrightarrow N$ then $\vdash N : B$.

is translate to

Computation-level Type in Beluga

$[\ \vdash\ \texttt{hastype M B}] \to [\ \vdash\ \texttt{step M N}] \to [\ \vdash\ \texttt{hastype N B}]$

Remark:

- $[\ \vdash\ \texttt{hastype M B}]$ is a contextual type. It stands for a closed typing derivation `hastype M B`.
- `M`, `N`, and `B` are implicitly quantified at the outside. Beluga infers the type of these free variables.

Type preservation: Proofs are Programs

Theorem: If $\mathcal{D} :: \ \vdash M : B$ and $\mathcal{S} :: M \longrightarrow N$ then $\vdash N : B$.
By structural induction on the derivation $\mathcal{S} :: M \longrightarrow N$.

$$\textbf{Case: } \mathcal{S} = \dfrac{\begin{array}{c}\mathcal{V} \\ V \text{ value}\end{array}}{\text{app (lam } x{:}A.M)\ V \longrightarrow [V/x]M}\ \text{E-APP-ABS}$$

$\mathcal{D} \ :: \ \vdash \text{app (lam } x{:}A.M)\ V : B$ — by assumption
$\mathcal{D}_1 :: \ \vdash \text{lam } x{:}A.M : A \Rightarrow B$ and $\mathcal{D}_2 :: \ \vdash V : A$ — by inversion using rule T-APP
$\mathcal{D}' :: x : A \vdash M : B$ — by inversion on $\mathcal{D}_1$ using rule T-ABS
$:: \ \vdash [V/x]M : B$ — by substitution lemma using V and $\mathcal{D}_2$ in $\mathcal{D}'$.

Computation in Beluga

```
rec tps: [ ⊢ hastype M T] → [ ⊢ step M N] → [ ⊢ hastype N T] =
/ total s (tps m t n d s)/
fn d ⇒ fn s ⇒ case s of
| [ ⊢ e_app_1 S1] ⇒ ?
| [ ⊢ e_app_2 S2 _ ] ⇒ ?
| [ ⊢ e_app_abs V] ⇒
  let [ ⊢ t_app (t_abs λx.λu. D') D2] = d in
    [ ⊢  D'[_, D2] ]
```

Type preservation - Full Proof

```
rec tps: [ ⊢ hastype M T] → [ ⊢ step M N] → [ ⊢ hastype N T] =
/ total s (tps m t n d s)/
fn d ⇒ fn s ⇒ case s of
| [ ⊢ e_app_1 S1] ⇒
  let [ ⊢ t_app D1 D2] = d in
  let [ ⊢ F1] = tps [ ⊢ D1] [ ⊢ S1] in
    [ ⊢ t_app F1 D2 ]
| [ ⊢ e_app_2 S2 _ ] ⇒
  let [ ⊢ t_app D1 D2] = d in
  let [ ⊢ F2] = tps [ ⊢ D2] [ ⊢ S2] in
    [ ⊢ t_app D1 F2]
| [ ⊢ e_app_abs V] ⇒
  let [ ⊢ t_app (t_abs λx.λu. D) D2] = d in
    [ ⊢ D[_, D2]]
;
```

- Totality declaration states what argument is decreasing
- We check that all cases are covered and all recursive calls are on smaller arguments
- Appealing to the IH corresponds to the recursive call

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions

- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions
  - Equality
  - Falsehood
- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees
  2. Proofs by induction involving falsehood

Uniqueness of Evaluation

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Theorem

If $\mathcal{S}_1 :: M \longrightarrow N_1$ and $M \longrightarrow N_2$ then $N_1 = N_2$.

Uniqueness of Evaluation

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Theorem

If $\mathcal{S}_1 :: M \longrightarrow N_1$ and $M \longrightarrow N_2$ then $N_1 = N_2$.

By structural induction on $\mathcal{S}_1 :: M \longrightarrow N_1$.

Case $\mathcal{S}_1 = \dfrac{W\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ W \longrightarrow [W/x]M}\ \textsc{E-App-Abs}$

Uniqueness of Evaluation

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Theorem

If $\mathcal{S}_1 :: M \longrightarrow N_1$ and $M \longrightarrow N_2$ then $N_1 = N_2$.

By structural induction on $\mathcal{S}_1 :: M \longrightarrow N_1$.

Case $\mathcal{S}_1 = \dfrac{W\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ W \longrightarrow [W/x]M}\ \textsc{E-App-Abs}$

Sub-Case 1: $\mathcal{S}_2 = \dfrac{W\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ W \longrightarrow [W/x]M}\ \textsc{E-App-Abs}$

$[W/x]M = [W/x]M$ by reflexivity

Uniqueness of Evaluation

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Theorem

If $\mathcal{S}_1 :: M \longrightarrow N_1$ and $M \longrightarrow N_2$ then $N_1 = N_2$.

By structural induction on $\mathcal{S}_1 :: M \longrightarrow N_1$.

Case $\mathcal{S}_1 = \dfrac{W\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ W \longrightarrow [W/x]M}\ \textsc{E-App-Abs}$

Sub-Case 2: $\mathcal{S}_2 = \dfrac{\begin{matrix}\mathcal{S} \\ \mathsf{lam}\ x.M \longrightarrow M'\end{matrix}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ N \longrightarrow M'\ N}\ \textsc{E-App1}$

lam $x.M$ value — by definition

$\perp$ — since there is no derivation for $\mathcal{S}$ (Lemma)

Uniqueness of Evaluation

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \textsc{E-App1} \qquad \frac{N \longrightarrow N' \quad V\ \mathsf{value}}{\mathsf{app}\ V\ N \longrightarrow \mathsf{app}\ V\ N'}\ \textsc{E-App2}$$

$$\frac{V\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam} x{:}A.M)\ V \longrightarrow [V/x]M}\ \textsc{E-App-Abs}$$

Theorem

If $\mathcal{S}_1 :: M \longrightarrow N_1$ and $M \longrightarrow N_2$ then $N_1 = N_2$.

By structural induction on $\mathcal{S}_1 :: M \longrightarrow N_1$.

Case $\mathcal{S}_1 = \dfrac{W\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ W \longrightarrow [W/x]M}\ \textsc{E-App-Abs}$

Sub-Case 3: $\mathcal{S}_2 = \dfrac{\overset{\mathcal{S}}{W \longrightarrow N'} \quad (\mathsf{lam}\ x.M)\ \mathsf{value}}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ W \longrightarrow \mathsf{app}\ (\mathsf{lam}\ x.M)\ N'}\ \textsc{E-App2}$

$\perp$ since W value there is no derivation for $\mathcal{S}$ (Lemma)

Lemma

If $M \longrightarrow M'$ *and* M value *then* $\perp$.

Step 1: Encoding Equality and Bottom

Equality (not built-in in Beluga)

LF representation in Beluga

```
LF equal : term → term → type =
| refl: equal M M;
```

Alternative: Define it structurally.

Bottom (Falsehood) (not built-in in Beluga)

LF representation in Beluga

```
not_possible : type.
```

Define an empty type with no constructors.

Step 2a: Encoding "Values don't step"

Lemma

If $M \longrightarrow M'$ and M value then $\perp$.

```
rec values_dont_step: [ ⊢ step M M' ] → [ ⊢ value M ] → [ ⊢ not_possible]=
/ total v (values_dont_step m m' s v)/
fn s ⇒ fn v ⇒ case v of
| [ ⊢ v_lam] ⇒ impossible s;
```

- **impossible** `s` is syntactic sugar for **case** `s` **of** `{}`, i.e. a case-expression with no branches.
- Corresponds to having derived $\perp$ in the on-paper proof

Step 2b: Encoding Uniqueness of Values

Theorem

If $\mathcal{S}_1 :: M \longrightarrow N_1$ and $M \longrightarrow N_2$ then $N_1 = N_2$.

```
rec unique : [⊢ step M N1] → [⊢ step M N2] → [⊢ equal N1 N2 ] =
/ total s1 (unique m m1 m2 s1)/
fn s1 ⇒ fn s2 ⇒ case s1 of
| [ ⊢ e_app_1 S]       ⇒  ?
| [ ⊢ e_app_2 S V ]    ⇒  ?
| [ ⊢ e_app_abs V]     ⇒  case s2 of
   | [⊢ e_app_abs _ ] ⇒ [ ⊢ refl]
   | [⊢ e_app_1 S]    ⇒ impossible  values_dont_step [⊢ S] [⊢ v_lam]
   | [⊢ e_app_2 S _ ] ⇒ impossible  values_dont_step [⊢ S] [⊢ V]
;
```

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions
  - Encoding equality
  - Encoding falsehood
- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees
  2. Proofs using falsehood

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions
  - Encoding equality
  - Encoding falsehood
- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees
  2. Proofs using falsehood
  3. Proofs by induction on open derivation tress

Type Uniqueness

Theorem

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

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B}\ \text{T-A}{\small BS}^x \quad \mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1 \\ \Gamma, x{:}A \vdash M : B'\end{array}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B'}\ \text{T-A}{\small BS}^x$$

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x{:}A \vdash M : B}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B}\ \text{T-ABS}^x \quad \mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma, x{:}A \vdash M : B'}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B'}\ \text{T-ABS}^x$$

$\mathcal{E} :$ eq B B' by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B}\ \text{T-ABS}^x \quad \mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1 \\ \Gamma, x{:}A \vdash M : B'\end{array}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B'}\ \text{T-ABS}^x$$

$\mathcal{E} :$ eq $B\ B'$ by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

$\mathcal{E} :$ eq $B\ B$ and $B = B'$ by inversion using reflexivity

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x{:}A \vdash M : B}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B}\ \textsc{T-Abs}^x \quad \mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma, x{:}A \vdash M : B'}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B'}\ \textsc{T-Abs}^x$$

$\mathcal{E} :$ eq $B\ B'$ by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

$\mathcal{E} :$ eq $B\ B$ and $B = B'$ by inversion using reflexivity

Therefore there is a proof for eq $(A \Rightarrow B)\ (A \Rightarrow B')$ by reflexivity.

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B}\ \text{T-ABS}^x \qquad \mathcal{C} = \frac{\begin{array}{c}\mathcal{C}_1 \\ \Gamma, x{:}A \vdash M : B'\end{array}}{\Gamma \vdash \mathsf{lam}\ x{:}A.M : A \Rightarrow B'}\ \text{T-ABS}^x$$

$\mathcal{E} :$ eq $B\ B'$ by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

$\mathcal{E} :$ eq $B\ B$ and $B = B'$ by inversion using reflexivity

Therefore there is a proof for eq $(A \Rightarrow B)\ (A \Rightarrow B')$ by reflexivity.

Case 2

$$\mathcal{D} = \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A}$$

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x{:}A \vdash M : B\end{array}}{\Gamma \vdash \text{lam } x{:}A.M : A \Rightarrow B}\ \text{T-ABS}^x \qquad \mathcal{C} = \dfrac{\begin{array}{c}\mathcal{C}_1\\ \Gamma, x{:}A \vdash M : B'\end{array}}{\Gamma \vdash \text{lam } x{:}A.M : A \Rightarrow B'}\ \text{T-ABS}^x$$

$\mathcal{E} :$ eq B B' by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

$\mathcal{E} :$ eq B B and $B = B'$ by inversion using reflexivity

Therefore there is a proof for eq $(A \Rightarrow B)$ $(A \Rightarrow B')$ by reflexivity.

Case 2

$$\mathcal{D} = \dfrac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \mathcal{C} = \dfrac{x{:}B \in \Gamma}{\Gamma \vdash x : B}$$

Type Uniqueness

Theorem

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

Induction on first typing derivation $\mathcal{D}$.

Case 1

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x{:}A \vdash M : B}}{\Gamma \vdash \text{lam } x{:}A.M : A \Rightarrow B}\ \text{T-ABS}^x \qquad \mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma, x{:}A \vdash M : B'}}{\Gamma \vdash \text{lam } x{:}A.M : A \Rightarrow B'}\ \text{T-ABS}^x$$

$\mathcal{E} :$ eq $B\ B'$ — by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

$\mathcal{E} :$ eq $B\ B$ and $B = B'$ — by inversion using reflexivity

Therefore there is a proof for eq $(A \Rightarrow B)\ (A \Rightarrow B')$ by reflexivity.

Case 2

$$\mathcal{D} = \dfrac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \mathcal{C} = \dfrac{x{:}B \in \Gamma}{\Gamma \vdash x : B}$$

Every variable x is associated with a unique typing assumption (property of the context), hence $A = B$.

Step 2a: Theorem as Type

Step 2a: Theorem as Type

Theorem

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

Step 2a: Theorem as Type

Theorem

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

is represented as

Computation-level Type in Beluga

```
{γ:ctx}[γ ⊢ hastype M A ] → [γ ⊢ hastype M B ] → [ ⊢ eq A B]
```

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas

Step 2a: Theorem as Type

Theorem

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

is represented as

Computation-level Type in Beluga

$\{\gamma$:ctx$\}[\gamma \vdash$ hastype M A[]] $\rightarrow$ $[\gamma \vdash$ hastype M B[]] $\rightarrow$ [$\vdash$ eq A B]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas

  schema ctx = some [t:tp] **block** x:term, u:hastype x T;

- M is a term that depends on γ; it has type [$\gamma \vdash$ term]

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

  Recall: All meta-variables are associated with a substitution.

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

  $\rightsquigarrow$ A and B are associated with a weakening substitution

Step 2a: Theorem as Type

Theorem

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

is represented as

Computation-level Type in Beluga

`{γ:ctx}[γ ⊢ hastype M A[] ] → [γ ⊢ hastype M B[] ] → [ ⊢ eq A B]`

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas

  schema `ctx = some [t:tp]` **block** `x:term, u:hastype x T;`

- `M` is a term that depends on γ; it has type `[γ ⊢ term]`

  `A` and `B` are types that are closed; they have type `[ ⊢ tp]`

  Recall: 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:term, u:hastype x t;
```

- The context $x : \mathrm{nat},\ y : \mathrm{nat} \Rightarrow \mathrm{nat}$ is represented as
  `b1:block (x:term,u:hastype x nat),b2:block (y:term,v:hastype y (arr nat nat))`

Intrinsic Support for Contexts

```
schema ctx =  some [t:tp] block  x:term, u:hastype x t;
```

- The context $x : \text{nat},\ y : \text{nat} \Rightarrow \text{nat}$ is represented as
  `b1:block (x:term,u:hastype x nat),b2:block (y:term,v:hastype y (arr nat nat))`
- Well-formedness: `b1:block  x:term,u:hastype y nat` is ill-formed.
  `x:term, y:term, u:hastype x nat` is ill-formed.

Intrinsic Support for Contexts

```
schema ctx =  some [t:tp] block  x:term, u:hastype x t;
```

- The context $x : \text{nat},\ y : \text{nat} \Rightarrow \text{nat}$ is represented as
  `b1:block (x:term,u:hastype x nat),b2:block (y:term,v:hastype y (arr nat nat))`
- Well-formedness: `b1:block x:term,u:hastype y nat` is ill-formed.
  `x:term, y:term, u:hastype x nat` is ill-formed.
- Declarations are unique: `b1` is different from `b2`

Intrinsic Support for Contexts

```
schema ctx =  some [t:tp] block  x:term, u:hastype x t;
```

- The context x : nat, y : nat $\Rightarrow$ nat is represented as
  `b1:block (x:term,u:hastype x nat),b2:block (y:term,v:hastype y (arr nat nat))`
- Well-formedness: `b1:block x:term,u:hastype y nat` is ill-formed.
  `x:term, y:term, u:hastype x nat` is ill-formed.
- Declarations are unique: `b1` is different from `b2`
  `b1.1` is different from `b2.1`

Intrinsic Support for Contexts

```
schema ctx = some [t:tp] block x:term, u:hastype x t;
```

- The context $x : \text{nat},\ y : \text{nat} \Rightarrow \text{nat}$ is represented as
  `b1:block (x:term,u:hastype x nat),b2:block (y:term,v:hastype y (arr nat nat))`
- Well-formedness: `b1:block x:term,u:hastype y nat` is ill-formed.
  `x:term, y:term, u:hastype x nat` is ill-formed.
- Declarations are unique: `b1` is different from `b2`
  `b1.1` is different from `b2.1`
- Later declarations overshadow earlier ones

Intrinsic Support for Contexts

```
schema ctx = some [t:tp] block x:term, u:hastype x t;
```

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

Type Uniqueness

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype M B[]] → [⊢ eq A B] =
```

Type Uniqueness

rec `unique:(`γ`:ctx)[`γ $\vdash$ `hastype M A[]]` $\rightarrow$ `[`γ $\vdash$ `hastype M B[]]` $\rightarrow$ `[`$\vdash$ `eq A B] =`

fn `d` $\Rightarrow$ **fn** `c` $\Rightarrow$ **case** `d` **of**

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype 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]
```

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype 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:term, u:hastype x _) ⊢ D[…, b.1, b.2]]
                        [γ,b ⊢ C[…,b.1, b.2]] in
   [ ⊢ ref]
```

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype 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:term, u:hastype x _) ⊢ D[…, b.1, b.2]]
                        [γ,b ⊢ C[…,b.1, b.2]] in
   [ ⊢ ref]

| [γ ⊢ #q.2] ⇒           % d : oft #q.1 T       % Assumption Case
  let [γ ⊢ #r.2] = c in  % c : oft #r.1  S
    [ ⊢ ref] ;
```

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype 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:term, u:hastype x _) ⊢ D[…, b.1, b.2]]
                        [γ,b ⊢ C[…,b.1, b.2]] in
   [ ⊢ ref]

| [γ ⊢ #q.2] ⇒            % d : oft #q.1 T       % Assumption Case
  let [γ ⊢ #r.2] = c in   % c : oft #r.1  S
    [ ⊢ ref] ;
```

Recalll:

#q:**block** x:term, u:hastype x T

#r:**block** x:term, u:hastype x S

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype 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:term, u:hastype x _) ⊢ D[…, b.1, b.2]]
                        [γ,b ⊢ C[…,b.1, b.2]] in
   [ ⊢ ref]

| [γ ⊢ #q.2] ⇒             % d : oft #q.1 T        % Assumption Case
  let [γ ⊢ #r.2] = c in    % c : oft #r.1  S
    [ ⊢ ref] ;
```

Recalll:

`#q:`**block** `x:term, u:hastype x T`

`#r:`**block** `x:term, u:hastype x S`

We also know: `#r.1` $=$ `#q.1`

Type Uniqueness

```
rec unique:(γ:ctx)[γ ⊢ hastype M A[]] → [γ ⊢ hastype 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:term, u:hastype x _) ⊢ D[…, b.1, b.2]]
                        [γ,b ⊢ C[…,b.1, b.2]] in
   [ ⊢ ref]

| [γ ⊢ #q.2] ⇒            % d : oft #q.1 T     % Assumption Case
  let [γ ⊢ #r.2] = c in   % c : oft #r.1  S
    [ ⊢ ref] ;
```

Recalll:

#q:**block** x:term, u:hastype x T

#r:**block** x:term, u:hastype x S

We also know: #r.1 = #q.1

Therefore: T = S

Key Ideas

- Contexts are first-class and are classified by schemas
- Contextual Types/Objects characterize derivation trees that depend on assumptions
- Parameter Variables distinguish between variables and general objects
- Simultaneous Substitutions allow us to move between contexts (Identity, Weakening, Uncurrying)
- Totality Checker verifies that all cases, including the variable cases, are covered and all recursive calls are well-founded.

Our proof/programming language has not changed - instead we have extended LF to model contexts, contextual objects, simultaneous substitutions, meta-variables and parameter variables.

Brief Comparison

- Twelf [Pf,Sch'99]: Encode proofs as relations within LF
  - Requires lemma to prove injectivity of `arr` constructor.
  - No explicit contexts
  - Parameter case folded into abstraction case
- Delphin [Sch,Pos'08]: Encode proofs as functions
  - Requires lemma to prove injectivity of constructor
  - Cannot express that types `T` and `S` and `eq T S` are closed.
  - Variable carrying continuation as extra argument to handle context
- Abella [Gacek'08]: 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 .
  - Equality built-into the logic
  - Contexts are represented as lists
  - Requires lemmas about these lists (for example that all assumptions occur uniquely)

Translation between lambda-terms and de Bruijn

Translation between lambda-terms and de Bruijn

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions
  - Encoding equality
  - Encoding falsehood

- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees
  2. Proofs using falsehood
  3. Proofs by induction on open derivation trees

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions
  - Encoding equality
  - Encoding falsehood
  - Inductive and stratified definitions
- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees
  2. Proofs using falsehood
  3. Proofs by induction on open derivation trees
  4. Proofs by logical relations

Translation between lambda-terms and de Bruijn

Challenging Benchmark: Proofs by Logical Relations

Weak Normalization of the simply-typed Lambda-Calculus

> *"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: Proofs by Logical Relations

Weak Normalization of the simply-typed Lambda-Calculus

"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]*

- Binders: lambda-binder, $\forall$ in reducibility definition, quantification over substitutions and contexts
- Contexts: Uniqueness of assumptions, weakening, etc.
- Simultanous substitution and algebraic properties: Substitution lemma, composition, decomposition, associativity, identity, etc.

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

a dozen such properties are needed

The Set-up: Simply Typed Lambda-Calculus - revisited

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: $\boxed{M \longrightarrow M'}$

Call-by-Name (to simplify things)

$$\frac{}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ N \longrightarrow [N/x]M}\ \mathsf{s_beta} \qquad \frac{}{M \longrightarrow M}\ \mathsf{s_refl}$$

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \mathsf{s_app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \mathsf{s_trans}$$

The Set-up: Simply Typed Lambda-Calculus - revisited

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: $\boxed{M \longrightarrow M'}$

Call-by-Name (to simplify things)

$$\frac{}{\mathsf{app}\ (\mathsf{lam}\ x.M)\ N \longrightarrow [N/x]M}\ \mathsf{s_beta} \qquad \frac{}{M \longrightarrow M}\ \mathsf{s_refl}$$

$$\frac{M \longrightarrow M'}{\mathsf{app}\ M\ N \longrightarrow \mathsf{app}\ M'\ N}\ \mathsf{s_app} \qquad \frac{M \longrightarrow M' \quad M' \longrightarrow N}{M \longrightarrow N}\ \mathsf{s_trans}$$

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

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

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 $\mathcal{R}_A$

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

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:

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

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

$[\sigma](\mathrm{lam}\ x.M) = \mathrm{lam}\ x.([\sigma, x/x]M)$ — 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

Step 1a: Represent Types and Lambda-terms in LF

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

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

Typing rules

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

Intrinsically typed Term Representation

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;
```

Step 1a: Represent Types and Lambda-terms in LF

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

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

Typing rules

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

Intrinsically typed Term Representation

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;
```

Step 1a: Represent Semantics in LF

Step 1b: Reducibility Candidates as Stratified Types

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

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

Step 1b: Reducibility Candidates as Stratified 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]` are contextual types [TOCL'08].
- Note: $\rightarrow$ is overloaded.
  - $\rightarrow$ is the LF function space : binders in the object language are modelled by LF functions (used inside `[ ]`)
  - $\rightarrow$ is a computation-level function (used outside `[ ]`)
- Not strictly positive definition, but stratified.

Step 1b: Reducibility Candidates as Inductive Types

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

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

Step 1b: Reducibility Candidates as Inductive 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 τ are first-class and have type $\Psi \vdash \Phi$ providing a mapping from Φ to Ψ.

Step 2: Theorems as Types

Lemma (Backward closed)

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

```
rec closed : [⊢ mstep 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] ] = ? ;
```

Step 2: Theorems as Types

Lemma (Backward closed)

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

```
rec closed : [⊢ mstep 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] ] = ? ;
```

Step 2: Fundamental Lemma

Step 2: Fundamental Lemma

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

Step 2: Fundamental Lemma

```
rec closed : [ ⊢mstep 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
```

Step 2: Fundamental Lemma

```
rec closed : [ ⊢mstep 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)
```

Step 2: Fundamental Lemma

```
rec closed : [ ⊢mstep 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)))
```

Step 2: Fundamental Lemma

```
rec closed : [ ⊢mstep 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
```

Step 2: Fundamental Lemma

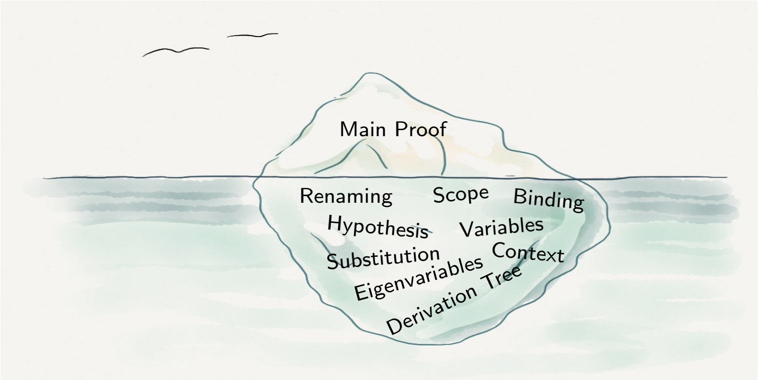
```
rec closed : [ ⊢mstep M M'] →Reduce [ ⊢A] [ ⊢M'] →Reduce [ ⊢A] [ ⊢M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢tm A[]]} RedSub [ ⊢σ] →Reduce [ ⊢A] [ ⊢M[σ]] =
mlam Γ⇒mlam M ⇒fn rs ⇒ case [Γ ⊢M] of
| [Γ ⊢#p] ⇒lookup [Γ] [Γ ⊢#p] rs                          % Variable

| [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

More Examples using Stratified and Inductive Types

- Proofs using logical relations
  Algorithmic Equality in LF [LFMTP'15]
- Proofs using context relations
  Completeness of algorithmic and declarative equality for lambda-terms [JAR'15]
- Program transformations
  - Type preserving Closure Conversion and Hoisting [CPP'13]
  - Normalization by Evaluation [POPL'12]

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]*

Revisiting the Design of Beluga

- Top : Functional programming with indexed types [POPL'08,POPL'12]

Case analysis	Case analysis and pattern matching
Inversion	Pattern matching using let-expression
Induction hypothesis	Recursive call

- Bottom: Contextual LF

On paper proof	In Beluga [IJCAR'10,CADE'15]
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
Substitutions (composition, identity)	Substitution type [LFMTP'13]

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

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.

Lessons Learned

- How to specify formal systems.
  - Binders in the object language are modelled using LF functions
  - Hypothetical and parametric derivations are modelled using LF functions
  - Encoding equality
  - Encoding falsehood
  - Inductive and stratified definitions
- How to write proofs as recursive functions using pattern matching
  1. Proofs by induction on closed derivation trees
  2. Proofs using falsehood
  3. Proofs by induction on open derivation trees
  4. Proofs by logical relations

Current Work

- 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`

- Coinduction in Beluga (D. Thibodeau, A. Cave)
  Extending work on simply-typed copatterns [POPL'13] to Beluga
  Long term: reason about reactive systems [POPL'14]
- Case study: Certified compiler (O. Savary Belanger) [CPP'13]
- Extending Beluga to full dependent types (A. Cave)
- Type reconstruction (F. Ferreira [PPDP'14] and [JFP'13])
- ORBI - Benchmarks for comparing systems supporting HOAS encodings [JAR'15,LFMTP'15] (A. Felty, A. Momigliano, March 2015)

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*