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).

### Properties

- **Memory Safety:** Program does not crash
- **Authenticity:** Communicates only within domain mcgill.ca
- **Type Safety:** Execution of program does not go wrong
“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]
“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]
"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]
“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 current work

“The limits of my language mean the limits of my world.”
- L. Wittgenstein
Simply Typed Lambda-calculus (Gentzen-style)

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

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

Evaluation Judgment: $M \rightarrow M'$ read as “$M$ steps to $M'$”

- $M \rightarrow M'$ \implies $\text{app } M \ N \rightarrow \text{app } M' \ N$ E-APP1
- $N \rightarrow N'$ \implies $\text{app } V \ N \rightarrow \text{app } V \ N'$ E-APP2
- $\text{app } (\text{lam } x:A.M) \ V \rightarrow [V/x]M$ E-APP-ABS
Simply Typed Lambda-calculus (Gentzen-style)

Types $A, B ::= \ i \mid A \Rightarrow B$

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

Evaluation Judgment: $M \rightarrow M'$ read as "$M$ steps to $M'$"

- **E-APP1**: $M \rightarrow M'$
  $\frac{M \rightarrow M'}{\text{app } M N \rightarrow \text{app } M' N}$

- **E-APP2**: $N \rightarrow N'$
  $\frac{N \rightarrow N'}{\text{app } V N \rightarrow \text{app } V N'}$

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

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

- **T-ABS**: $\frac{\text{lam } x: A. M : A \Rightarrow B}{\text{lam } x: A. M : A \Rightarrow B}$
- **T-APP**: $\frac{M : A \Rightarrow B \quad N : A}{\text{app } M N : B}$
Simply Typed Lambda-calculus with Contexts

Types and Terms

Types \( A, B ::= \) \( i \)
\[ | \quad A \Rightarrow B \]

Terms \( M, N ::= \) \( x \mid c \)
\[ | \quad \text{lam } x : A . M \]
\[ | \quad \text{app } M N \]

Evaluation Judgment: \( M \rightarrow M' \)
read as “\( M \) steps to \( M' \)”

\[
\frac{M \rightarrow M'}{\text{app } M N \rightarrow \text{app } M' N} \quad \text{E-APP1}
\]
\[
\frac{N \rightarrow N'}{\text{app } V N \rightarrow \text{app } V' N} \quad \text{E-APP2}
\]
\[
\frac{V \text{ value}}{\text{app } (\text{lam } x : A . M) V \rightarrow [V/x]M} \quad \text{E-APP-ABS}
\]

Typing Judgment: \( \Gamma \vdash M : A \)
read as “\( M \) has type \( A \) in context \( \Gamma \)”

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

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

\[
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x : A.M : A \Rightarrow B} \quad \frac{\Gamma \vdash M : A \Rightarrow B}{\Gamma \vdash \text{app } M \ N : B}
\]

Evaluation rules

\[
\frac{V \text{ value}}{\text{app (lam } x : A.M) \ V \rightarrow [V/x]M}
\]
Derivations Under the Magnifying Glass

Typing rules

\[
\begin{align*}
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} & \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B} & \quad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \text{app } M N : B}
\end{align*}
\]

T-ABS

T-APP

Evaluation rules

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

- What kinds of variables are used?

  | Bound variables, Eigenvariables, Schematic variables, Context variables |
  | Substitution and Renaming for bound variable, Substitution for schematic variables, Substitution for hypothesis and eigenvariables |
  | (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

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

Evaluation rules

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

- What kinds of variables are used? Bound variables, Eigenvariables, Schematic variables, Context variables
Derivations Under the Magnifying Glass

Typing rules

\[ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \]

\[ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : A \Rightarrow B} \quad \text{T-ABS}^x \]

\[ \frac{\Gamma \vdash M : A \Rightarrow B, \Gamma \vdash N : B}{\Gamma \vdash \text{app} M N : B} \quad \text{T-APP} \]

Evaluation rules

\[ \frac{V \text{ value}}{\text{app} (\lambda x : A. M) V \rightarrow [V/x]M} \quad \text{E-APP-ABS} \]

- 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

\[
\begin{array}{c}
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x : A. M : A \Rightarrow B} \quad \text{T-ABS}^x \\
\frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \text{app } M N : B} \quad \text{T-APP}
\end{array}
\]

Evaluation rules

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

- 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**
Derivations Under the Magnifying Glass

Typing rules

\[ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \]

\[ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam}x:M : A \Rightarrow B} \quad \text{T-ABS}^x \]

\[ \frac{\Gamma \vdash N : B}{\Gamma \vdash \text{app} M N : B} \quad \text{T-APP} \]

Evaluation rules

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

- 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

\[
\begin{align*}
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x : A.M : A \Rightarrow B} & \quad \text{T-ABS}^x \\
\frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \text{app } M N : B} & \quad \text{T-APP}
\end{align*}
\]

Evaluation rules

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

- 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? (Structured) sequences, Uniqueness of declaration, Weakening, Substitution lemma, etc.**
Derivations Under the Magnifying Glass

Typing rules

\[
\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x : A. M : A \Rightarrow B} \quad \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \text{app } M N : B}
\]

Evaluation rules

\[
\frac{\text{app } (\text{lam } x : A. M) \ V \longrightarrow \ [V/\!\!x]M}{\text{value}}
\]

- 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

\[ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \]

\[ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam } x: A. M : A \Rightarrow B} \quad \text{T-Abs}^x \]

\[ \frac{\Gamma \vdash M : A \Rightarrow B, \Gamma \vdash N : B}{\Gamma \vdash \text{app } M \ N : B} \quad \text{TApp} \]

Evaluation rules

\[ \frac{V \text{ value}}{\text{app } (\text{lam } x : A. M) \ V \longrightarrow [V/x]M} \quad \text{E-APP-ABS} \]
Derivations Under the Magnifying Glass

Typing rules

\[ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \]
\[ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{lam} \, x : A. M : A \Rightarrow B} \] \quad \text{T-Abs}^x
\[ \frac{\Gamma \vdash M : A \Rightarrow B \quad \Gamma \vdash N : B}{\Gamma \vdash \text{app} \, M \, N : B} \] \quad \text{TApp}

Evaluation rules

\[ \frac{V \text{ value}}{\text{app} \, (\text{lam} \, x : A. M) \, V \rightarrow [V/x]M} \] \quad \text{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
  \(\sim\) support for \(\alpha\)-renaming, substitution, adequate representations
- Well-scoped derivation trees
- First-class contexts and substitutions
  + equational theory about substitutions
Step 1: Representing Types and Terms in LF

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

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

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

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

#### LF representation in Beluga

**LF**

**tp: type =**
- nat: tp
- arr: tp $\rightarrow$ tp $\rightarrow$ tp

**tm: type =**
- lam: tp $\rightarrow$ (tm $\rightarrow$ tm) $\rightarrow$ tm
- app: tm $\rightarrow$ tm $\rightarrow$ tm
Step 1: Representing Types and Terms in LF

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

Terms $M ::= x | \text{lam } x:A.M | \text{app } M N$

LF representation in Beluga

- **Types**
  - LF tp: type =
    - | nat: tp
    - | arr: tp → tp → tp;

- **Terms**
  - 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 ($\lambda x.x$)
- lam nat ($\lambda x. \text{lam (arr nat nat)} (\lambda x.x)$)
- lam (arr nat nat) ($\lambda f. \text{lam (arr nat nat)} (\lambda g. \text{lam nat} (\lambda x. \text{app } f \ (\text{app } g \ x)))$)

- Binding in the object language are modelled using **LF functions**.
- Inherit $\alpha$-renaming and single substitutions
Step 2a: Representation of Semantics in LF

Evaluation Judgment: \[ M \rightarrow M' \]

\[
\frac{M \rightarrow M'}{\text{app } M \, N \rightarrow \text{app } M' \, N} \quad \text{E-APP1}
\]

\[
\frac{N \rightarrow N' \quad V \text{ value}}{\text{app } V \, N \rightarrow \text{app } V \, N'} \quad \text{E-APP2}
\]

\[
\frac{V \text{ value}}{\text{app } (\text{lam} x. M) \, V \rightarrow [V/x]M} \quad \text{E-APP-ABS}
\]
Step 2a: Representation of Semantics in LF

Evaluation Judgment: \[ M \rightarrow M' \]

read as “\( M \) steps to \( M' \)"

\[
\begin{align*}
M & \rightarrow M' & E\text{-APP1} \\
\text{app } M N & \rightarrow \text{app } M' N
\end{align*}
\]

\[
\begin{align*}
N & \rightarrow N' & \text{value } V & \rightarrow V' & E\text{-APP2} \\
\text{app } V N & \rightarrow \text{app } V N'
\end{align*}
\]

V value

app (lam x. M) V \rightarrow [V/x]M \quad E\text{-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

\[
\text{LF step: term } \rightarrow \text{term } \rightarrow \text{type } =
\]

\[
\begin{align*}
e\text{_app}_1 & : \text{step } M1 M1' \\
& \rightarrow \text{step } (\text{app } M1 M2) (\text{app } M1' M2)
\end{align*}
\]

\[
\begin{align*}
e\text{_app}_2 & : \text{step } M2 M2' \rightarrow \text{value } M1 \\
& \rightarrow \text{step } (\text{app } M1 M2) (\text{app } M1 M2')
\end{align*}
\]

\[
\begin{align*}
e\text{_app}_\text{abs} & : \text{value } M2 \\
& \rightarrow \text{step } (\text{app } (\text{lam } M) M2) (M M2)
\end{align*}
\]
Step 2b: Representation of Typing in LF

Typing Rules

\[ \frac{M : A \Rightarrow B \quad N : A}{\text{app } M \ N : B} \quad \text{T-APP} \]

\[ \frac{x : A \quad u}{\text{lam } x:A. M : A \Rightarrow B} \quad \text{T-ABS}^{x,u} \]
Step 2b: Representation of Typing in LF

Typing Rules

\[ \frac{M : A \Rightarrow B \quad N : A}{\text{app } M N : B} \quad \text{T-APP} \]

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

\[
\text{LF } \text{hastype: } \text{tm} \rightarrow \text{tp} \rightarrow \text{type} = \\
| \text{t_app: hashtype } M \ (\text{arr } A B) \rightarrow \text{hastype } N A \rightarrow \text{hastype } (\text{app } M N) B \\
| \text{t_abs: } (\forall x : \text{tm}. \text{hastype } x A \rightarrow \text{hastype } (M x) B) \\
\]

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

\[ \frac{x : \text{nat}}{D} \]

\[ (\text{lam } y : \text{nat}. y) : (\text{nat} \Rightarrow \text{nat}) \]

\[ (\text{lam } x : \text{nat}. \text{lam } y : \text{nat}. y) : (\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}) \]

\[ t_{\text{lam}}^{x,u} \]

is represented as \[ \vdash t_{\text{abs}} x. u. D \]
Step 2b: Representation of Typing in LF

Typing Rules

\[
\frac{M : A \Rightarrow B \quad N : A}{\text{app } M N : B} \quad \text{T-APP} \quad \frac{M : B}{\text{lam } x : A. M : A \Rightarrow B} \quad \text{T-ABS}^x,u
\]

**LF**

\[
\text{hastype: tm } \rightarrow \text{ tp } \rightarrow \text{ type } = \\
| \text{t_app: hashtype } M \ (\text{arr } A \ B) \quad | \text{t_abs: } (\prod x : \text{tm}. \text{hastype } x \ A \rightarrow \text{hastype } (M x) \ B) \\
\rightarrow \text{hastype } N \ A \quad \rightarrow \text{hastype } (\text{lam } A \ M) \ (\text{arr } A \ B); \\
\rightarrow \text{hastype } (\text{app } M N) \ B
\]

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

\[
\frac{x : \text{nat}}{D} \quad (\text{lam } y : \text{nat}. y) : (\text{nat } \Rightarrow \text{ nat}) \\
(t \lambda x : \text{nat}. \text{lam } y : \text{nat}. y) : (\text{nat } \Rightarrow \text{ nat } \Rightarrow \text{ nat}) \\
\text{t} \lambda \text{lam}^x,u \quad \text{is represented as } [\vdash \text{t_abs } x.u.D[x,u]]
\]
**Theorem**

If $D :: \vdash M : B$ and $S :: M \rightarrow N$ then $\vdash N : B$.

**Proof.**

By structural induction on the derivation $S :: M \rightarrow N$.

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

$D :: \vdash \text{app (lam } x: A.M) V : B$ \hspace{1cm} \text{by assumption}

$D_1 :: \vdash \text{lam } x: A.M : A \Rightarrow B$ \hspace{1cm} \text{and} $D_2 :: \vdash V : A$ \hspace{1cm} \text{by inversion using rule T-APP}

$D' :: x : A \vdash M : B$ \hspace{1cm} \text{by inversion on } D_1 \text{ using rule T-ABS}

:: \vdash [V/x]M : B$ \hspace{1cm} \text{by substitution lemma using } V \text{ and } D_2 \text{ in } D.
Beluga\(\mu\): Proofs as Programs

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

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

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Proofs as functions in Beluga</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theorem</td>
<td>Type</td>
</tr>
<tr>
<td>Case analysis</td>
<td>Case analysis and pattern matching</td>
</tr>
<tr>
<td>Inversion</td>
<td>Pattern matching using let-expression</td>
</tr>
<tr>
<td>Induction Hypothesis</td>
<td>Recursive call</td>
</tr>
</tbody>
</table>

Beluga programs manipulate directly derivation trees and contexts
Type preservation: Theorems as Types

**Theorem**

If \( D :: \vdash M : B \) and \( S :: M \rightarrow N \) then \( \vdash N : B \).

is translate to

**Computation-level Type in Beluga**

\[
\left[ \vdash \text{hastype } M \ B \right] \rightarrow \left[ \vdash \text{step } M \ N \right] \rightarrow \left[ \vdash \text{hastype } N \ B \right]
\]

**Remark:**

- \( \left[ \vdash \text{hastype } M \ B \right] \) is a contextual type. It stands for a closed typing derivation \( \text{hastype } M \ B \).
- \( M, N, \) and \( B \) are implicitly quantified at the outside. Beluga infers the type of these free variables.
Type preservation: Proofs are Programs

**Theorem:** If $D :: \vdash M : B$ and $S :: M \rightarrow N$ then $\vdash N : B$.

By structural induction on the derivation $S :: M \rightarrow N$.

**Case:** $S = \text{app (lam } x : A. M) V \rightarrow [V/x]M$

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

$D_1 :: \vdash \text{lam } x : A. M : A \Rightarrow B$ and $D_2 :: \vdash V : A$ by inversion using rule T-App

$D' :: x : A \vdash M : B$ by inversion on $D_1$ using rule T-Abs

$:: \vdash [V/x]M : B$ by substitution lemma using $V$ and $D_2$ in $D$.

**Computation in Beluga**

```
rec tps: [\vdash \text{hastype } M \ T] \rightarrow [\vdash \text{step } M \ N] \rightarrow [\vdash \text{hastype } N \ T] = /
/ \text{total s (tps m t n d s)}/
fn d \Rightarrow fn s \Rightarrow \text{case s of}
| [\vdash e\_app\_1 S1] \Rightarrow ?
| [\vdash e\_app\_2 S2 _] \Rightarrow ?
| [\vdash e\_app\_abs V] \Rightarrow 
  let [\vdash t\_app (t\_abs \lambda x.\lambda u. D') D2] = d in
  [\vdash D'[_ , D2] ]
```
Type preservation - Full Proof

```plaintext
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

\[
\begin{align*}
& \frac{M \rightarrow M'}{\text{app } M N \rightarrow \text{app } M' N} & \text{E-App1} \\
& \frac{N \rightarrow N'}{\text{app } V N \rightarrow \text{app } V N'} & \text{E-App2} \\
& \frac{V \text{ value}}{\text{app } (\text{lam } x : A.M) V \rightarrow [V/x]M} & \text{E-App-Abs}
\end{align*}
\]

Theorem

If \( S_1 :: M \rightarrow N_1 \) and \( M \rightarrow N_2 \) then \( N_1 = N_2 \).
Uniqueness of Evaluation

\[
\begin{align*}
M & \rightarrow M' & \text{E-App1} & \quad N & \rightarrow N' & \text{V value} & \quad \text{E-App2} \\
\text{app } M N & \rightarrow \text{app } M' N & & \text{app } V N & \rightarrow \text{app } V N' & & \\
\end{align*}
\]

\[
\frac{V \text{ value}}{\text{app } (\text{lam } x : A. M) V \rightarrow [V/x] M} \quad \text{E-App-ABS}
\]

**Theorem**

If \( S_1 :: M \rightarrow N_1 \) and \( M \rightarrow N_2 \) then \( N_1 = N_2 \).

By structural induction on \( S_1 :: M \rightarrow N_1 \).

**Case** \( S_1 = \)

\[
\frac{W \text{ value}}{\text{app } (\text{lam } x . M) W \rightarrow [W/x] M} \quad \text{E-App-ABS}
\]
Uniqueness of Evaluation

\[ M \xrightarrow{\text{app}} M' \quad \text{E-App1} \quad N \xrightarrow{\text{app}} N' \quad \text{app} \quad \text{app} \quad \text{app} \quad \text{app} \]

\[ \text{E-App2} \quad \text{E-App-Abs} \]

**Theorem**

If \( S_1 :: M \rightarrow N_1 \) and \( M \rightarrow N_2 \) then \( N_1 = N_2 \).

By structural induction on \( S_1 :: M \rightarrow N_1 \).

**Case** \( S_1 = \text{W value} \)

\[ \text{app} \ (\text{lam} \ x.M) \ W \rightarrow [W/x]M \quad \text{E-App-Abs} \]

**Sub-Case 1**: \( S_2 = \text{W value} \)

\[ \text{app} \ (\text{lam} \ x.M) \ W \rightarrow [W/x]M \quad \text{E-App-Abs} \]

\[ [W/x]M = [W/x]M \quad \text{by reflexivity} \]
Introduction

Uniqueness of Evaluation

\[
\frac{M \rightarrow M'}{\text{app } M N \rightarrow \text{app } M' N} \quad \text{E-App1}
\]

\[
\frac{N \rightarrow N'}{\text{app } V N \rightarrow \text{app } V N'} \quad \text{E-App2}
\]

\[
\frac{V \text{ value}}{\text{app } (\text{lam } x : A.M) V \rightarrow [V/x] M} \quad \text{E-App-Abs}
\]

Theorem

If \( S_1 :: M \rightarrow N_1 \) and \( M \rightarrow N_2 \) then \( N_1 = N_2 \).

By structural induction on \( S_1 :: M \rightarrow N_1 \).

**Case** \( S_1 = W \text{ value} \)

\[
\frac{W \text{ value}}{\text{app } (\text{lam } x . M) W \rightarrow [W/x] M} \quad \text{E-App-Abs}
\]

\[
\frac{\text{lam } x . M \rightarrow M'}{S} \quad \text{E-App1}
\]

**Sub-Case 2:** \( S_2 = \text{app } (\text{lam } x . M) N \rightarrow M' N \)

\[
\frac{\text{lam } x . M \text{ value}}{\text{app } (\text{lam } x . M) N \rightarrow M' N} \quad \text{by definition}
\]

\[
\perp \quad \text{since there is no derivation for } S \quad \text{(Lemma)}
\]
Introduction

Uniqueness of Evaluation

\[
\frac{M \rightarrow M'}{\text{app } M N \rightarrow \text{app } M' N} \quad \text{E-App1} \hspace{1cm} \frac{N \rightarrow N' \quad V \text{ value}}{\text{app } V N \rightarrow \text{app } V N'} \quad \text{E-App2}
\]

\[
\frac{V \text{ value}}{\text{app } (\text{lam } x:A.M) V \rightarrow [V/x]M} \quad \text{E-App-Abs}
\]

Theorem

If \( S_1 :: M \rightarrow N_1 \) and \( M \rightarrow N_2 \) then \( N_1 = N_2 \).

By structural induction on \( S_1 :: M \rightarrow N_1 \).

\textbf{Case} \( S_1 = \frac{W \text{ value}}{\text{app } (\text{lam } x.M) W \rightarrow [W/x]M} \quad \text{E-App-Abs} \)

\( W \rightarrow N' \quad (\text{lam } x.M) \text{ value} \)

\textbf{Sub-Case 3:} \( S_2 = \frac{\text{app } (\text{lam } x.M) W \rightarrow \text{app } (\text{lam } x.M) N'}{\text{E-App2}} \)

\( \perp \)

since \( W \) value there is no derivation for \( S \) (Lemma)

Lemma

\textit{If } M \rightarrow M' \text{ and } M \text{ value then } \perp.\)
### Step 1: Encoding Equality and Bottom

#### Equality (not built-in in Beluga)

**LF representation in Beluga**

```latex
\text{LF} \quad \text{equal} : \text{term} \rightarrow \text{term} \rightarrow \text{type} = \\
| \text{refl: equal M M};
```

Alternative: Define it structurally.

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

**LF representation in Beluga**

```latex
\text{not\_possible} : \text{type}.
```

Define an empty type with no constructors.
Step 2a: Encoding “Values don’t step”

Lemma

If $M \rightarrow M'$ and $M$ value then $\bot$.

```plaintext
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 $\bot$ in the on-paper proof
Step 2b: Encoding Uniqueness of Values

Theorem

If \( S_1 :: M \rightarrow N_1 \) and \( M \rightarrow N_2 \) then \( N_1 = N_2 \).

```
rec unique : \[ \vdash \text{step} M N_1 \] \rightarrow \[ \vdash \text{step} M N_2 \] \rightarrow \[ \vdash \text{equal} N_1 N_2 \] =
/ total s1 (unique m m1 m2 s1)/
fn s1 \Rightarrow fn s2 \Rightarrow case s1 of
| \[ \vdash e_{\text{app\_1}} S \] \Rightarrow ?
| \[ \vdash e_{\text{app\_2}} S V \] \Rightarrow ?
| \[ \vdash e_{\text{app\_abs}} V \] \Rightarrow case s2 of
  | \[ \vdash e_{\text{app\_abs}} _ \] \Rightarrow [ \vdash \text{refl}]
  | \[ \vdash e_{\text{app\_1}} S \] \Rightarrow \text{impossible} \ values_{\text{dont\_step}} [\vdash S] [\vdash v_{\text{lam}}]
  | \[ \vdash e_{\text{app\_2}} S _ \] \Rightarrow \text{impossible} \ values_{\text{dont\_step}} [\vdash S] [\vdash 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 trees
Type Uniqueness

Theorem

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

Induction on first typing derivation $D$.

Case 1

$D = D_1 \Gamma, x : A \vdash M : B$

$T\text{-Abs}_x \Gamma \vdash \text{lam } x : A. M : A \Rightarrow B$

$E_1 = E_1 \Gamma, x : A \vdash M : B$

$T\text{-Abs}_x \Gamma \vdash \text{lam } x : A. M : A \Rightarrow B'$

$E : \text{eq } B B'$ by i.h. using $D_1$ and $C_1$

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

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

Case 2

$D = x : A \in \Gamma \Gamma \vdash x : A$

$C = x : B \in \Gamma \Gamma \vdash x : B$

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

**Theorem**

If \( D : \Gamma \vdash M : A \) and \( C : \Gamma \vdash M : B \) then \( E : \text{eq } A B \).

Induction on first typing derivation \( D \).

**Case 1**

\[
D = \begin{array}{c}
\Gamma, x:A \vdash M : B \\
\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B
\end{array}
\]

\( T\text{-ABS}^x \)

\[
C = \begin{array}{c}
\Gamma, x:A \vdash M : B' \\
\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B'
\end{array}
\]

\( T\text{-ABS}^x \)
Type Uniqueness

Theorem

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

Induction on first typing derivation \( \mathcal{D} \).

Case 1

\[
\mathcal{D} = \frac{\mathcal{D}_1}{\Gamma, x:A \vdash M : B} \quad \text{T-ABS}^x
\]

\[
\mathcal{C} = \frac{\mathcal{C}_1}{\Gamma, x:A \vdash M : B'} \quad \text{T-ABS}^x
\]

\( \mathcal{E} : \text{eq } B B' \)

by i.h. using \( \mathcal{D}_1 \) and \( \mathcal{C}_1 \)
Type Uniqueness

Theorem

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

Induction on first typing derivation $D$.

**Case 1**

$$D = \Gamma, x : A \vdash M : B$$

$$\Gamma \vdash \text{lam } x : A. M : A \Rightarrow B$$

$$T-\text{ABS}^x$$

$$C = \Gamma, x : A \vdash M : B'$$

$$\Gamma \vdash \text{lam } x : A. M : A \Rightarrow B'$$

$$T-\text{ABS}^x$$

$E : \text{eq } B B'$

$C_1$ by i.h. using $D_1$ and $C_1$

$E : \text{eq } B B$ and $B = B'$

by inversion using reflexivity
Type Uniqueness

Theorem

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

Induction on first typing derivation $D$.

Case 1

$D = \Gamma, x : A \vdash M : B$

$C = \Gamma, x : A \vdash M : B'$

$E : \text{eq } B B'$

$E : \text{eq } B B'$ and $B = B'$

Therefore there is a proof for $\text{eq } (A \Rightarrow B) (A \Rightarrow B')$ by reflexivity.
Type Uniqueness

**Theorem**

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

**Induction on first typing derivation $D$.**

**Case 1**

\[
D_1 = \frac{\Gamma, x:A \vdash M : B}{\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B} \quad \text{T-ABS}^x
\]

\[
C_1 = \frac{\Gamma, x:A \vdash M : B'}{\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B'} \quad \text{T-ABS}^x
\]

$E : \text{eq } B B'$

$E : \text{eq } B B$ and $B = B'$

by i.h. using $D_1$ and $C_1$

by inversion using reflexivity

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

**Case 2**

\[
D = \frac{x:A \in \Gamma}{\Gamma \vdash x : A}
\]
Type Uniqueness

Theorem

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

Induction on first typing derivation \( \mathcal{D} \).

Case 1

\[
\begin{align*}
\mathcal{D} &= \frac{\mathcal{D}_1 \quad \mathcal{C}_1}{\Gamma \vdash \text{lam } x : A. M : A \Rightarrow B} \\
\mathcal{E} &= \text{eq } B B' & \text{by i.h. using } \mathcal{D}_1 \text{ and } \mathcal{C}_1 \\
\mathcal{E} &= \text{eq } B B \text{ and } B = B' & \text{by inversion using reflexivity}
\end{align*}
\]

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

Case 2

\[
\begin{align*}
\mathcal{D} &= \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
\mathcal{C} &= \frac{x : B \in \Gamma}{\Gamma \vdash x : B}
\end{align*}
\]
Introduction

Beluga: Design and implementation

Type Uniqueness

Theorem

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

Induction on first typing derivation \( \mathcal{D} \).

Case 1

\[
\mathcal{D}_1 \quad \Gamma, x:A \vdash M : B \\
\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B
\]

\( \mathcal{C}_1 \quad \Gamma, x:A \vdash M : B' \\
\Gamma \vdash \text{lam } x:A.M : A \Rightarrow B'
\]

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

\( \mathcal{E} : \text{eq } B B \) and \( B = B' \) by inversion using reflexivity

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

Case 2

\[
\mathcal{D} = x:A \in \Gamma \\
\Gamma \vdash x : A
\]

\( \mathcal{C} = 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

Theorem

If \( \text{D} : \Gamma \vdash M : A \) and \( C : \Gamma \vdash M : B \) then \( E : \text{eq} A B \).

is represented as

Computation-level Type in Beluga

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

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

\text{schema} \ \text{ctx} = \text{some} \ [t : \text{tp}] \ \text{block} \ x : \text{term}, u : \text{hastype} x T;

• \( M \) is a term that depends on \( \gamma \);
  • It has type \( [ \gamma \vdash \text{term} ] \)
• \( A \) and \( B \) are types that are closed;
  • They have type \( [ \vdash \text{tp} ] \)

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

\( \Rightarrow \) \( M \) is implicitly associated with the identity substitution

\( \Rightarrow \) \( A \) and \( B \) are associated with a weakening substitution

• \( [ \Psi ] M \) has type \( A [ \Psi ] \) and stands for a contextual object
  • \( M \) which has type \( A \) in the context \( \Psi \)

\[\text{NPP'08}\]
Step 2a: Theorem as Type

Theorem

If \( D : \Gamma \vdash M : A \) and \( C : \Gamma \vdash M : B \) then \( \varepsilon : \text{eq } A B \).
Step 2a: Theorem as Type

Theorem

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

is represented as

Computation-level Type in Beluga

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

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
Step 2a: Theorem as Type

Theorem

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

is represented as

Computation-level Type in Beluga

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

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  - schema $\text{ctx} = \text{some } [t : tp] \text{ block } x : \text{term}, u : \text{hastype } x \ T$;
- $M$ is a term that depends on $\gamma$; it has type $[\gamma \vdash \text{term}]$
- $A$ and $B$ are types that are closed; they have type $[\vdash \text{tp}]$

Recall: All meta-variables are associated with a substitution.
- $\rightsquigarrow M$ is implicitly associated with the identity substitution
- $\rightsquigarrow A$ and $B$ are associated with a weakening substitution
Step 2a: Theorem as Type

Theorem

If \( D : \Gamma \vdash M : A \) and \( C : \Gamma \vdash M : B \) then \( E : \text{eq} A B. \)

is represented as

Computation-level Type in Beluga

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

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  \[ \text{schema} \ \text{ctx} = \text{some} \ [t : \text{tp}] \ \text{block} \ x: \text{term}, u: \text{hastype} x T; \]
- \( M \) is a term that depends on \( \gamma \); it has type \([\gamma \vdash \text{term}]\)

\( A \) and \( B \) are types that are closed; they have type \([\vdash \text{tp}]\)

Recall: All meta-variables are associated with a substitution.
\( \leadsto M \) is implicitly associated with the identity substitution
\( \leadsto A \) and \( B \) are associated with a weakening substitution
Intrinsic Support for Contexts

```plaintext
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
  \( \text{b1: } \text{block} \ (x: \text{term}, u: \text{hastype} \ x \ \text{nat}), \text{b2: } \text{block} \ (y: \text{term}, v: \text{hastype} \ y \ \text{(arr nat nat)}) \)
Intrinsic Support for Contexts

```plaintext
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
  \[ b_1:\text{block} (x:\text{term}, u:\text{hastype} x \text{ nat}), b_2:\text{block} (y:\text{term}, v:\text{hastype} y (\text{arr} \text{ nat nat})) \]

- Well-formedness: \[ b_1:\text{block} \ x:\text{term}, u:\text{hastype} y \text{ nat} \] is ill-formed.
  \[ x:\text{term}, y:\text{term}, u:\text{hastype} x \text{ nat} \] 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
  $b_1 : \text{block} \ (x: \text{term}, u: \text{hastype} \ x \ \text{nat}), b_2 : \text{block} \ (y: \text{term}, v: \text{hastype} \ y \ (\text{arr} \ \text{nat} \ \text{nat}))$

- Well-formedness: $b_1 : \text{block} \ x : \text{term}, u : \text{hastype} \ y \ \text{nat}$ is ill-formed.
  $x : \text{term}, y : \text{term}, u : \text{hastype} \ x \ \text{nat}$ is ill-formed.

- Declarations are unique: $b_1$ is different from $b_2$
Intrinsic Support for Contexts

\[ \text{schema} \ \text{ctx} = \ \text{some} \ [t:tp] \ \text{block} \ x: \text{term}, \ u: \text{hastype} \ x \ t; \]

- The context \( x : \text{nat}, \ y : \text{nat} \rightarrow \text{nat} \) is represented as
  \[ \text{b1: block} (x: \text{term}, u: \text{hastype} \ x \ \text{nat}), \text{b2: block} (y: \text{term}, v: \text{hastype} \ y \ (\text{arr} \ \text{nat} \ \text{nat})) \]

- Well-formedness: \( \text{b1: block} \ x: \text{term}, u: \text{hastype} \ y \ \text{nat} \) is ill-formed.

- Declarations are unique: \( \text{b1} \) is different from \( \text{b2} \)
  \[ \text{b1.1} \] is different from \( \text{b2.1} \)
Intrinsic Support for Contexts

\[
\text{schema } \text{ctx} = \text{some } [t:tp] \text{ block } x: \text{term}, u: \text{hastype} x t;
\]

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

```plaintext
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
  
  \[ b1 : \text{block} (x : \text{term}, u : \text{hastype} x \text{ nat}), b2 : \text{block} (y : \text{term}, v : \text{hastype} y (\text{arr} \text{ nat nat})) \]

- Well-formedness: \( b1 : \text{block} \ x : \text{term}, u : \text{hastype} y \text{ nat} \) is ill-formed.
  
  \( x : \text{term}, y : \text{term}, u : \text{hastype} x \text{ nat} \) 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

\[\text{rec} \quad \text{unique} : (\gamma : \text{ctx}) \vdash \text{hastype} M A \rightarrow (\gamma \vdash \text{hastype} M B \rightarrow (\vdash \text{eq} A B) = \text{fn} d \Rightarrow \text{fn} c \Rightarrow \text{case} d \rightarrow (\gamma \vdash \text{t_app} D1 D2) \Rightarrow \% \text{Application Case} \quad \text{let} (\gamma \vdash \text{t_app} C1 C2) = c \quad \text{in} \quad \text{let} (\vdash \text{ref}) = \text{unique} (\gamma \vdash D1) (\gamma \vdash C1) \quad \text{in} \quad (\vdash \text{ref}) \quad (\gamma \vdash \text{t_lam} \lambda x. \lambda u. D) \Rightarrow \% \text{Abstraction Case} \quad \text{let} (\gamma \vdash \text{t_lam} \lambda x. \lambda u. C) = c \quad \text{in} \quad \text{let} (\vdash \text{ref}) = \text{unique} (\gamma, b : \text{block}(x : \text{term}, u : \text{hastype} x _)) (\gamma, b \vdash D [...], b.1, b.2) \) (\gamma, b \vdash C [...], b.1, b.2) \quad \text{in} \quad (\vdash \text{ref}) \quad (\gamma \vdash \#q.2) \Rightarrow \% d : \text{oft} \#q.1 T \% \text{Assumption Case} \quad \text{let} (\gamma \vdash \#r.2) = c \quad \text{in} \quad \% c : \text{oft} \#r.1 S \quad \text{Recalll:} \quad \#q: \text{block}(x : \text{term}, u : \text{hastype} x T) \quad \#r: \text{block}(x : \text{term}, u : \text{hastype} x S) \quad \text{We also know:} \quad \#r.1 = \#q.1 \quad \text{Therefore:} \quad T = S\]
Introduction

Type Uniqueness

\[
\text{rec } \text{unique} : (\gamma : \text{ctx})[\gamma \vdash \text{hastype } M A[]] \rightarrow [\gamma \vdash \text{hastype } M B[]] \rightarrow [\vdash \text{eq } A B] =
\]
Type Uniqueness

\[
\text{rec unique} : (\gamma : \text{ctx})[\gamma \vdash \text{hastype M A}] \rightarrow [\gamma \vdash \text{hastype M B}] \rightarrow [\vdash \text{eq A B}] = \\
\text{fn d } \Rightarrow \text{fn c } \Rightarrow \text{case d of}
\]

\[
\text{fn d } \Rightarrow \text{fn c } \Rightarrow \text{case d of}
\]

Recall:

\#q : \text{block x:term, u:hastype x T}
\#r : \text{block x:term, u:hastype x S}

We also know:

\#r.1 = \#q.1

Therefore:

T = S
Type Uniqueness

\[
\text{rec unique:} (\gamma : \text{ctx}) \left[ \gamma \vdash \text{hastype} M A[] \right] \rightarrow \left[ \gamma \vdash \text{hastype} M B[] \right] \rightarrow [\vdash \text{eq} A B] = \ \\
\text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of} \ \\
\mid \left[ \gamma \vdash \text{t_app} D_1 D_2 \right] \Rightarrow \ \\
\text{let } \left[ \gamma \vdash \text{t_app} C_1 C_2 \right] = c \text{ in} \ \\
\text{let } \left[ \vdash \text{ref} \right] = \text{unique } \left[ \gamma \vdash D_1 \right] \left[ \gamma \vdash C_1 \right] \text{ in} \ \\
\left[ \vdash \text{ref} \right]
\]
Type Uniqueness

\[
\text{rec unique:} (\gamma: \text{ctx}) [\gamma \vdash \text{hastype } M A[]} \rightarrow [\gamma \vdash \text{hastype } M B[]> \rightarrow [\vdash \text{eq } A B] =
\]

\[
\text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of}
\]

\[
| [\gamma \vdash \text{t_app } D1 D2] \Rightarrow \quad % \text{Application Case}
\]

\[
\text{let } [\gamma \vdash \text{t_app } C1 C2] = c \text{ in}
\]

\[
\text{let } [\vdash \text{ref}] = \text{unique } [\gamma \vdash D1] [\gamma \vdash C1] \text{ in}
\]

\[
[\vdash \text{ref}]
\]

\[
| [\gamma \vdash \text{t_lam } \lambda x. \lambda u. D] \Rightarrow \quad % \text{Abstraction Case}
\]

\[
\text{let } [\gamma \vdash \text{t_lam } \lambda x. \lambda u. C] = c \text{ in}
\]

\[
\text{let } [\vdash \text{ref}] = \text{unique } [\gamma, b:\text{block}(x:\text{term}, u:\text{hastype } x _) \vdash D[... , b.1, b.2]]
\]

\[
[\gamma, b \vdash C[... , b.1, b.2]] \text{ in}
\]

\[
[\vdash \text{ref}]
\]
Introduction

Beluga: Design and implementation

Type Uniqueness

```haskell
rec unique:(\gamma:ctx) [\gamma \vdash \text{hastype M A}] \rightarrow [\gamma \vdash \text{hastype M B}] \rightarrow [\vdash \text{eq A B}] =

\text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of }

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

| [\gamma \vdash \text{t_lam } \lambda x. \lambda u. D] \Rightarrow | \% Abstraction Case
| \quad \text{let } [\gamma \vdash \text{t_lam } \lambda x. \lambda u. C] = c \text{ in } |
| \quad \text{let } [\vdash \text{ref}] = \text{unique } [\gamma, b : \text{block}(x:term, u:hastype x _) \vdash D[... b.1, b.2]] |
| \quad [\gamma, b \vdash C[... b.1, b.2]] \text{ in } |
| \quad [\vdash \text{ref}] |

| [\gamma \vdash \#q.2] \Rightarrow | \% d : oft \#q.1 T \% Assumption Case
| \quad \text{let } [\gamma \vdash \#r.2] = c \text{ in } % c : oft \#r.1 S |
| \quad [\vdash \text{ref}] ;
```

Recall:
- \#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
```
Type Uniqueness

```plaintext
rec unique: (\gamma:ctx) [\gamma \vdash \text{hastype } M \ A[]] \rightarrow [\gamma \vdash \text{hastype } M \ B[]] \rightarrow [\vdash \text{eq } A \ B] =
fn \ d \Rightarrow fn \ c \Rightarrow \text{case } d \ of
  | [\gamma \vdash \text{t_app } D1 \ D2] \Rightarrow % Application Case
    \text{let } [\gamma \vdash \text{t_app } C1 \ C2] = c \ \text{in}
    \text{let } [\vdash \text{ref}] = \text{unique } [\gamma \vdash D1] \ [\gamma \vdash C1] \ \text{in}
    [\vdash \text{ref}]

  | [\gamma \vdash \text{t_lam } \lambda x. \lambda u. \ D] \Rightarrow % Abstraction Case
    \text{let } [\gamma \vdash \text{t_lam } \lambda x. \lambda u. \ C] = c \ \text{in}
    \text{let } [\vdash \text{ref}] = \text{unique } [\gamma, b:\text{block}(x: \text{term}, u: \text{hastype } x \ _) \vdash D[...\ b.1, b.2]]
    [\gamma, b \vdash C[...\ b.1, b.2]] \ \text{in}
    [\vdash \text{ref}]

  | [\gamma \vdash \#q.2] \Rightarrow % d : oft \ #q.1 T % Assumption Case
    \text{let } [\gamma \vdash \#r.2] = c \ \text{in} % c : oft \ #r.1 S
    [\vdash \text{ref}]

Recall:\n
#q:\text{block} \ x: \text{term}, u: \text{hastype } x \ T
#r:\text{block} \ x: \text{term}, u: \text{hastype } x \ S
```
Type Uniqueness

\[
\text{rec unique:}\ (\gamma:\text{ctx}) \to [\gamma \vdash \text{hastype } M ] A[] \to [\gamma \vdash \text{hastype } M ] B[] \to [\vdash \text{eq } A B] = \]

\[
\text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of}
\]

| [\gamma \vdash \text{t_app } D1 D2] \Rightarrow | \hspace{1cm} % Application Case
| let \ [\gamma \vdash \text{t_app } C1 C2] = c \text{ in}
| let \ [\vdash \text{ref}] = \text{unique } [\gamma \vdash D1] [\gamma \vdash C1] \text{ in}
| [\vdash \text{ref}]

| [\gamma \vdash \text{t_lam } \lambda x.\lambda u. D] \Rightarrow | \hspace{1cm} % Abstraction Case
| let \ [\gamma \vdash \text{t_lam } \lambda x.\lambda u. C] = c \text{ in}
| let \ [\vdash \text{ref}] = \text{unique } [\gamma, b: \text{block}(x:\text{term}, u: \text{hastype } x _) \vdash D[\ldots, b.1, b.2]]
| [\gamma, b \vdash C[\ldots, b.1, b.2]] \text{ in}
| [\vdash \text{ref}]

| [\gamma \vdash \#q.2] \Rightarrow | \hspace{1cm} % Assumption Case
| let \ [\gamma \vdash \#r.2] = c \text{ in} \hspace{1cm} % c : \text{oft } \#r.1 S
| [\vdash \text{ref};

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

\#q:\text{block } x:\text{term}, u: \text{hastype } x T
\#r:\text{block } x:\text{term}, u: \text{hastype } x S
Type Uniqueness

\[
\text{rec } \text{unique}: (\gamma: \text{ctx})(\gamma \vdash \text{hastype } M A[]) \rightarrow (\gamma \vdash \text{hastype } M B[]) \rightarrow (\vdash \text{eq } A B) = \\
\text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of} \\
| (\gamma \vdash \text{t_app } D1 D2) \Rightarrow & \quad \% \text{Application Case} \\
\text{let } (\gamma \vdash \text{t_app } C1 C2) = c \text{ in} \\
\text{let } (\vdash \text{ref}) = \text{unique } (\gamma \vdash D1) (\gamma \vdash C1) \text{ in} \\
(\vdash \text{ref}) \\
| (\gamma \vdash \text{t_lam } \lambda x. \lambda u. D) \Rightarrow & \quad \% \text{Abstraction Case} \\
\text{let } (\gamma \vdash \text{t_lam } \lambda x. \lambda u. C) = c \text{ in} \\
\text{let } (\vdash \text{ref}) = \text{unique } ((\gamma, b: \text{block}(x: \text{term}, u: \text{hastype } x \_)) \vdash D[\text{...}, b.1, b.2]) \\
((\gamma, b) \vdash C[\text{...}, b.1, b.2]) \text{ in} \\
(\vdash \text{ref}) \\
| (\gamma \vdash \text{#q.2}) \Rightarrow & \quad \% \text{d} : \text{oft } \text{#q.1 } T \quad \% \text{Assumption Case} \\
\text{let } (\gamma \vdash \text{#r.2}) = c \text{ in} \quad \% \text{c} : \text{oft } \text{#r.1 } S \\
(\vdash \text{ref}) ;
\]

Recall:
\#q: \text{block } x: \text{term}, u: \text{hastype } x \_ T
\#r: \text{block } x: \text{term}, u: \text{hastype } x \_ S

We also know: \text{#r.1 } = \text{#q.1}

Therefore: \text{T } = \text{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 `G`, an extension of first-order logic with a new quantifier `∇`, and develop inductive proofs in `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.
- **Simultaneous substitution and algebraic properties:** Substitution lemma, composition, decomposition, associativity, identity, etc.

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

a dozen such properties are needed
The Set-up: Simply Typed Lambda-Calculus - revisited

Types $A, B ::= i$

| $A \Rightarrow B$

Terms $M, N ::= x | c$

| lam $x.M$
| app $M N$

Evaluation Judgment: $M \rightarrow M'$

Call-by-Name (to simplify things)

- $\text{app} \ (\text{lam} \ x. M) \ N \rightarrow [N/x]M$
  - $\text{sbeta}$
- $M \rightarrow M$
  - $\text{srfl}$
- $M \rightarrow M'\quad M' \rightarrow N$
  - $\text{strans}$
The Set-up: Simply Typed Lambda-Calculus - revisited

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

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

Evaluation Judgment: $M \rightarrow M'$

- $\text{app } (\text{lam } x.M) N \rightarrow [N/x]M$
  - $\text{s}_{\text{beta}}$

- $M \rightarrow M'$
  - $\text{app } M N \rightarrow \text{app } M' N$
  - $\text{s}_{\text{app}}$

- $M \rightarrow M' \quad M' \rightarrow N$
  - $\text{s}_{\text{trans}}$

Call-by-Name (to simplify things)

Typing Judgment: $M : A$

- $x : A^u$
- $\vdots$
- $M : B$
- $\text{lam } x.M : A \Rightarrow B$
- $\text{lam}^{x,u}$
- $M : A \Rightarrow B$
- $N : A$
- $\text{app } M N : B$

read as "$M$ has type $A$" (Gentzen-style)
Weak Normalization for Simply Typed Lambda-calculus

Theorem

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

Proof.

1. Define reducibility candidate $R_A = \{M | M$ halts $\}$

2. $R_A \Rightarrow B = \{M | M$ halts and $\forall N \in R_A, (\text{app } M N) \in R_B \}$

3. If $M \in R_A$ then $M$ halts.

4. Backwards closed: If $M' \in R_A$ and $M \rightarrow M'$ then $M \in R_A$.

4. Fundamental Lemma: If $\vdash M : A$ then $M \in R_A$. (Requires a generalization)
Theorem

If \( \vdash M : A \) then \( M \) halts, i.e. there exists a value \( V \) s.t. \( M \rightarrow^* V \).
Weak Normalization for Simply Typed Lambda-calculus

**Theorem**

If ⊢ M : A then M halts, i.e. there exists a value V s.t. M →* V.

**Proof.**

1. Define reducibility candidate $R_A$
   
   \[ R_i = \{ M | M \text{ halts} \} \]
   \[ R_{A\rightarrow B} = \{ M | M \text{ halts and } \forall N \in R_A, (\text{app } M \ N) \in R_B \} \]

2. If $M \in R_A$ then $M$ halts.

3. Backwards closed: If $M' \in R_A$ and $M \rightarrow M'$ then $M \in R_A$.

4. **Fundamental Lemma:** If ⊢ M : A then $M \in R_A$. (Requires a generalization)
**GENERALIZATION OF FUNDAMENTAL LEMMA**

**Lemma (Main Lemma)**

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

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

\[
\begin{align*}
\sigma \in \mathcal{R}_\Gamma & \quad N \in \mathcal{R}_A \\
(\sigma, N/x) \in \mathcal{R}_{\Gamma,x:A}
\end{align*}
\]
Generalization of Fundamental Lemma

Lemma (Main lemma)

If $D : \Gamma \vdash M : A$ and $\sigma \in R_\Gamma$ then $[\sigma]M \in R_A$.

Proof.

Case $D = \frac{D_1}{\Gamma, x:A \vdash M : B}$

$\Gamma \vdash \text{lam } x.M : A \Rightarrow B$  

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

by properties of substitution

halts $\text{lam } x.[\sigma, x/x]M$  

since it is a value

Suppose $N \in R_A$.

$[\sigma, N/x]M \in R_B$  

by l.H. on $D_1$ since $\sigma \in R_\Gamma$

$[N/x][\sigma, x/x]M \in R_B$  

by properties of substitution

app $\text{lam } x. [\sigma, x/x]M \ N \in R_B$  

by Backwards closure

Hence $[\sigma](\text{lam } x.M) \in R_{A \Rightarrow B}$  

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

Types $A, B ::= i$
- $A \Rightarrow B$

Terms $M, N ::= x | c$
- $\lambda x.M$
- $app M N$

Typing rules

<table>
<thead>
<tr>
<th>$x : A$</th>
<th>$u$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$M : B$</td>
<td>$\lambda x.M : A \Rightarrow B$</td>
</tr>
</tbody>
</table>

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 $A, B ::= i$

$\mid A \Rightarrow B$

Terms $M, N ::= x \mid c$

$\mid \text{lam} \, x. \, M$

$\mid \text{app} \, M \, N$

Typing rules

$\frac{x : A \quad u}{\_}$

$\frac{\_}{\_}$

$\frac{M : B}{\_}$

$\frac{\text{const}}{c : i}$

$\frac{\_}{\_}$

$\frac{\_}{\_}$

$\frac{\_}{\_}$

Intrinsically typed Term Representation

LF representation in Beluga

\begin{verbatim}
LF tp: type =
| i: tp
| arr: tp \rightarrow tp \rightarrow tp;

LF tm: tp \rightarrow type =
| c : tm i
| lam: (tm A \rightarrow tm B) \rightarrow tm (arr A B)
| app: tm (arr A B) \rightarrow tm A \rightarrow tm B;
\end{verbatim}
Step 1a: Represent Semantics in LF
Step 1b: Reducibility Candidates as Stratified Types

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

\[
\begin{align*}
\mathcal{R}_i &= \{ M \mid \text{halts } M \} \\
\mathcal{R}_{A \Rightarrow B} &= \{ M \mid \text{halts } M \text{ and } \forall N \in \mathcal{R}_A, (\text{app } M N) \in \mathcal{R}_B \}
\end{align*}
\]
Step 1b: Reducibility Candidates as Stratified Types

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

$$
\mathcal{R}_i = \{ M \mid \text{halts } M \}
$$

$$
\mathcal{R}_{A \Rightarrow B} = \{ M \mid \text{halts } M \text{ and } \forall N \in \mathcal{R}_A, (\text{app } M N) \in \mathcal{R}_B \}
$$

Computation-level data types in Beluga

```plaintext
stratified  Reduce : \{A:\[\vdash\text{tp}]\} \{M:\[\vdash\text{tm }A]\} \text{ type = }
| I  : [ \vdash \text{halts } M] \rightarrow \text{Reduce } [ \vdash i] [\vdash M]
| Arr : [ \vdash \text{halts } M] \rightarrow

(\{N:\[\vdash \text{tm } A]\} \text{ Reduce } [ \vdash A] [\vdash N] \rightarrow \text{Reduce } [ \vdash B] [\vdash \text{app } M N])

\rightarrow \text{Reduce } [ \vdash \text{arr } A B] [\vdash M];
```

- $[\vdash \text{app } M N]$ and $[\vdash \text{arr } A B]$ 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 $[\vdash ]$)
  - $\rightarrow$ is a computation-level function (used outside $[\vdash ]$)
- Not strictly positive definition, but stratified.
Step 1b: Reducibility Candidates as Inductive Types

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

- \( \cdot \in \mathcal{R} \)
- \( \sigma \in \mathcal{R}_\Gamma \)
- \( N \in \mathcal{R}_A \)
- \( (\sigma, N/x) \in \mathcal{R}_{\Gamma,x:A} \)
Step 1b: Reducibility Candidates as Inductive Types

Reducibility candidates for substitutions $\sigma \in R_\Gamma$:

\[
\frac{\cdot \in R.}{\sigma \in R_\Gamma \quad N \in R_A} \quad \frac{\cdot \in R.}{(\sigma, N/x) \in R_{\Gamma, x:A}}
\]

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
  
  \[\text{schema} \quad \text{ctx} = x:tm A.\]

- Substitution $\tau$ are first-class and have type $\Psi \vdash \Phi$ providing a mapping from $\Phi$ to $\Psi$. 
Step 2: Theorems as Types

Lemma (Backward closed)

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

rec closed : \[\vdash \text{mstep } M M'] \rightarrow \text{Reduce } [\vdash A] [\vdash M'] \rightarrow \text{Reduce } [\vdash A] [\vdash M] = ? ;

Lemma (Main lemma)

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

rec main: \{\Gamma:ctx\}{M:[\Gamma \vdash \text{tm } A]} \text{RedSub } [\vdash \sigma] \rightarrow \text{Reduce } [\vdash A] [\vdash M[\sigma] ] = ? ;
## Step 2: Theorems as Types

**Lemma (Backward closed)**

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

```plaintext
rec closed : \[ \Gamma \vdash \text{mstep } M \rightarrow M' \] → Reduce \[ \Gamma \vdash A \] \[ \Gamma \vdash M' \] → Reduce \[ \Gamma \vdash A \] \[ \Gamma \vdash M \] = ? ;
```

**Lemma (Main lemma)**

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

```plaintext
rec main: \{\Gamma : \text{ctx}\} \{M : [\Gamma \vdash \text{tm } A[]]\} \text{RedSub } [\Gamma \vdash \sigma] \rightarrow \text{Reduce } [\Gamma \vdash A] [\Gamma \vdash M[\sigma]] = ? ;
```
Step 2: Fundamental Lemma
Step 2: Fundamental Lemma

\texttt{rec closed : [ ⊢ \textit{mstep} M M'] → \textit{Reduce} [ ⊢ A] [ ⊢ M'] → \textit{Reduce} [ ⊢ A] [ ⊢ \textit{M}]} = ? ;

\texttt{rec main : \{Γ:ctx\}\{M:[Γ ⊢ \textit{tm} A[\]]\} \textit{RedSub} [ ⊢ \sigma] → \textit{Reduce} [ ⊢ A] [ ⊢ \textit{M}[\sigma]] =}
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] ⇒
            let Arr ha f = main [Γ] [Γ ⊢ M1] rs in
            f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2] rs)
```

• 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
Step 2: Fundamental Lemma

\[
\text{rec closed : } [\vdash \text{mstep M M'}] \rightarrow \text{Reduce } [\vdash A] [\vdash \text{M'}] \rightarrow \text{Reduce } [\vdash A] [\vdash M] = \?
; \]

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

\[
\text{mlam } \Gamma \Rightarrow \text{mlam } M \Rightarrow \text{fn } rs \Rightarrow \text{case } [\Gamma \vdash M] \text{ of}
\]

| [\Gamma \vdash \#p] \Rightarrow \text{lookup } [\Gamma] [\Gamma \vdash \#p] rs & \% \text{ Variable} \\
| [\Gamma \vdash \text{app M1 M2}] \Rightarrow & \% \text{ Application} \\
| \text{let Arr ha f = main } [\Gamma] [\Gamma \vdash M1] rs \text{ in} & \\
| f [\vdash _] (\text{main } [\Gamma] [\Gamma \vdash M2] rs) & \\
| [\Gamma \vdash \text{lam } \lambda x. M1] \Rightarrow & \% \text{ Abstraction} \\
| \text{Arr } [\vdash \text{h_value s_refl v_lam}] & \\
| (\text{mlam } N \Rightarrow \text{fn } rN \Rightarrow \text{closed } [\vdash s_{\text{beta}}] & \\
| (\text{main } [\Gamma,x:\text{tm } _] [\Gamma,x \vdash M1] (\text{Cons rs rN})) & \\
|}
Step 2: Fundamental Lemma

\texttt{rec} \quad \textit{closed} : \begin{array}{l}
\quad \vdash \text{mstep} \; M \; M' \Rightarrow \text{Reduce} \; \vdash A \; \vdash M' \Rightarrow \text{Reduce} \; \vdash A \; \vdash M = ? ;
\end{array}

\texttt{rec} \quad \textit{main} : \begin{array}{l}
\quad \{\Gamma : \text{ctx}\} \{M : [\Gamma \vdash \text{tm} A[]]\} \; \text{RedSub} \; \vdash \sigma \Rightarrow \text{Reduce} \; \vdash A \; \vdash M'[\sigma] = \text{mlam} \; \Gamma \Rightarrow \text{mlam} \; M \Rightarrow \text{fn} \; rs \Rightarrow \text{case} \; [\Gamma \vdash M] \; \text{of}
\end{array}

\begin{array}{l}
\mid [\Gamma \vdash \#p] \Rightarrow \text{lookup} \; [\Gamma] \; [\Gamma \vdash \#p] \; rs \quad \% \; \text{Variable} \\
\mid [\Gamma \vdash \text{app} \; M_1 \; M_2] \Rightarrow \quad \% \; \text{Application} \\
\qquad \text{let} \quad \text{Arr} \; \text{ha} \; f = \text{main} \; [\Gamma] \; [\Gamma \vdash M_1] \; rs \; \text{in} \\
\qquad \quad f \; [\vdash \_ \_] \; (\text{main} \; [\Gamma] \; [\Gamma \vdash M_2] \; rs) \\
\mid [\Gamma \vdash \text{lam} \; \lambda x. \; M_1] \Rightarrow \quad \% \; \text{Abstraction} \\
\qquad \text{Arr} \; [\vdash \text{h_value} \; \text{s_refl} \; \text{v_lam}] \\
\qquad \quad (\text{mlam} \; N \Rightarrow \text{fn} \; rN \Rightarrow \text{closed} \; [\vdash \text{s_beta}] \\
\qquad \qquad \quad (\text{main} \; [\Gamma, x : \text{tm} \_ \_] \; [\Gamma, x \vdash M_1] \; (\text{Cons} \; rs \; rN)))) \\
\mid [\Gamma \vdash c] \Rightarrow I \; [\vdash \text{h_value} \; \text{s_refl} \; \text{v_c}] ; \quad \% \; \text{Constant}
\end{array}
Step 2: Fundamental Lemma

\[ \text{rec} \text{ closed : } [\vdash \text{mstep } M M'] \rightarrow \text{Reduce } [\vdash A] [\vdash M'] \rightarrow \text{Reduce } [\vdash A] [\vdash M] = \ ? ; \]

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

\[ \text{mlam } \Gamma \Rightarrow \text{mlam } M \Rightarrow \text{fn } rs \Rightarrow \text{case } [\Gamma \vdash M] \text{ of} \]

\[ | [\Gamma \vdash \#p] \Rightarrow \text{lookup } [\Gamma] [\Gamma \vdash \#p] rs \% \text{ Variable} \]

\[ | [\Gamma \vdash \text{app } M1 M2] \Rightarrow \]

\[ \text{let } \text{Arr } ha f = \text{main } [\Gamma] [\Gamma \vdash M1] \text{ rs in} \]

\[ f [\vdash _] (\text{main } [\Gamma] [\Gamma \vdash M2] \text{ rs}) \% \text{ Application} \]

\[ | [\Gamma \vdash \text{lam } \lambda x. M1] \Rightarrow \]

\[ \text{Arr } [\vdash \text{h_value s_refl v_lam}] \]

\[ (\text{mlam } N \Rightarrow \text{fn } rN \Rightarrow \text{closed } [\vdash \text{s_beta}] \]

\[ (\text{main } [\Gamma,x:tm \_] [\Gamma,x \vdash M1] \text{ (Cons rs rN)}) \% \text{ Abstraction} \]

\[ | [\Gamma \vdash c] \Rightarrow \text{I } [\vdash \text{h_value s_refl v_c}] ; \% \text{ Constant} \]

- 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]
“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
  - Inversion
  - Induction hypothesis

- **Bottom**: Contextual LF
  - On paper proof
  - In Beluga [IJCAR’10, CADE’15]

<table>
<thead>
<tr>
<th>Well-formed derivations</th>
<th>Dependent types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Renaming, Substitution</td>
<td>$\alpha$-renaming, $\beta$-reduction in LF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Well-scoped derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
</tr>
<tr>
<td>Properties of contexts (weakening, uniqueness)</td>
</tr>
<tr>
<td>Substitutions (composition, identity)</td>
</tr>
</tbody>
</table>

| Contextual types and objects [TOCL’08] |
| Context schemas |
| Typing for schemas |
| 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 $G$, an
  Extension of first-order logic with a new quantifier $\nabla$, and develop inductive
  Proofs in $G$ 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])

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