Beluga\(\mu\): Programming proofs in context ...

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada
Motivation

How to program and reason with formal systems and proofs?
How to program and reason with formal systems and proofs?

- Formal systems (given via axioms and inference rules) play an important role when designing and implementing software.
Motivation

How to program and reason with formal systems and proofs?

- Formal systems (given via axioms and inference rules) play an important role when designing and implementing software.
- Proofs (that a given property is satisfied) are an integral part of the software.
Motivation

How to program and reason with formal systems and proofs?

- Formal systems (given via axioms and inference rules) play an important role when designing and implementing software.

- Proofs (that a given property is satisfied) are an integral part of the software.

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

Design and implementation of Beluga

• Introduction
• Example: Simply typed lambda calculus
• Writing a proof in Beluga . . .
• Wanting more: . . .
  • Evaluation using closures
  • Normalization
• Conclusion

“The limits of my language mean the limits of my world.”
- L. Wittgenstein
This talk

Design and implementation of Beluga

- Introduction
- Example: Simply typed lambda calculus
- Writing a proof in Beluga . . .
- Wanting more: . . .
  - Evaluation using closures
  - Normalization
- Conclusion

“The limits of my language mean the limits of my world.”
- L. Wittgenstein
Simply typed lambda-calculus

Types and Terms

Types $T ::= \text{nat} \mid \text{arr } T_1 \ T_2$

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

Typing Judgment: $\odot M T$ read as "$M$ has type $T$"

Typing rules (Gentzen-style, context-free)

\begin{align*}
\text{oft } & x \ T \ u \\
\text{oft } & M \ S \\
\text{oft } & (\text{lam } x : T . M) (\text{arr } T_1 \ T_2) \ t \\
\text{lam } & x , \ u \ \text{oft } M \ (\text{arr } T_1 \ T_2) \ \text{oft } N \ T \\
\text{oft } & (\text{app } M \ N) \ S \ t
\end{align*}
Simply typed lambda-calculus

Types and Terms

Types $T ::= \text{nat} \mid \text{arr} T_1 T_2$

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

Typing Judgment: $\text{oft} \ M \ T$ read as “$M$ has type $T$”
Simply typed lambda-calculus

Types and Terms

Types \( T \) ::= nat \hspace{1cm} \text{Terms} \ M ::= x \hspace{1cm} \begin{array}{l}
| \text{arr } T_1 \ T_2 \\
| \text{lam } x : T . M \\
| \text{app } M \ N
\end{array}

Typing Judgment: \( \text{oft } M \ T \) read as “\( M \) has type \( T \)”

Typing rules (Gentzen-style, context-free)

\[
\begin{array}{c}
\frac{\text{oft } x \ T \ u}{\text{oft } x \ T} \\
\vdots \\
\frac{\text{oft } M \ S}{\text{oft } (\text{lam } x : T . M) \ (\text{arr } T \ S) \ t_{\text{lam}^x,u}}
\end{array}
\]
Simply typed lambda-calculus

Types and Terms

Types $T ::= \text{nat} \\
| \text{arr } T_1 \ T_2$

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

Typing Judgment: oft $M \ T$ read as “$M$ has type $T$”

Typing rules (Gentzen-style, context-free)

- $\frac{\text{oft } x \ T \ u}{\text{oft } x \ T}$
- $\frac{\text{oft } M \ S}{\text{oft } (\text{lam } x : T . M) (\text{arr } T \ S)}$ t$_{\text{lam}}^{x,u}$
- $\frac{\text{oft } M (\text{arr } T \ S) \ \text{oft } N \ T}{\text{oft } (\text{app } M \ N) \ S}$ t$_{\text{app}}$
Simply typed lambda-calculus

Types and Terms

Types $T ::= \text{nat}$

Terms $M ::= x$

$| \text{arr } T_1 T_2$

$| \text{lam } x: T . M$

$| \text{app } M N$

Typing Judgment: $\text{oft } M T$ read as “$M$ has type $T$”

Typing rules (Gentzen-style, context-free)

\[
\begin{align*}
\text{oft } x & \rightarrow u \\
\text{oft } M S & \rightarrow t_{\text{lam}}^{x,u} \\
\text{oft } (\text{lam } x: T . M) (\text{arr } T S) & \rightarrow t_{\text{app}} \\
\text{oft } M (\text{arr } T S) & \rightarrow \text{oft } N T \\
\text{oft } (\text{app } M N) S & \rightarrow t_{\text{app}}
\end{align*}
\]

Context $\Gamma ::= \cdot | \Gamma, x, \text{oft } x T$

We are introducing the variable $x$ together with the assumption $\text{oft } x T$
Simply typed lambda-calculus

Types and Terms

Types $T ::= \text{nat} \mid T_1 \rightarrow T_2$

Terms $M ::= x \mid T_1 \rightarrow T_2 \mid \text{lam } x : T.M \mid \text{app } M N$

Typing Judgment: $\Gamma \vdash \text{oft } M T$ read as “$M$ has type $T$ in context $\Gamma$”

Typing rules

\[
\frac{x, u : \text{oft } x T \in \Gamma}{\Gamma \vdash \text{oft } x T} \quad u
\]

\[
\frac{\Gamma, x, u : \text{oft } x T \vdash \text{oft } M S}{\Gamma \vdash \text{oft } (\text{lam } x : T.M) (\text{arr } T S)} \quad \text{t_lam}^{x,u}
\]

\[
\frac{\Gamma \vdash \text{oft } M (\text{arr } T S) \quad \Gamma \vdash \text{oft } N T}{\Gamma \vdash \text{oft } (\text{app } M N) S} \quad \text{t_app}
\]

Context $\Gamma ::= \cdot \mid \Gamma, x, \text{oft } x T$ We are introducing the variable $x$ together with the assumption $\text{oft } x T$
Talking about derivations

Typing rules

\[
\frac{x, u : \text{oft} \times T \in \Gamma}{\Gamma \vdash \text{oft} \times T} \quad \text{u} \\
\frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S}{\Gamma \vdash \text{oft} (\text{lam} x : T . M) (\text{arr} T S)} \quad \text{t}_{\text{lam}}^{x,u} \\
\frac{\Gamma \vdash \text{oft} M \text{ (arr} T S) \quad \Gamma \vdash \text{oft} N T}{\Gamma \vdash \text{oft} \text{ (app} M N) S} \quad \text{t}_{\text{app}}
\]
Talking about derivations

Typing rules

\[ \Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S \quad \text{t}_{\text{lam}^{x,u}} \]

\[ \Gamma \vdash \text{oft} (\text{app} M N) S \quad \text{t}_{\text{app}} \]

• What kinds of variables are used?
Talking about derivations

Typing rules

\[ \frac{x, u : \text{oft} \times T \in \Gamma}{\Gamma \vdash \text{oft} \times T} \quad u \]

\[ \frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S}{\Gamma \vdash \text{oft} \left( \text{lam} x: T . M \right) \left( \text{arr} T S \right)} \quad t_{\text{lam}}^{x, u} \]

\[ \frac{\Gamma \vdash \text{oft} M \left( \text{arr} T S \right)}{\Gamma \vdash \text{oft} N T} \quad \frac{\Gamma \vdash \text{oft} N T}{\Gamma \vdash \text{oft} \left( \text{app} M N \right) S} \quad t_{\text{app}} \]

- What kinds of variables are used? **Bound variables, Schematic variables**
  in particular: **Meta-variables, Parameter variables, Context variables**
Talking about derivations

Typing rules

\[ \frac{x, u : \text{oft } x T \in \Gamma}{\Gamma \vdash \text{oft } x T} \]
\[ x, u : \text{oft } x T \vdash \text{oft } M S \quad \Gamma \vdash \text{oft } N T \]
\[ \Gamma, x, u : \text{oft } x T \vdash \text{oft } (\text{lam } \lambda x : T. M \; \text{arr } T S) \quad \Gamma \vdash \text{oft } (\text{app } M N) S \]

- What kinds of variables are used? **Bound variables, Schematic variables** in particular: Meta-variables, Parameter variables, Context variables
- What operations on variables are needed?
Talking about derivations

Typing rules

\[
\frac{x, u : \text{oft} \times T \in \Gamma}{\Gamma \vdash \text{oft} \times T \quad u}
\]

\[
\frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S}{\Gamma \vdash \text{oft} (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S) \quad \text{t_lam}^{x,u}}
\]

\[
\frac{\Gamma \vdash \text{oft} M (\text{arr} \ T \ S) \quad \Gamma \vdash \text{oft} N T}{\Gamma \vdash \text{oft} (\text{app} \ M \ N) \ S \quad \text{t_app}}
\]

- What kinds of variables are used? **Bound variables, Schematic variables**
  in particular: **Meta-variables, Parameter variables, Context variables**

- What operations on variables are needed? **Substitution for bound variable,**
  **Renaming of bound variables, Substitution for schematic variables**
Talking about derivations

Typing rules

\[
\frac{x, u : \text{oft} \times T \in \Gamma}{\Gamma \vdash \text{oft} \times T}
\]

\[
\frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S}{\Gamma \vdash \text{oft} (\text{lam} x : T . M) (\text{arr} T S)}
\]

\[
\frac{\Gamma \vdash \text{oft} M (\text{arr} T S) \quad \Gamma \vdash \text{oft} N T}{\Gamma \vdash \text{oft} (\text{app} M N) S}
\]

- What kinds of variables are used? **Bound variables, Schematic variables**
  - in particular: **Meta-variables, Parameter variables, Context variables**

- What operations on variables are needed? **Substitution for bound variable, Renaming of bound variables, Substitution for schematic variables**

- How should we represent contexts? **What properties do contexts have?**
Talking about derivations

Typing rules

\[ \frac{x, u : \text{oft} \times T \in \Gamma}{\Gamma \vdash \text{oft} \times T} u \]

\[ \frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S \quad \text{t}_{\text{lam}}^{x,u}}{\Gamma \vdash \text{oft} (\text{lam } x : T . M) (\text{arr } T S)} \]

\[ \frac{\Gamma \vdash \text{oft} M (\text{arr } T S) \quad \Gamma \vdash \text{oft} N T}{\Gamma \vdash \text{oft} (\text{app } M N) S} \quad \text{t}_{\text{app}} \]

- What kinds of variables are used? **Bound variables**, **Schematic variables** in particular: Meta-variables, Parameter variables, Context variables

- What operations on variables are needed? **Substitution for bound variable**, **Renaming of bound variables**, **Substitution for schematic variables**

- How should we represent contexts? What properties do contexts have? **(Structured) Sequences**, **Every declaration is unique**, **weakening**, **substitution lemma**, etc.
Talking about derivations

Typing rules

\[
\frac{x, u : \text{oft} \times T \in \Gamma}{\Gamma \vdash \text{oft} \times T \quad u}
\]

\[
\frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M S}{\Gamma \vdash \text{oft} (\text{lam} x : T.M) (\text{arr} T S) \quad t_{\text{lam}}^{x, u}}
\]

\[
\frac{\Gamma \vdash \text{oft} M (\text{arr} T S) \quad \Gamma \vdash \text{oft} N T}{\Gamma \vdash \text{oft} (\text{app} M N) S \quad t_{\text{app}}}
\]

- What kinds of variables are used? **Bound variables**, **Schematic variables** in particular: **Meta-variables**, **Parameter variables**, **Context variables**

- What operations on variables are needed? **Substitution for bound variable**, **Renaming of bound variables**, **Substitution for schematic variables**

- How should we represent contexts? What properties do contexts have? **(Structured) Sequences**, **Every declaration is unique**, **weakening**, **substitution lemma**, etc.

Any mechanization of proofs must deal with these issues; it is just a matter how much support one gets in a given meta-language.
Type uniqueness

Theorem

If \( D : \Gamma \vdash \text{oft} M T \) and \( C : \Gamma \vdash \text{oft} M S \) then \( \mathcal{E} : \text{eq} T S \).
**Type uniqueness**

**Theorem**

If \( D : \Gamma \vdash \text{oft} \ M \ T \) and \( C : \Gamma \vdash \text{oft} \ M \ S \) then \( E : \text{eq} \ T \ S \).

**Induction on first typing derivation \( D \).**

**Case 1**

\[
D_1 = \frac{\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S}{\Gamma \vdash \text{oft} (\text{lam} \ x : T.M) (\text{arr} \ T \ S)}
\]

\[
C_1 = \frac{\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S'}{\Gamma \vdash \text{oft} (\text{lam} \ x : T.M) (\text{arr} \ T \ S')}
\]
Type uniqueness

**Theorem**

If $\mathcal{D} : \Gamma \vdash \text{oft } M \ T$ and $\mathcal{C} : \Gamma \vdash \text{oft } M \ S$ then $\mathcal{E} : \text{eq } T \ S$.

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

**Case 1**

\[
\begin{align*}
\mathcal{D} &= \frac{\Gamma, x, u : \text{oft } x \ T \vdash \text{oft } M \ S}{\Gamma \vdash \text{oft } (\text{lam } x : T.M) \ (\text{arr } T \ S)} \quad \text{t\_lam} & \quad \mathcal{C} &= \frac{\Gamma, x, u : \text{oft } x \ T \vdash \text{oft } M \ S'}{\Gamma \vdash \text{oft } (\text{lam } x : T.M) \ (\text{arr } T \ S')} \quad \text{t\_lam} \\
\mathcal{E} &= \text{eq } S \ S' & \quad \text{by i.h. using } \mathcal{D}_1 \text{ and } \mathcal{C}_1
\end{align*}
\]
Type uniqueness

Theorem

If $\mathcal{D} : \Gamma \vdash \text{oft } M \ T$ and $\mathcal{C} : \Gamma \vdash \text{oft } M \ S$ then $\mathcal{E} : \text{eq } T \ S$.

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

Case 1

\[
\mathcal{D} = \frac{\mathcal{D}_1}{\Gamma \vdash \text{oft } \text{lam } (\text{lam } x: T. M) (\text{arr } T \ S)}
\]

$\mathcal{E} : \text{eq } S \ S'$

$\mathcal{E} : \text{eq } S \ S$ and $S = S'$

$\mathcal{C} = \frac{\mathcal{C}_1}{\Gamma \vdash \text{oft } \text{lam } (\text{lam } x: T. M) (\text{arr } T \ S')}$

by i.h. using $\mathcal{D}_1$ and $\mathcal{C}_1$

by inversion using reflexivity
### Theorem

If \( \mathcal{D} : \Gamma \vdash \text{oft} \ M \ T \) and \( \mathcal{C} : \Gamma \vdash \text{oft} \ M \ S \) then \( \mathcal{E} : \text{eq} \ T \ S \).

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

**Case 1**

\[
\mathcal{D} = \frac{\mathcal{D}_1}{t\_\text{lam}} \quad \mathcal{C} = \frac{\mathcal{C}_1}{t\_\text{lam}}
\]

\[
\mathcal{D}_1 = \frac{\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S}{t\_\text{lam}} \quad \mathcal{C}_1 = \frac{\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S'}{t\_\text{lam}}
\]

\[
\begin{align*}
\mathcal{E} : \text{eq} & \ S \ S' \\
\mathcal{E} : \text{eq} & \ S \ S \quad \text{and} \quad S = S'
\end{align*}
\]

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

by inversion using reflexivity

Therefore there is a proof for \( \text{eq} \ (\text{arr} \ T \ S) \ (\text{arr} \ T \ S') \) by reflexivity.
Type uniqueness

Theorem

If $D : \Gamma \vdash \text{oft } M \; T$ and $C : \Gamma \vdash \text{oft } M \; S$ then $E : \text{eq } T \; S$.

Induction on first typing derivation $D$.

Case 1

$D = \Gamma \vdash \text{oft } (\text{lam } x : T.M) \; (\text{arr } T \; S)$

$E : \text{eq } S \; S'$

$E : \text{eq } S \; S$ and $S = S'$

Therefore there is a proof for $\text{eq } (\text{arr } T \; S) \; (\text{arr } T \; S')$ by reflexivity.

Case 2

$D = x, u : \text{oft } x \; T \in \Gamma$

$D = \Gamma \vdash \text{oft } x \; T$

$C = \Gamma \vdash \text{oft } (\text{lam } x : T.M) \; (\text{arr } T \; S')$

$C = \Gamma \vdash \text{oft } x \; T'$

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

$E : \text{eq } S \; S$ by inversion using reflexivity
Type uniqueness

**Theorem**

If \( \mathcal{D} : \Gamma \vdash \text{oft} \ M \ T \) and \( \mathcal{C} : \Gamma \vdash \text{oft} \ M \ S \) then \( \mathcal{E} : \text{eq} \ T \ S \).

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

**Case 1**

\[
\begin{align*}
\mathcal{D} & = \frac{\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S}{\Gamma \vdash \text{oft} \ (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S)} \quad \text{t\_lam} \\
\mathcal{C} & = \frac{\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S'}{\Gamma \vdash \text{oft} \ (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S')} \quad \text{t\_lam}
\end{align*}
\]

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

\( \mathcal{E} : \text{eq} \ S \ S \) and \( S = S' \)

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

by inversion using reflexivity

Therefore there is a proof for \( \text{eq} \ (\text{arr} \ T \ S) \ (\text{arr} \ T \ S') \) by reflexivity.

**Case 2**

\[
\begin{align*}
\mathcal{D} & = \frac{x, u : \text{oft} \ x \ T \in \Gamma}{\Gamma \vdash \text{oft} \ x \ T} \quad u \\
\mathcal{C} & = \frac{x, v : \text{oft} \ x \ S \in \Gamma}{\Gamma \vdash \text{oft} \ x \ S} \quad v
\end{align*}
\]
Type uniqueness

Theorem

If \( D : \Gamma \vdash \text{oft} \ M \ T \) and \( C : \Gamma \vdash \text{oft} \ M \ S \) then \( E : \text{eq} \ T \ S \).

Induction on first typing derivation \( D \).

**Case 1**

\[
\begin{align*}
\mathcal{D}_1 &: \Gamma, x, u : \text{oft} \times T \vdash \text{oft} \ M \ S \\
\mathcal{C}_1 &: \Gamma, x, u : \text{oft} \times T \vdash \text{oft} \ M \ S' \\
\end{align*}
\]

\[
\begin{align*}
\mathcal{D} &= \frac{\Gamma \vdash \text{oft} (\text{lam} x : \text{T}.M) \ (\text{arr} \ T \ S)}{t_{\text{lam}}} & \mathcal{C} &= \frac{\Gamma \vdash \text{oft} (\text{lam} x : \text{T}.M) \ (\text{arr} \ T \ S')}{t_{\text{lam}}} \\
\end{align*}
\]

\( E : \text{eq} \ S \ S' \)

\( E : \text{eq} \ S \ S \) and \( S = S' \)

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

by inversion using reflexivity

Therefore there is a proof for \( \text{eq} (\text{arr} \ T \ S) \ (\text{arr} \ T \ S') \) by reflexivity.

**Case 2**

\[
\begin{align*}
\mathcal{D} &= \frac{\Gamma \vdash \text{oft} \times T}{u} \\
\mathcal{C} &= \frac{\Gamma \vdash \text{oft} \times S}{v} \\
\end{align*}
\]

Every variable \( x \) is associated with a unique typing assumption (property of the context), hence \( v = u \) and \( S = T \).
This talk

Design and implementation of Beluga

- Introduction
- Example: Simply typed lambda calculus
- Writing a proof in Beluga . . .
- Wanting more . . .
  - Evaluation using closures
  - Normalization
- Conclusion
Beluga$\mu$: two level approach

Logical framework LF [HHP’93]

- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types
Beluga$^\mu$: two level approach

Logical framework LF [HHP’93]

- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types
  $\leadsto$ support for $\alpha$-renaming, substitution, adequate representations
Belugaμ: two level approach

Logical framework LF [HHP'93]

- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types
  \( \leadsto \) support for \( \alpha \)-renaming, substitution, adequate representations

Programming proofs [Pientka'08, Pientka,Dunfield'10]

Proof term language for first-order logic over a specific domain (= contextual LF) together with domain-specific induction principle and recursive definitions

- Contextual LF: Contextual types characterize contextual objects [NPP'08]
  \( \leadsto \) support well-scoped derivations
  \( \leadsto \) abstract notion of contexts and substitution
Beluga$^\mu$: two level approach

Logical framework LF [HHP'93]
- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types
  $\Rightarrow$ support for $\alpha$-renaming, substitution, adequate representations

Programming proofs [Pientka'08, Pientka,Dunfield’10]
Proof term language for first-order logic over a specific domain (= contextual LF) together with domain-specific induction principle and recursive definitions
- Contextual LF: Contextual types characterize contextual objects [NPP’08]
  $\Rightarrow$ support well-scoped derivations
  $\Rightarrow$ abstract notion of contexts and substitution
- Recursive definitions = Indexed Recursive Types [Cave,Pientka’12]
Beluga$\mu$: two level approach

Logical framework LF [HHP’93]
- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types
  $\rightsquigarrow$ support for $\alpha$-renaming, substitution, adequate representations

Programming proofs [Pientka’08, Pientka,Dunfield’10]
Proof term language for first-order logic over a specific domain (＝ contextual LF) together with domain-specific induction principle and recursive definitions
- Contextual LF: Contextual types characterize contextual objects [NPP’08]
  $\rightsquigarrow$ support well-scoped derivations
  $\rightsquigarrow$ abstract notion of contexts and substitution
- Recursive definitions = Indexed Recursive Types [Cave,Pientka’12]

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Proofs as functions in Beluga</th>
</tr>
</thead>
<tbody>
<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>
Step 1: Represent types and lambda-terms in LF

Types $T ::= \text{nat} \mid \text{arr } T_1 \ T_2$

Terms $M ::= x \mid \text{lam } x : T. M \mid \text{app } M N$
Step 1: Represent types and lambda-terms in LF

Types \( T \) ::= nat | arr \( T_1 \ T_2 \)

Terms \( M \) ::= \( x \) | lam \( x: T. M \) | app \( M N \)

LF representation in Beluga

```plaintext
datatype tp: type =
| nat: tp
| arr: tp \(\rightarrow\) tp \(\rightarrow\) tp;

datatype tm: type =
| lam: tp \(\rightarrow\) (tm \(\rightarrow\) tm) \(\rightarrow\) tm
| app: tm \(\rightarrow\) tm \(\rightarrow\) tm;
```

B. Pientka
Beluga\(\mu\):Programming proofs in context ...
Step 1: Represent types and lambda-terms in LF

Types $T ::= \text{nat} \mid \text{arr } T_1 T_2$

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

LF representation in Beluga

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

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

Typing rules

\[
\frac{\text{oft } M (\text{arr } T S) \quad \text{oft } N T}{\text{oft } (\text{app } M N) S} \quad \text{t_app}
\]

\[
\frac{\quad \vdots \quad u}{\text{oft } \lambda x: T.M} \quad \text{t_lam}^{x,u}
\]
Step 1: Represent types and lambda-terms in LF

Types $T ::= \text{nat} \\
| \text{arr } T_1 \ T_2$

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

LF representation in Beluga

```
datatype tp: type = 
| nat: tp \\
| arr: tp → tp → tp;

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

Typing rules

\[
\begin{align*}
\text{oft } M \ (\text{arr } T \ S) \ & \text{oft } N \ T \\
\text{oft } (\text{app } M \ N) \ S \\
\text{oft } (\text{lam } x: T.M) \ (\text{arr } T \ S) \\
\hline
\text{t_app} & \text{t_lam}^{x, u}
\end{align*}
\]

```
datatype oft: tm → tp → type = 
| t_app: oft M (arr T S) → oft N T → oft (app M N) S \\
| t_lam: (Π x:tm.oft x T → oft (M x) S) → oft (lam T M) (arr T S);
```
Step 2a: Theorem as type

Theorem

If \( \Gamma \vdash \text{oft} \ M \ T \) and \( \Gamma \vdash \text{oft} \ M \ S \) then \( \text{E} : \text{eq} \ T \ S \).

is represented as

Computation-level Type in Beluga

\[(\Gamma : \text{ctx}) \Rightarrow ([\Gamma \vdash \text{oft} (M \ldots) T] \rightarrow ([\Gamma \vdash \text{oft} (M \ldots) S] \rightarrow [\vdash \text{eq} \ T \ S])\]

Read as: “For all contexts \( \Gamma \) of the schema \text{ctx}, ...


• \([\Gamma \vdash \text{oft} (M \ldots) \ T]\) and \([\vdash \text{eq} \ T \ S]\) are contextual types [NPP’08].

• ...

... describes dependency on context.

\( T \) is a closed object \( (M \ldots) \) is an object which may depend on context \( \Gamma \).

• Contexts are structured sequences and are classified by context schemas

\( \text{schema} \ \text{ctx} = \text{some} [T : \text{tp}] \text{block} \ x : \text{tm}, u : \text{oft} x T. \)
Step 2a: Theorem as type

**Theorem**

If $\mathcal{D} : \Gamma \vdash \text{oft } M \ T$ and $\mathcal{C} : \Gamma \vdash \text{oft } M \ S$ then $\mathcal{E} : \text{eq } T \ S$. 

Read as: “For all contexts $\Gamma$ of the schema $\text{ctx}$, ...

$T$ is a closed object $(M ...)$ is an object which may depend on context $\Gamma$.

Contexts are structured sequences and are classified by context schemas $\text{schema } \text{ctx} = \text{some } [T: \text{tp}]$ block $x: \text{tm}, u: \text{oft } x \ T$. 

B. Pientka

Beluga$^\mu$: Programming proofs in context ...
Step 2a: Theorem as type

**Theorem**

If \( D : \Gamma \vdash \mathrm{oft} \ M \ T \) and \( C : \Gamma \vdash \mathrm{oft} \ M \ S \) then \( E : \mathrm{eq} \ T \ S \).

is represented as

**Computation-level Type in Beluga**

\[
(\Gamma : \text{ctx}) \ [\Gamma \vdash \mathrm{oft} \ (M \ldots) \ T] \rightarrow [\Gamma \vdash \mathrm{oft} \ (M \ldots) \ S] \rightarrow [\vdash \mathrm{eq} \ T \ S]
\]

Read as: "For all contexts \( \Gamma \) of the schema \( \text{ctx} \), ..."
Step 2a: Theorem as type

**Theorem**

If $D : \Gamma \vdash \text{oft } M T$ and $C : \Gamma \vdash \text{oft } M S$ then $E : \text{eq } T S$.

is represented as

**Computation-level Type in Beluga**

$$(\Gamma : \text{ctx}) \ [\Gamma \vdash \text{oft } (M \ldots) T] \rightarrow [\Gamma \vdash \text{oft } (M \ldots) S] \rightarrow [\vdash \text{eq } T S]$$

Read as: "For all contexts $\Gamma$ of the schema ctx, ...

- $[\Gamma \vdash \text{oft } (M \ldots) T]$ and $[\vdash \text{eq } T S]$ are contextual types [NPP'08].
- ... describes dependency on context.
  $T$ is a closed object $(M \ldots)$ is an object which may depend on context $\Gamma$.
- Contexts are structured sequences and are classified by context schemas
Step 2a: Theorem as type

**Theorem**

If $D : \Gamma \vdash \text{oft } M \, T$ and $C : \Gamma \vdash \text{oft } M \, S$ then $E : \text{eq } T \, S$.

is represented as

**Computation-level Type in Beluga**

$$(\Gamma : \text{ctx}) \, [\Gamma \vdash \text{oft } (M \ldots) \, T] \rightarrow [\Gamma \vdash \text{oft } (M \ldots) \, S] \rightarrow [\vdash \text{eq } T \, S]$$

Read as: "For all contexts $\Gamma$ of the schema $\text{ctx}$, ...

- $[\Gamma \vdash \text{oft } (M \ldots) \, T]$ and $[\vdash \text{eq } T \, S]$ are contextual types [NPP’08].
- ... describes dependency on context.
  - $T$ is a closed object $(M \ldots)$ is an object which may depend on context $\Gamma$.
- Contexts are structured sequences and are classified by context schemas
  
  ```
  \text{schema } \text{ctx} = \text{some } [T : \text{tp}] \ \text{block } x : \text{tm}, u : \text{oft } x \, T.
  ```
Step 2b: Proofs as Programs
Step 2b: Proofs as Programs

\[
\text{rec } \text{unique} : (\Gamma : \text{ctx})[\Gamma \vdash \text{oft } (M \ldots)T] \rightarrow [\Gamma \vdash \text{oft } (M \ldots)S] \rightarrow [\vdash \text{eq } T S] =
\]
Step 2b: Proofs as Programs

\[\text{rec unique:}(\Gamma: \text{ctx})[\Gamma \vdash \text{oft } (M \ldots)T] \rightarrow [\Gamma \vdash \text{oft } (M \ldots)S] \rightarrow [\vdash \text{eq } T \ S] =\]

\[\text{fn d} \Rightarrow \text{fn c} \Rightarrow \text{case } d \text{ of}\]
Step 2b: Proofs as Programs

\[
\begin{align*}
\text{rec} & \quad \text{unique:} (\Gamma : \text{ctx}) [\Gamma \vdash \text{oft} (M \ldots) T] \rightarrow [\Gamma \vdash \text{oft} (M \ldots) S] \rightarrow [\vdash \text{eq} T S] = \\
& \quad \text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of} \\
& \quad | \ [\Gamma \vdash t\text{\_app} (D_1 \ldots) (D_2 \ldots)] \Rightarrow \% \text{Application Case} \\
& \quad \quad \text{let } [\Gamma \vdash t\text{\_app} (C_1 \ldots) (C_2 \ldots)] = c \text{ in} \\
& \quad \quad \text{let } [\vdash e\text{\_ref}] = \text{unique} [\Gamma \vdash D_1 \ldots] [\Gamma \vdash C_1 \ldots] \text{ in} \\
& \quad \quad \quad [\vdash e\text{\_ref}] 
\end{align*}
\]
Step 2b: Proofs as Programs

\[
\begin{align*}
\text{rec} & \quad \text{unique:} (\Gamma: \text{ctx}) \rightarrow [\Gamma \vdash \text{oft} (M \ldots) T] \rightarrow [\Gamma \vdash \text{oft} (M \ldots) S] \rightarrow [\vdash \text{eq} T S] = \\
\text{fn} & \quad d \Rightarrow \text{fn} c \Rightarrow \text{case} d \text{ of} \\
\quad | [\Gamma \vdash t_{\text{app}} (D1 \ldots) (D2 \ldots)] & \Rightarrow \% \text{Application Case} \\
\quad & \quad \text{let} \quad [\Gamma \vdash t_{\text{app}} (C1 \ldots) (C2 \ldots)] = c \in \\
\quad & \quad \text{let} \quad [\vdash e_{\text{ref}}] = \text{unique} [\Gamma \vdash D1 \ldots] [\Gamma \vdash C1 \ldots] \in \\
\quad & \quad [\vdash e_{\text{ref}}] \\
\quad | [\Gamma \vdash t_{\text{lam}} (\lambda x. \lambda u. D \ldots x u)] & \Rightarrow \% \text{Abstraction Case} \\
\quad & \quad \text{let} \quad [\Gamma \vdash t_{\text{lam}} (\lambda x. \lambda u. C \ldots x u)] = c \in \\
\quad & \quad \text{let} \quad [\vdash e_{\text{ref}}] = \text{unique} [\Gamma, b: \text{block} x: \text{tm}, u: \text{oft} x \_ \vdash D \ldots b.1 b.2] \\
\quad & \quad \quad [\Gamma, b \vdash C \ldots b.1 b.2] \in \\
\quad & \quad [\vdash e_{\text{ref}}]
\end{align*}
\]
Step 2b: Proofs as Programs

rec unique: (Γ:ctx) [Γ ⊢ oft (M ... ) T] → [Γ ⊢ oft (M ... ) S] → [⊢ eq T S] =

fn d ⇒ fn c ⇒ case d of
   | [Γ ⊢ t_app (D1 ...) (D2 ...)] ⇒ % Application Case
     let [Γ ⊢ t_app (C1 ...) (C2 ...)] = c in
     let [ ⊢ e_ref] = unique [Γ ⊢ D1 ...] [Γ ⊢ C1 ...] in
     [ ⊢ e_ref]

   | [Γ ⊢ t_lam (λx.λu. D ... x u)] ⇒ % Abstraction Case
     let [Γ ⊢ t_lam (λx.λu. C ... x u)] = c in
     let [ ⊢ e_ref] = unique [Γ, b: block x:tm, u:oft x _ ⊢ D ... b.1 b.2]
                            [Γ, b ⊢ C ... b.1 b.2] in
     [ ⊢ e_ref]

   | [Γ ⊢ #q.2 ...] ⇒ % d : oft (#q.1 ...) T % Assumption Case
     let [Γ ⊢ #r.2 ...] = c in % c : oft (#r.1 ...) S
     [ ⊢ e_ref];

Recall:
#q: block x:tm, u:oft x T
#r: block x:tm, u:oft x S

We also know:
#r.1 = #q.1
Therefore:
T = S
Step 2b: Proofs as Programs

```plaintext
rec unique: (Γ : ctx) [Γ ⊢ oft (M ...) T → [Γ ⊢ oft (M ...) S] → [⊢ eq T S] =

fn d ⇒ fn c ⇒ case d of
| [Γ ⊢ t_app (D1 ...) (D2 ...)] ⇒ % Application Case
  let [Γ ⊢ t_app (C1 ...) (C2 ...)] = c in
  let [⊢ e_ref] = unique [Γ ⊢ D1 ...] [Γ ⊢ C1 ...] in
  [⊢ e_ref]

| [Γ ⊢ t_lam (λx.λu. D ... x u)] ⇒ % Abstraction Case
  let [Γ ⊢ t_lam (λx.λu. C ... x u)] = c in
  let [⊢ e_ref] = unique [Γ, b : block x : tm, u : oft x _ ⊢ D ... b.1 b.2]
    [Γ, b ⊢ C ... b.1 b.2] in
  [⊢ e_ref]

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

Recall:

#q : block x : tm, u : oft x T
#r : block x : tm, u : oft x S
Step 2b: Proofs as Programs

\[
\begin{align*}
\text{rec } & \text{unique} : (\Gamma : \text{ctx})[\Gamma \vdash \text{oft } (M \ldots)T ] \rightarrow [\Gamma \vdash \text{oft } (M \ldots)S ] \rightarrow [\vdash \text{eq } T S ] = \\
\text{fn } & \text{d } \Rightarrow \text{fn } \text{c } \Rightarrow \text{case } \text{d } \text{of} \\
& | [\Gamma \vdash t_{\text{app}} (D_1 \ldots) (D_2 \ldots)] \Rightarrow \quad \% \text{Application Case} \\
& \quad \text{let } [\Gamma \vdash t_{\text{app}} (C_1 \ldots) (C_2 \ldots)] = \text{c } \text{in} \\
& \quad \text{let } [\vdash e_{\text{ref}}] = \text{unique } [\Gamma \vdash D_1 \ldots] [\Gamma \vdash C_1 \ldots] \text{ in} \\
& \quad [\vdash e_{\text{ref}}] \\
& | [\Gamma \vdash t_{\text{lam}} (\lambda x. \lambda u. D \ldots x u) ] \Rightarrow \quad \% \text{Abstraction Case} \\
& \quad \text{let } [\Gamma \vdash t_{\text{lam}} (\lambda x. \lambda u. C \ldots x u)] = \text{c } \text{in} \\
& \quad \text{let } [\vdash e_{\text{ref}}] = \text{unique } [\Gamma, b : \text{block } x : \text{tm}, u : \text{oft } x \_ \vdash D \ldots b.1 \ b.2] \\
& \quad \quad [\Gamma, b \vdash C \ldots b.1 \ b.2] \text{ in} \\
& \quad [\vdash e_{\text{ref}}] \\
& | [\Gamma \vdash \#q.2 \ldots] \Rightarrow \quad \% \text{d } : \text{oft } (\#q.1 \ldots) T \quad \% \text{Assumption Case} \\
& \quad \text{let } [\Gamma \vdash \#r.2 \ldots] = \text{c } \text{in} \ % \text{c } : \text{oft } (\#r.1 \ldots) S \\
& \quad [\vdash e_{\text{ref}}] ;
\end{align*}
\]

Recall:

|\#q: \text{block } x : \text{tm}, u : \text{oft } x \ T |
|\#r: \text{block } x : \text{tm}, u : \text{oft } x \ S |

We also know: \#r.1 = \#q.1
Step 2b: Proofs as Programs

```
rec unique:(\Gamma:ctx)[\Gamma \vdashoft (M ...) T] \rightarrow[\Gamma \vdashoft (M ...) S] \rightarrow[\vdasheq T S] =

fn d ⇒ fn c ⇒ case d of
| [\Gamma \vdash t_app (D1 ...) (D2 ...)] ⇒ % Application Case
  let [\Gamma \vdash t_app (C1 ...) (C2 ...)] = c in
  let [\vdash e_ref] = unique [\Gamma \vdash D1 ...] [\Gamma \vdash C1 ...] in
  [\vdash e_ref]

| [\Gamma \vdash t_lam (\lambda x.\lambda u. D ... x u)] ⇒ % Abstraction Case
  let [\Gamma \vdash t_lam (\lambda x.\lambda u. C ... x u)] = c in
  let [\vdash e_ref] = unique [\Gamma,b:block x:tm, u:oft x _ \vdash D ... b.1 b.2]
              [\Gamma,b \vdash C ... b.1 b.2] in
  [\vdash e_ref]

| [\Gamma \vdash #q.2 ...] ⇒ % d : oft (#q.1 ...) T % Assumption Case
  let [\Gamma \vdash #r.2 ...] = c in % c : oft (#r.1 ...) S
  [\vdash e_ref] ;
```

Recall:

```
#q:block x:tm, u:oft x T
#r:block x:tm, u:oft x S
```

We also know:  

```
#r.1 = #q.1
```

Therefore:

```
T = S
```
Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Implementation in Beluga [IJCAR’10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-formed derivations</td>
<td>Dependent types</td>
</tr>
<tr>
<td>Renaming, Substitution</td>
<td>α-renaming, β-reduction in LF</td>
</tr>
</tbody>
</table>
Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Implementation in Beluga [IJCAR’10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-formed derivations</td>
<td>Dependent types</td>
</tr>
<tr>
<td>Renaming, Substitution</td>
<td>$\alpha$-renaming, $\beta$-reduction in LF</td>
</tr>
<tr>
<td>Well-scoped derivation</td>
<td>Contextual types and objects</td>
</tr>
</tbody>
</table>
## Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Implementation in Beluga [IJCAR’10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-formed derivations</td>
<td>Dependent types</td>
</tr>
<tr>
<td>Renaming, Substitution</td>
<td>$\alpha$-renaming, $\beta$-reduction in LF</td>
</tr>
<tr>
<td>Well-scoped derivation</td>
<td>Contextual types and objects</td>
</tr>
<tr>
<td>Context</td>
<td>Context schemas</td>
</tr>
</tbody>
</table>
Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Implementation in Beluga [IJCAR’10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-formed derivations</td>
<td>Dependent types</td>
</tr>
<tr>
<td>Renaming, Substitution</td>
<td>(\alpha)-renaming, (\beta)-reduction in LF</td>
</tr>
<tr>
<td>Well-scoped derivation</td>
<td>Contextual types and objects</td>
</tr>
<tr>
<td>Context</td>
<td>Context schemas</td>
</tr>
<tr>
<td>Properties of contexts (weakening, uniqueness)</td>
<td>Typing for schemas</td>
</tr>
</tbody>
</table>
Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Implementation in Beluga [IJCAR’10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-formed derivations</td>
<td>Dependent types</td>
</tr>
<tr>
<td>Renaming, Substitution</td>
<td>$\alpha$-renaming, $\beta$-reduction in LF</td>
</tr>
<tr>
<td>Well-scoped derivation</td>
<td>Contextual types and objects</td>
</tr>
<tr>
<td>Context</td>
<td>Context schemas</td>
</tr>
<tr>
<td>Properties of contexts (weakening, uniqueness)</td>
<td>Typing for schemas</td>
</tr>
</tbody>
</table>

- Compact representation of proofs as functions [POPL’08, PPDP08]

<table>
<thead>
<tr>
<th>Case analysis</th>
<th>Case analysis and pattern matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inversion</td>
<td>Pattern matching using let-expression</td>
</tr>
<tr>
<td>Induction Hypothesis</td>
<td>Recursive call</td>
</tr>
</tbody>
</table>
Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

<table>
<thead>
<tr>
<th>On paper proof</th>
<th>Implementation in Beluga [IJCAR’10]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-formed derivations</td>
<td>Dependent types</td>
</tr>
<tr>
<td>Renaming, Substitution</td>
<td>(\alpha)-renaming, (\beta)-reduction in LF</td>
</tr>
<tr>
<td>Well-scoped derivation</td>
<td>Contextual types and objects</td>
</tr>
<tr>
<td>Context</td>
<td>Context schemas</td>
</tr>
<tr>
<td>Properties of contexts (weakening, uniqueness)</td>
<td>Typing for schemas</td>
</tr>
</tbody>
</table>

- Compact representation of proofs as functions [POPL’08, PPDP08]

<table>
<thead>
<tr>
<th>Case analysis</th>
<th>Case analysis and pattern matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inversion</td>
<td>Pattern matching using let-expression</td>
</tr>
<tr>
<td>Induction Hypothesis</td>
<td>Recursive call</td>
</tr>
</tbody>
</table>
Comparison

- **Twelf [Pf, Sch’99]:** Encode proofs as relations
  - Requires lemma to prove injectivity of \( arr \) constructor.
  - No explicit contexts (cannot express types \( T \) and \( S \) and \( eq \ T \ S \) are closed)
  - 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 lookup

- **Abella [Gacek’08], Tac[Baelde’10]:** Proof assistants based on proof theory
  - Equality built-into the logic
  - Contexts are represented as lists
  - Requires lemmas about these lists (for example that all assumptions occur uniquely)
This talk

Design and implementation of Beluga

- Introduction
- Example: Simply typed lambda calculus
- Writing a proof in Beluga . . .
- Wanting more . . .
  - Evaluation using closures
  - Normalization
- Conclusion
Example: Evaluator using closures

- Lambda-terms and closures

Terms \( M, N := x | \lambda x. M | M N \)

Closures \( C := \text{Cl}(x.M, \rho) \)

Environment \( \rho := \cdot | \rho, (x, C) \)

- Meaning of \( \text{Cl}(x.M, \rho) \): \( \rho \) provides instantiations for all the free variables in \( x.M \).

- Environment \( \rho \) is a mapping from variables to closures
Example: Evaluator using closures

- Lambda-terms and closures

Terms \[ M, N ::= x \mid \lambda x. M \mid M N \]

Closures \[ C ::= \text{Cl}(x.M, \rho) \]

Environment \[ \rho ::= \cdot \mid \rho, (x, C) \]

- Meaning of \( \text{Cl}(x.M, \rho) \): \( \rho \) provides instantiations for all the free variables in \( x.M \).

- Environment \( \rho \) is a mapping from variables to closures

- Evaluation: \( (M, \rho) \Downarrow C \)

\[
\frac{\text{lookup } x \rho = C}{(x, \rho) \Downarrow C} \quad \frac{(\lambda x. M, \rho) \Downarrow \text{Cl}(x.M, \rho)}{(M_1, \rho) \Downarrow \text{Cl}(x.N, \rho') \quad (M_2, \rho) \Downarrow C \quad (N, \rho', (x, C)) \Downarrow C'}{\quad (M_1 M_2, \rho) \Downarrow C'}
\]
Representing terms, contexts and closures

LF representation in Beluga

```plaintext
datatype tm: type =
  lam: (tm → tm) → tm
  app: tm → tm → tm;

schema ctx = tm;    % Define context schema
```

Note: → is overloaded.

- tm → tm is the LF function space: binders in the object language are modelled by LF functions
- \[ ψ \vdash tm \] → Clos is a computation-level function mapping variables of type tm in the context \( ψ \) to closures.
Representing terms, contexts and closures

LF representation in Beluga

```plaintext
datatype tm: type =
    lam: (tm → tm) → tm
    app: tm → tm → tm ;
schema ctx = tm; % Define context schema
```

Computation-level data types in Beluga

```plaintext
datatype Clos : ctype =
    Cl : (ψ:ctx) [ψ, x:tm ⊢ tm] → ([ψ ⊢ tm] → Clos)→ Clos ;
```

Note:
- `tm → tm` is the LF function space: binders in the object language are modelled by LF functions.
- `[[ψ ⊢ tm] → Clos]` is a computation-level function mapping variables of type `tm` in the context `ψ` to closures.
Representing terms, contexts and closures

LF representation in Beluga

```ml
datatype tm : type =
  lam : (tm -> tm) -> tm
  app : tm -> tm -> tm ;

schema ctx = tm;  % Define context schema
```

Computation-level data types in Beluga

```ml
datatype Clos : ctype =
  Cl : (ψ : ctx) [ψ, x:tm ⊨ tm] → ([ψ ⊨ tm] → Clos) → Clos ;
```

Note: → is overloaded.

- \( tm \rightarrow tm \) is the LF function space: binders in the object language are modelled by LF functions
- \([ψ ⊨ tm] \rightarrow Clos\) is a computation-level function mapping variables of type \( tm \) in the context \( ψ \) to closures.
Representing terms, contexts and closures (revised)

LF representation in Beluga

```
datatype tm: type = 
  lam: (tm → tm) → tm
  app: tm → tm → tm ;
schema ctx = tm; % Define context schema
```

Computation-level data types in Beluga

```
datatype Var : {ψ:ctx} ctype = V : {#p:[ψ ⊢ tm]} Var [ψ];
datatype Clos : ctype = 
  Cl : (ψ:ctx) [ψ, x:tm ⊢ tm] → (Var [ψ] → Clos) → Clos ;
```
Representing terms, contexts and closures (revised)

LF representation in Beluga

```plaintext
datatype tm : type =
  lam : (tm → tm) → tm
  app : tm → tm → tm ;
schema ctx = tm;    % Define context schema
```

Computation-level data types in Beluga

```plaintext
datatype Var : {ψ:ctx} ctype = V : {#p:[ψ ⊢ tm]} Var [ψ];
datatype Clos : ctype =
  Cl : (ψ:ctx) [ψ, x:tm ⊢ tm] → (Var [ψ] → Clos) → Clos ;
```

Note: Index computation-level types [POPL’12]

- `Var [ψ]` is an indexed type
- `V : {#p:[ψ . tm]} Var [ψ]` defines a constructor `v` which takes variables of type `tm` in the context `ψ` as argument (Cast)
Evaluation using closures
Define recursive program parametric in context

\[
\text{rec eval: } (\psi : \text{ctx}) [\psi \vdash \text{tm}] \to (\text{Var } [\psi] \to \text{Clos}) \to \text{Clos} =
\]
Evaluation using closures

\[
\text{rec eval: (ψ:ctx) [ψ ⊢ tm] → (Var [ψ] → Clos) → Clos = }
\]

\[
\text{fn e ⇒ fn env ⇒ case e of}
\]
Evaluation using closures

\[
\text{rec eval: } (\psi: \text{ctx}) \left[ \psi \vdash \text{tm} \right] \to (\text{Var } [\psi] \to \text{Clos}) \to \text{Clos } = \\
\text{fn e } \Rightarrow \text{fn env } \Rightarrow \text{case e of } \\
| \left[ \psi \vdash \#p \ldots \right] \Rightarrow \text{env (V } [\psi \vdash \#p \ldots ] \right)
\]
Evaluation using closures

\[
\text{rec eval: } (\psi: \text{ctx})[\psi \vdash \text{tm}] \rightarrow (\text{Var } [\psi] \rightarrow \text{Clos}) \rightarrow \text{Clos } =
\]

\[
\begin{align*}
\text{fn } e \Rightarrow \text{fn } \text{env } \Rightarrow \text{case } e \text{ of } \\
| [\psi \vdash \#p \ldots] \Rightarrow \text{env } (V [\psi \vdash \#p \ldots]) \\
| [\psi \vdash \text{lam } \lambda x. E \ldots x] \Rightarrow \text{Cl } [\psi, x: \text{tm } \vdash E \ldots x] \text{ env}
\end{align*}
\]
Evaluation using closures

\[
\text{rec eval}: (\psi:\text{ctx}) [\psi \vdash \text{tm}] \to (\text{Var} [\psi] \to \text{Clos}) \to \text{Clos} = \\
\text{fn } e \Rightarrow \text{fn } \text{env} \Rightarrow \text{case } e \text{ of} \\
\mid [\psi \vdash \#p \ldots] \Rightarrow \text{env} (V [\psi \vdash \#p \ldots]) \\
\mid [\psi \vdash \lambda x. E \ldots x] \Rightarrow \text{Cl} [\psi, x:tm \vdash E \ldots x] \text{ env} \\
\mid [\psi \vdash \text{app} (E1 \ldots) (E2 \ldots)] \Rightarrow \\
\quad \text{let } \text{Cl} [\phi, x:tm \vdash E \ldots x] \text{ env'} = \text{eval} [\psi \vdash E1 \ldots] \text{ env} \text{ in} \\
\quad \text{let } w = \text{eval} [\psi \vdash E2 \ldots] \text{ env} \text{ in} \\
\quad \text{eval} [\phi, x:tm \vdash E \ldots x] \\
\qquad (\text{fn } x \Rightarrow \text{case } x \text{ of} \\
\quad \quad | V [\phi, x:tm \vdash x] \Rightarrow w \\
\quad \quad | V [\phi, x:tm \vdash \#p \ldots] \Rightarrow \text{env'} (V [\phi \vdash \#p \ldots]))
\]
Evaluation using closures

```latex
rec eval: (ψ:ctx) [ψ ⊢ tm] → (Var [ψ] → Clos) → Clos =

fn e ⇒ fn env ⇒ case e of
| [ψ ⊢ #p ...] ⇒ env (V [ψ ⊢ #p ...])
| [ψ ⊢ lam λx. E ...x] ⇒ Cl [ψ, x:tm ⊢ E ...x] env
| [ψ ⊢ app (E1 ...) (E2 ...)] ⇒
  let Cl [φ, x:tm ⊢ E ... x] env’ = eval [ψ ⊢ E1 ...] env in
  let w = eval [ψ ⊢ E2 ...] env in
  eval [φ, x:tm ⊢ E ... x]
    (fn x ⇒ case x of
      | V [φ, x:tm ⊢ x] ⇒ w
      | V [φ, x:tm ⊢ #p ...] ⇒ env’ (V [φ ⊢ #p ...])
    )
```

Features

- Pattern matching on contextual objects \textbf{and} computation-level data constructors
- Matching on contexts to lookup variables
Weak Normalization

- Good benchmark
  - Twelf, Delphine are too weak (to do it directly)
  - Coq/Agda lack support for substitutions and binders
  - Abella allows normalization proofs but lacks support for contexts
Weak Normalization

- Good benchmark
  - Twelf, Delphin are too weak (to do it directly)
  - Coq/Agda lack support for substitutions and binders
  - Abella allows normalization proofs but lacks support for contexts

- Weak normalization for simply typed lambda calculus

**Theorem**

If \( \vdash M : A \) then \( M \) halts.

**Proof.**

1. Define reducibility candidate \( \mathcal{R}_A \)
2. If \( M \in \mathcal{R}_A \) then \( M \) halts.
3. Backwards closed: If \( M' \in \mathcal{R}_A \) and \( M \rightarrow M' \) then \( M \in \mathcal{R}_A \).
4. **Fundamental Lemma:** If \( \vdash M : A \) then \( M \in \mathcal{R}_A \). (Requires a generalization)
Representing terms and evaluation in LF

Revisiting our definition of lambda-terms

```
datatype tm : tp -> type =
| c : tm i
| lam: (tm A -> tm B) -> tm (arr A B)
| app: tm (arr A B) -> tm A -> tm B;
```

Operational semantics

```
datatype mstep : tm A -> tm A -> type =
| s/beta : mstep (app (lam M) N) (M N)
| s/app : mstep M M' -> mstep (app M N) (app M' N)
| s/refl : mstep M M
| s/trans: mstep M M' -> mstep M' N -> mstep M N;
```

A term $M$ halts if there exists a value $V$ s.t. $M \rightarrow^* V$.

```
datatype halts : tm A -> type =
| h/value : mstep M M' -> value M' -> halts M;
```
Reducibility Candidates

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

\[
\begin{align*}
\mathcal{R}_i &= \{ M \mid \text{halts } M \} \\
\mathcal{R}_{A \rightarrow B} &= \{ M \mid \text{halts } M \text{ and } \forall N \in \mathcal{R}_A, (M N) \in \mathcal{R}_B \}
\end{align*}
\]
Reducibility Candidates

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

\[
\mathcal{R}_i = \{ M \mid \text{halts } M \}
\]

\[
\mathcal{R}_{A \rightarrow B} = \{ M \mid \text{halts } M \text{ and } \forall N \in \mathcal{R}_A, (M \ N) \in \mathcal{R}_B \}
\]

Computation-level data types in Beluga

```ml
datatype Reduce : {A:[ ⊢ tp]} {M:[ ⊢ tm A]} ctype =
| 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];
```

- Not strictly positive definition, but stratified.
Redducibility Candidates

Redducibility candidates for terms $M \in R_A$:

\[
R_i = \{ M \mid \text{halts } M \}
\]

\[
R_{A \rightarrow B} = \{ M \mid \text{halts } M \text{ and } \forall N \in R_A, (M N) \in R_B \}
\]

Computation-level data types in Beluga

```
datatype Reduce : {A: [ \vdash tp]} {M: [ \vdash tm A]} ctype =
| I : [ \vdash \text{halts } M] \rightarrow Reduce [ \vdash i] [\vdash M]
| Arr : [ \vdash \text{halts } M] \rightarrow
  (\{N: [ \vdash tm A]\} Reduce [ \vdash A] [ \vdash N] \rightarrow Reduce [ \vdash B] [ \vdash \text{app } M N])
  \rightarrow Reduce [ \vdash \text{arr } A B] [ \vdash M];
```

- Not strictly positive definition, but stratified.

Redducibility candidates for substitutions $\sigma \in R_{\Gamma}$:

```
datatype RedSub : (\Gamma:ctx){\sigma: \vdash \Gamma} ctype =
| Nil : RedSub [ \vdash \_ ]
| Cons : RedSub [ \vdash \sigma] \rightarrow Reduce [ \vdash A] [ \vdash M] \rightarrow RedSub [ \vdash \sigma M ];
```
Generalization of Fundamental Lemma

Lemma (Main lemma)

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

Proof.

Case $\Gamma, x : A \vdash M : B$

$\Gamma \vdash \lambda x. M : A \rightarrow B$

$[\sigma](\lambda x. M) = \lambda x. ([\sigma, x/x]M)$ by properties of substitution

halts $(\lambda x. [\sigma, x/x]M)$ since it is a value

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

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

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

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

Hence $[\sigma](\lambda x. M) \in \mathcal{R}_{A \rightarrow B}$ by definition
Theorems as Computation-level Types

Lemma (Backward closed)

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

Computation-level Type in Beluga

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

Lemma (Main lemma)

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

Computation-level Type in Beluga

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

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

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

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

\[
\text{rec main : } \{\Gamma : \text{ctx}\}{M : [\Gamma \vdash \text{tm } A]} \rightarrow \text{RedSub } [\vdash \sigma] \rightarrow \text{Reduce } [\vdash A] [\vdash M \sigma] = \\
\text{mlam } \Gamma \Rightarrow \text{mlam } M \Rightarrow \text{fn } rs \Rightarrow \text{case } [\Gamma \vdash M \ldots] \text{ of} \\
\mid [\Gamma \vdash \# p \ldots] \Rightarrow \text{lookup } [\Gamma] [\Gamma \vdash \# p \ldots] rs
\]
**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
  | [Γ ⊢ lam (λx. M1 ... x)] ⇒
    Arr [ ⊢ h/value s/refl v/lam]
    (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
     (main [Γ,x:tm _] [Γ,x ⊢ M1 ... x] (Cons rs rN)))
```
Fundamental Lemma

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

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

mlam Γ ⇒ mlam M ⇒ fn rs ⇒ case [Γ \vdash M...] of
| [Γ \vdash #p ...] ⇒ lookup [Γ] [Γ \vdash #p ...] rs
| [Γ \vdash \text{lam} (\lambda x. M1 ... x)] ⇒
  \text{Arr} [ \vdash \text{h/value} s/\text{refl} v/lam]
  (mlam N ⇒ fn rN ⇒ closed [ \vdash \text{s/beta}]
    (main [Γ, x : \text{tm} _] [Γ, x \vdash M1 ... x] \text{(Cons rs rN)}))

| [Γ \vdash \text{app} (M1 ...) (M2 ...)] ⇒
  let Arr ha f = main [Γ] [Γ \vdash M1 ...] rs in
  f [ \vdash _ ] (main [Γ] [Γ \vdash M2 ...] rs)
Fundamental Lemma

\[
\text{rec closed : } \forall M, M'. [\vdash \text{mstep } M \Rightarrow M'] \Rightarrow \text{Reduce } [\vdash \text{A}] [\vdash M'] \Rightarrow \text{Reduce } [\vdash \text{A}] [\vdash M] = ? \; ;
\]

\[
\text{rec main : } \forall \Gamma, M. [\Gamma \vdash \text{tm } A] \Rightarrow \text{RedSub } [\vdash \sigma] \Rightarrow \text{Reduce } [\vdash \text{A}] [\vdash M \sigma] =
\]

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

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

\[
| [\Gamma \vdash \text{lam } (\lambda x. M_1 \ldots x)] \Rightarrow
\quad \text{Arr } [\vdash \text{h/value } s/\text{refl } v/\text{lam}]
\]

\[
\quad (\text{mlam } N \Rightarrow \text{fn } rN \Rightarrow \text{closed } [\vdash s/\text{beta}]
\]

\[
\quad (\text{main } [\Gamma, x: \text{tm } _\ldots] [\Gamma, x \vdash M_1 \ldots x] (\text{Cons } rs rN))
\]

\[
| [\Gamma \vdash \text{app } (M_1 \ldots) (M_2 \ldots)] \Rightarrow
\quad \text{let } \text{Arr } h/a = \text{main } [\Gamma] [\Gamma \vdash M_1 \ldots] rs \text{ in }
\]

\[
\quad f [\vdash _\ldots] (\text{main } [\Gamma] [\Gamma \vdash M_2 \ldots] rs)
\]

\[
| [\Gamma \vdash c] \Rightarrow \text{I } [\vdash \text{h/value } s/\text{refl } v/c];
\]

- 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
Other examples and comparison

- Other examples:
  - Weak normalization for which evaluates under lambda-abstraction
  - Algorithmic equality for LF (A. Cave) (draft available)

  \[\Rightarrow\] Sufficient evidence that Beluga is ideally suited to support such advanced proofs

- Comparison (concentrating on the given weak normalization proof)
  - Coq/Agda formalization with well-scoped de Bruijn indices: dozen additional lemmas
  - Abella: 4 additional lemmas and diverges a bit from on-paper proof
  - Twelf: Too weak to for directly encoding such proofs; Implement auxiliary logic.
What have we achieved?

- Foundation for programming proofs in context
  (joint work with A. Cave [POPL’12])
  - Proof term language for first-order logic over contextual LF as domain
  - Uniform treatment of contextual types, context, ...
  - Modular foundation for dependently-typed programming with phase-distinction ⇒ Generalization of DML and ATS
  - Non-termination or effects are allowed, although we often want to concentrate on pure total programs.

- Extending contextual LF with first-class substitutions and their equational theory (joint work with A. Cave [LFMTP’13])

- Rich set of examples
  - Type-preserving compiler for simply typed lambda-calculus (joint work with O. Savary Belanger, S. Monnier [CPP’13])
  - (Weak) Normalization proofs (A. Cave)

- Latest release in Jan’14: Support for indexed data types, first-class substitutions, equational theory behind substitutions
This talk

Design and implementation of Beluga

- Introduction
- Example: Simply typed lambda calculus
- Writing a proof in Beluga . . .
- Wanting more . . .
  - Evaluation using closures
  - Normalization
- Conclusion
This talk

Design and implementation of Beluga

- Introduction
- Example: Simply typed lambda calculus
- Writing a proof in Beluga . . .
- Wanting more . . .
  - Evaluation using closures
  - Normalization
- Conclusion
Conclusion

Beluga$\mu$: programming proofs in context

- **Level 1: Contextual LF**
  - Supports for specifying formal systems in LF
  - Embed contexts and contextual LF objects into computations and types
  - First-class substitution and contexts together with rich equational theory

- **Level 2: Functional programming language supporting indexed types**
  - Pattern match and manipulate contextual LF objects
  - Proof terms language for first-order logic over contextual LF
  - Supports indexed recursive types

⇒ Elegant and compact framework for programming proofs.

“A language that doesn’t affect the way you think about programming, is not worth knowing.” — Alan Perlis
Current work

- Prototype in OCaml (ongoing) (providing an interactive programming mode)
- Structural recursion (S. S. Ruan, A. Abel)
  Develops a foundation of structural recursive functions for Beluga; proof of normalization; prototype implementation under way
- Coinduction in Beluga (D. Thibodeau)
  Extending work on simply-typed copatterns [POPL’13] to Beluga
- Case study:
  - Certified compiler (O. Savary Belanger, CPP’13)
  - Proof-carrying authorization with constraints (Tao Xue)
- Extending Beluga to full dependent types (A. Cave)
- Type reconstruction for dependently typed programs (F. Ferreira)
Current work

- Prototype in OCaml (ongoing) (providing an interactive programming mode)
- Structural recursion (S. S. Ruan, A. Abel)
  Develops a foundation of structural recursive functions for Beluga; proof of normalization; prototype implementation under way
- Coinduction in Beluga (D. Thibodeau)
  Extending work on simply-typed copatterns [POPL’13] to Beluga
- Case study:
  - Certified compiler (O. Savary Belanger, CPP’13)
  - Proof-carrying authorization with constraints (Tao Xue)
- Extending Beluga to full dependent types (A. Cave)
- Type reconstruction for dependently typed programs (F. Ferreira)
- ORBI - Benchmarks for comparing systems supporting HOAS encodings (A. Felty, A. Momigliano)
Thank you!

Download prototype and examples at

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

Current Belugians: Brigitte Pientka, Mathias Puech, Tao Xue, Olivier Savary Belanger, Andrew Cave, Francisco Ferreira, Stefan Monnier, David Thibodeau, Sherry Shanshan Ruan, Shawn Otis