Introduction

Beluga: Design and implementation

Programming in context

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?
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.
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: Type uniqueness proof
- Writing a proof in Beluga . . .
- Wanting more: Programming code transformations
  - Sketching closure conversion
  - Sketching normalization by evaluation
- Conclusion

“The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.”

- Edsger Dijkstra
This talk

Design and implementation of Beluga

- Introduction
- Example: Type uniqueness proof
- Writing a proof in Beluga . . .
- Wanting more: Programming code transformations
  - Sketching closure conversion
  - Sketching normalization by evaluation
- Conclusion

“The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.”

- Edsger Dijkstra
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: $\Gamma \vdash M : T$ read as "$M$ has type $T$"

Typing rules (Gentzen-style, context-free)

1. $\Gamma \vdash x : T$
2. $\Gamma \vdash M : S$
3. $\Gamma, x : T \vdash \text{lam } x : T . M : \text{arr } T S$
4. $\Gamma \vdash M : \text{arr } T S$
5. $\Gamma \vdash N : T$
6. $\Gamma \vdash (\text{app } M \ N) : S$
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 $T ::= \text{nat}$

| arr $T_1 \ T_2$

Terms $M ::= \ x$

| lam $x : T \ M$

| app $M \ N$

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

Typing rules (Gentzen-style, context-free)

\[
\begin{align*}
\text{oft} \ x & \ T \ u \\
\vdots \\
\text{oft} \ M \ S & \\
\text{oft} \ (\text{lam} \ x : T \ M) \ (\text{arr} \ T \ S) & \ \text{t\text{-}lam}^{x,u}
\end{align*}
\]
Simply typed lambda-calculus

Types and Terms

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

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

Typing rules (Gentzen-style, context-free)

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

\( t_{\text{lam}}^{x, u} \) \hspace{1cm} \( t_{\text{app}} \)
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: \( \text{oft} \, M \, T \) read as “\( M \) has type \( T \)”

Typing rules (Gentzen-style, context-free)

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

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 \( T \) ::= nat

\( | \) \( T_1 \rightarrow T_2 \)

Terms \( M \) ::= \( x \)

\( | \) \( \text{lam } x : T . M \)

\( | \) \( \text{app } M N \)

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

Typing rules

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

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

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

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

We are introducing the variable \( x \) together with the assumption \( \text{oft } x \ T \)
Talking about derivations

Typing rules

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

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

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

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

What properties do contexts have?

- 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.
Talking about derivations

Typing rules

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

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

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

• What kinds of variables are used?
Talking about derivations

Typing rules

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

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

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

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

Typing rules

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

\[
\begin{align*}
\frac{\Gamma, x, u : \text{oft} \times T \vdash \text{oft} M \; S}{\Gamma \vdash \text{oft} (\lambda x : T. M) \; (\text{arr} T \; S)} & \quad \text{t}_{\lambda x, u} \\
\frac{\Gamma \vdash \text{oft} M \; (\text{arr} T \; S) \quad \Gamma \vdash \text{oft} N \; T}{\Gamma \vdash \text{oft} (\text{app} M \; N) \; S} & \quad \text{t}_{\text{app}}
\end{align*}
\]

- 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

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

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

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

- 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**
Talking about derivations

Typing rules

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

\[
\begin{align*}
\Gamma, x, u : \text{oft} \times T & \vdash \text{oft} M S & \text{t\textsubscript{lam}\textsubscript{x,u}} \\
\Gamma & \vdash \text{oft} (\text{lam} x : T.M) (\text{arr} T S) \\
\Gamma & \vdash \text{oft} (\text{app} M N) S
\end{align*}
\]

- 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**

- What properties do contexts have?
Talking about derivations

Typing rules

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

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

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

- 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**

- What properties do contexts have? **Every declaration is unique, weakening, substitution lemma, etc.**
Talking about derivations

Typing rules

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

- 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**
- What properties do contexts have? **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.
**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

$D = D_1 : \Gamma, x, u : \text{oft } x T \vdash \text{oft } M S$

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

$C = C_1 : \Gamma, x, u : \text{oft } x T \vdash \text{oft } M S'$

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

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

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

Case 2

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

$u \Gamma \vdash \text{oft } x T$

$C = x, v : \text{oft } x S \in \Gamma$

$v \Gamma \vdash \text{oft } x S$

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

B. Pientka

Beluga$\mu$: Programming proofs in context ...
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**

$\mathcal{D}_1$

\[
\mathcal{D} = \frac{\Gamma, x, u : \text{oft } x \, T \vdash \text{oft } M \, S}{\Gamma \vdash \text{oft} \ (\text{lam } x : T \cdot M) \ (\text{arr } T \, S)} \quad \text{t}_{\text{lam}}
\]

$\mathcal{C}_1$

\[
\mathcal{C} = \frac{\Gamma, x, u : \text{oft } x \, T \vdash \text{oft } M \, S'}{\Gamma \vdash \text{oft} \ (\text{lam } x : T \cdot M) \ (\text{arr } T \, S')} \quad \text{t}_{\text{lam}}
\]
Type uniqueness

Theorem

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

Induction on first typing derivation \( D \).

Case 1

\[
\begin{align*}
D &= D_1 \\
\Gamma, x, u : \text{oft} \ x \ T &\vdash \text{oft} \ M \ S \\
\Gamma &\vdash \text{oft} \ (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S) \\
\mathcal{E} &\vdash \text{eq} \ S \ S'
\end{align*}
\]

\[
\begin{align*}
C &= C_1 \\
\Gamma, x, u : \text{oft} \ x \ T &\vdash \text{oft} \ M \ S' \\
\Gamma &\vdash \text{oft} \ (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S') \\
\text{by i.h. using } D_1 \text{ and } C_1
\end{align*}
\]
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**

\[
\begin{align*}
\mathcal{D} &= \quad \mathcal{D}_1 \\
\Gamma, x, u : \text{oft } x \ T &\vdash \text{oft } M \ S \\
\Gamma &\vdash \text{oft } (\text{lam } x : T.M) (\text{arr } T \ S) \\
\mathcal{E} &\vdash \text{eq } S \ S' \\
\mathcal{E} &\vdash \text{eq } S \ S \quad \text{and } S = S'
\end{align*}
\]

\[
\begin{align*}
\mathcal{C} &= \quad \mathcal{C}_1 \\
\Gamma, x, u : \text{oft } x \ T &\vdash \text{oft } M \ S' \\
\Gamma &\vdash \text{oft } (\text{lam } x : T.M) (\text{arr } T \ S') \\
\mathcal{E} &\vdash \text{eq } S \ S' \\
\mathcal{E} &\vdash \text{eq } S \ S \quad \text{and } S = S'
\end{align*}
\]

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

by inversion using 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**

\[
D = \begin{array}{l}
\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S \\
\Gamma \vdash \text{oft} \ (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S)
\end{array}
\]

\[
C = \begin{array}{l}
\Gamma, x, u : \text{oft} \ x \ T \vdash \text{oft} \ M \ S' \\
\Gamma \vdash \text{oft} \ (\text{lam} \ x : T.M) \ (\text{arr} \ T \ S')
\end{array}
\]

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

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

By i.h. using \( D_1 \) and \( C_1 \)

By inversion using reflexivity

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

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

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

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

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

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

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

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

by i.h. using \( D_1 \) and \( C_1 \)

by inversion using reflexivity

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

**Case 2**

\[
D = \frac{x, u : \text{oft} \ x \ T \in \Gamma}{\Gamma \vdash \text{oft} \ x \ T} \quad \frac{C = x, v : \text{oft} \ x \ S \in \Gamma}{\Gamma \vdash \text{oft} \ x \ S}
\]
**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**

\[
D_1 = \Gamma, x, u: \text{oft} \ x \ T \vdash \text{oft} \ M \ S
\]

By i.h. using \( D_1 \) and \( C_1 \)

\[
C_1 = \Gamma, x, u: \text{oft} \ x \ T \vdash \text{oft} \ M \ S'
\]

by inversion using reflexivity

\[
E : \text{eq} \ S \ S'
\]

\[
E : \text{eq} \ S \ S \quad \text{and} \quad S = S'
\]

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

**Case 2**

\[
D = x, u: \text{oft} \ x \ T \in \Gamma
\]

\[
C = x, v: \text{oft} \ S \in \Gamma
\]

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: Type uniqueness
- Writing a proof in Beluga ...
- Wanting more: Programming code transformations
  - Sketching closure conversion
  - Sketching normalization by evaluation
- 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
  $\rightsquigarrow$ support for $\alpha$-renaming, substitution, adequate representations
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, 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>
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, 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>

- Contextual types characterize contextual objects [NPP’08]
  $\Rightarrow$ support well-scoped derivations
Beluga\(\mu\): two level approach

Logical framework LF [HHP’93]

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

Programming proofs [Pientka’08, Pientka,Dunfield’10, 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>

- Contextual types characterize contextual objects [NPP’08]
  \(\rightsarrow\) support well-scoped derivations
- Context variables parameterize computations
  \(\rightsarrow\) fine grained invariants; distinguish between different contexts
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

Programming proofs [Pientka’08, Pientka,Dunfield’10, 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>

- Contextual types characterize contextual objects [NPP’08]
  $\leadsto$ support well-scoped derivations
- Context variables parameterize computations
  $\leadsto$ fine grained invariants; distinguish between different contexts
- Recursive types express relationships between contexts and contextual objects
  $\leadsto$ adds expressive power! (See POPL’12)
Step 1: Represent types and lambda-terms in LF

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

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

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

Typing rules

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

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

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

Typing rules

```
\[
\begin{align*}
\text{oft } M \ (\text{arr } T \ S) & \quad \text{oft } N \ T \\
\text{oft } (\text{app } M \ N) \ S & \quad \text{oft } M \ S \\
\text{oft } \text{app } & \quad \text{oft } \text{lam } ^x u \\
\end{align*}
\]
```

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

```haskell
datatype oft: exp → tp → type =
| t_app: oft M (arr T S) → oft N T → oft (app M N) S
| t_lam: (Π x:exp.oft x T → oft (M x) S) → oft (lam T M) (arr T S);
```
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

\[\text{Computation-level Type in Beluga}\]

\[(g: \text{ctx}) \ [g.\text{oft } (M \ ...)] \rightarrow \ [g.\text{oft } (M \ ...)] \rightarrow \ .\text{eq } T \ S\]

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

\[\text{\bullet }\ [g.\text{oft } (M \ ...)] \text{ and } .\text{eq } T \ S\] are contextual types [NPP’08].

\[\text{\bullet }\ ...\] describes dependency on context.

\(T\) is a closed object \((M \ ...)\) is an object which may depend on context \(g\).
Step 2a: Theorem as type

Theorem

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

$$(g : \text{ctx}) \left[ g.\text{oft} \,(M \, \ldots) \, T \right] \rightarrow \left[ g.\text{oft} \,(M \, \ldots) \, S \right] \rightarrow \left[ \text{eq} \, T \, S \right]$$

Read as: "For all contexts $g$ 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

$\langle g: \text{ctx} \rangle [g.\text{oft } (M ...) T] \rightarrow [g.\text{oft } (M ...) S] \rightarrow [\text{eq } T S]$  

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

- $[g.\text{oft } (M ...) T]$ and $[\text{eq } T S]$ are contextual types [NPP’08].
- ... describes dependency on context.
  - $T$ is a closed object
  - $(M ...)$ is an object which may depend on context $g$. 
Intrinsic support for contexts

<table>
<thead>
<tr>
<th>Computation-level Type in Beluga</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(g:\text{ctx}) \left[ g.\text{oft} (M \ldots) T \right] \rightarrow \left[ g.\text{oft} (M \ldots) S \right] \rightarrow \left[ .\text{eq} \ T \ S \right]]</td>
</tr>
</tbody>
</table>

- Parameterize computation over contexts, Distinguish between contexts.
Intrinsic support for contexts

Computation-level Type in Beluga

\[(g:\text{ctx}) \ [g.\text{oft} \ (M\ldots) \ T] \rightarrow [g.\text{oft} \ (M\ldots) \ S] \rightarrow [.eq \ T \ S]\]

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

Computational-level Type in Beluga

\[(g:ctx) [g.oft (M \ldots) T] \rightarrow [g.oft (M \ldots) S] \rightarrow [.eq T S]\]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  
  \[
  \text{schema } \text{ctx} = \text{some } [T:tp] \text{ block } x:exp, u:oft x T.
  \]
Intrinsic support for contexts

**Computation-level Type in Beluga**

\[(g\,ctx) \, [g\,oft\ (M\ ...)\ T] \rightarrow [g\,oft\ (M\ ...)\ S] \rightarrow [.eq\ T\ S]\]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  
  ```schema ctx = some [T:tp] block x:exp, u:oft x T.```

- \(x, u\): oft \(x\) nat, \(y, v\): oft \(y\) (arr nat nat) is represented as
  
Intrinsic support for contexts

Computation-level Type in Beluga

\[(g: \text{ctx}) \ [g.\text{oft} (M \ldots) \ T] \rightarrow [g.\text{oft} (M \ldots) \ S] \rightarrow \ [.\text{eq} \ T \ S]\]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  \[\text{schema} \ \text{ctx} = \text{some} \ [T: \text{tp}] \ \text{block} \ x: \text{exp}, u: \text{oft} x \ T.\]
- \(x, u: \text{oft} x \ \text{nat}, y, v: \text{oft} y \ (\text{arr} \ \text{nat} \ \text{nat})\) is represented as
  \[b1: \text{block} \ x: \text{exp}, u: \text{oft} x \ \text{nat}, b2: \text{block} \ y: \text{exp}, v: \text{oft} y \ (\text{arr} \ \text{nat} \ \text{nat}).\]
- Well-formedness: \(b1: \text{block} \ x: \text{exp}, u: \text{oft} y \ \text{nat}\) is ill-formed.
  \[x: \text{exp}, y: \text{exp}, u: \text{oft} x \ \text{nat}\] is ill-formed.
Intrinsic support for contexts

Parameterize computation over contexts, Distinguish between contexts.

Contexts are classified by context schemas

schema ctx = some [T:tp] block x:exp, u:oft x T.

x, u: oft x nat, y, v: oft y (arr nat nat) is represented as

Well-formedness: b1: block x:exp, u:oft y nat is ill-formed.

Declarations are unique: b1 is different from b2
Intrinsic support for contexts

Computation-level Type in Beluga

\[(g:\text{ctx}) \ [g.\text{oft} \ (M \ldots) \ T] \rightarrow [g.\text{oft} \ (M \ldots) \ S] \rightarrow [.eq \ T \ S]\]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  \[
  \text{schema} \ ctx = \text{some} \ [T]:\text{tp} \ \text{block} \ x:exp, u:oft \ x \ T.
  \]
- \(x, u: \text{oft} \ x \ \text{nat}, y, v: \text{oft} \ y \ (\text{arr} \ \text{nat} \ \text{nat})\) is represented as
  \[
  b1: \text{block} \ x:exp, u:oft \ x \ \text{nat}, \ b2: \text{block} \ y:exp, v:oft \ y \ (\text{arr} \ \text{nat} \ \text{nat}).
  \]
- Well-formedness:
  
  \[
  b1: \text{block} \ x:exp, u:oft \ y \ \text{nat} \quad \text{is ill-formed.}
  \]
  
  \[
  x:exp, y:exp, u:oft \ x \ \text{nat} \quad \text{is ill-formed.}
  \]
- Declarations are unique:
  
  \[
  b1 \quad \text{is different from} \quad b2
  \]
  
  \[
  b1.1 \quad \text{is different from} \quad b2.1
  \]
Intrinsic support for contexts

Computation-level Type in Beluga

\[(g:\text{ctx}) [g.\text{oft} (M \ldots) T] \rightarrow [g.\text{oft} (M \ldots) S] \rightarrow [.\text{eq} T S]\]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  
  schema ctx = some [T:tp] block x:exp, u:oft x T.

- \(x, u: \text{oft} x \\text{nat}, y, v: \text{oft} y (\text{arr nat nat})\) is represented as
  

- Well-formedness:
  
  b1: block x:exp, u:oft y nat is ill-formed.
  
  x:exp, y:exp, u:oft x nat is ill-formed.

- Declarations are unique:
  
  b1 is different from b2
  
  b1.1 is different from b2.1

- Later declarations overshadow earlier ones
Intrinsic support for contexts

**Computation-level Type in Beluga**

\[(g:ctx) \ [g.oft (M \ldots) T] \rightarrow [g.oft (M \ldots) S] \rightarrow [.eq T S]\]

- Parameterize computation over contexts, Distinguish between contexts.
- Contexts are classified by context schemas
  
  **schema** \(ctx = \text{some } [T:tp] \text{ block } x:exp, u:oft x T.\)

- \(x, u: \text{oft } x \text{ nat}, y, v: \text{oft } y \text{ (arr nat nat)}\) is represented as
  
  \(b_1: \text{block } x:exp, u:oft x \text{ nat}, b_2: \text{block } y:exp, v:oft y \text{ (arr nat nat)}\).

- Well-formedness: \(b_1: \text{block } x:exp, u:oft y \text{ nat} \) is ill-formed.
  
  \(x:exp, y:exp, u:oft x \text{ nat} \) is ill-formed.

- Declarations are unique: \(b_1 \) is different from \(b_2\)
  
  \(b_1.1 \) is different from \(b_2.1\)

- Later declarations overshadow earlier ones

- Weakening, Substitution lemma
Accessing objects in contexts

- How do we access objects from a context?
### Accessing objects in contexts

- **How do we access objects from a context?**

<table>
<thead>
<tr>
<th>Context</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b : \text{block} \ x : \text{exp}, u : \text{oft} x \text{ nat} )</td>
<td>( b.2 ) concrete parameter retrieves the second component of ( b )</td>
</tr>
</tbody>
</table>
# Accessing objects in contexts

- **How do we access objects from a context?**

<table>
<thead>
<tr>
<th>Context</th>
<th>Element</th>
</tr>
</thead>
</table>
| $\text{b: block } x: \text{exp}, u: \text{oft } x \text{ nat}$ | $b.2$ concrete parameter retrieves the second component of $b$

$$g, b: \text{block } x: \text{exp}, u: \text{oft } x \text{ nat}$$
### Accessing objects in contexts

- **How do we access objects from a context?**

<table>
<thead>
<tr>
<th>Context</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b : \text{block} \ x : \text{exp}, \ u : \text{oft } x \text{ nat} )</td>
<td>( b.2 ) concrete parameter retrieves the second component of ( b )</td>
</tr>
<tr>
<td>( g, b : \text{block} \ x : \text{exp}, \ u : \text{oft } x \text{ nat} )</td>
<td>#p.2 ... parameter variable retrieves the second component of a declaration in ( g )</td>
</tr>
</tbody>
</table>
Accessing objects in contexts

- How do we access objects from a context?

<table>
<thead>
<tr>
<th>Context</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ b : \text{block} \ x : \text{exp}, \ u : \text{oft} x \text{ nat} ]</td>
<td>[ b.2 ] concrete parameter retrieves the second component of [ b ]</td>
</tr>
<tr>
<td>[ g, b : \text{block} \ x : \text{exp}, \ u : \text{oft} x \text{ nat} ]</td>
<td>[ #p.2 \ldots \text{parameter variable} ] retrieves the second component of a declaration in [ g ]</td>
</tr>
</tbody>
</table>

- Allow projections on variables and parameter variables only
## Accessing objects in contexts

- **How do we access objects from a context?**

<table>
<thead>
<tr>
<th>Context</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:block x:exp, u:oft x nat</td>
<td>b.2 concrete parameter retrieves the second component of b</td>
</tr>
<tr>
<td>g, b:block x:exp, u:oft x nat</td>
<td>#p.2 parameter variable retrieves the second component of a declaration in g</td>
</tr>
</tbody>
</table>

- **Allow projections on variables and parameter variables only**

> "Making something variable is easy. Controlling duration of constancy is the trick."

---

**Alan Perlis**
Step 2b: Proofs as Programs

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

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

Therefore:
T = S
Step 2b: Proofs as Programs

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

\[
\text{rec unique:} (g:\text{ctx}) [g.\text{oft} (M \ldots) T] \rightarrow [g.\text{oft} (M \ldots) S] \rightarrow [.\text{eq} T S] = \text{fn } d \Rightarrow \text{fn } c \Rightarrow \text{case } d \text{ of}
\]
Step 2b: Proofs as Programs

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

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

| [g.t_app (D1 \ldots) (D2 \ldots)] | ⇒ | % Application Case
| \text{let } [g.t_app (C1 \ldots) (C2 \ldots)] = c \text{ in}\n| \text{let } [.e\text{\_ref}] = \text{unique} [g.D1 \ldots] [g.C1 \ldots] \text{ in}\n| [.e\text{\_ref}] |
Step 2b: Proofs as Programs

\[
\text{rec unique:}(g:\text{ctx}) \quad [g.\text{oft}(M \ldots)T] \rightarrow [g.\text{oft}(M \ldots)S] \rightarrow [.eq T S] = \\
\text{fn} \quad d \Rightarrow \text{fn} \quad c \Rightarrow \text{case} \quad d \text{ of} \\
\quad | \quad [g.\text{t\_app}(D1 \ldots)(D2 \ldots)] \Rightarrow \quad \% \text{Application Case} \\
\quad \text{let} \quad [g.\text{t\_app}(C1 \ldots)(C2 \ldots)] = c \text{ in} \\
\quad \text{let} \quad [.e\_ref] = \text{unique} \quad [g.D1 \ldots][g.C1 \ldots] \text{ in} \\
\quad \quad [.e\_ref] \\
\quad | \quad [g.\text{t\_lam}(\lambda x.\lambda u.D \ldots x u)] \Rightarrow \quad \% \text{Abstraction Case} \\
\quad \text{let} \quad [g.\text{t\_lam}(\lambda x.\lambda u.C \ldots x u)] = c \text{ in} \\
\quad \text{let} \quad [.e\_ref] = \text{unique} \quad [g,b:\text{block} \quad x:\text{exp}, u:\text{oft} \quad x \_ \_ D \ldots b.1 \quad b.2] \\
\quad \quad [g,b \_ C \ldots b.1 \quad b.2] \text{ in} \\
\quad \quad [.e\_ref] \\
\]

Recall:
\#q: \text{block} \quad x:\text{exp}, u:\text{oft} \quad x \_ \_ T
\#r: \text{block} \quad x:\text{exp}, u:\text{oft} \quad x \_ \_ S

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

Therefore:
T = S
Step 2b: Proofs as Programs

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

fn d ⇒ fn c ⇒ case d of

| [g.t_app (D1 ...) (D2 ...)] ⇒ % Application Case
  let [g.t_app (C1 ...) (C2 ...)] = c in
  let [.e_ref] = unique [g.D1 ...] [g.C1 ...] in
  [.e_ref]

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

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

Recall:

#q: block x:exp, u:oft x T

#r: block x:exp, u:oft x S

We also know:

#r.1 = #q.1

Therefore:

T = S
Step 2b: Proofs as Programs

\[ \text{rec unique:} (g: \text{ctx}) \ [g.\text{oft} (M \ldots) T] \to [g.\text{oft} (M \ldots) S] \to [.\text{eq} T S] = \]

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

\[ | \ [g.\text{t_app} (D1 \ldots) (D2 \ldots)] \Rightarrow % \text{Application Case} \]

\[ \text{let} \ [g.\text{t_app} (C1 \ldots) (C2 \ldots)] = c \ \text{in} \]

\[ \text{let} \ [.\text{e_ref}] = \text{unique} \ [g.D1 \ldots] [g.C1 \ldots] \ \text{in} \]

\[ [.\text{e_ref}] \]

\[ | \ [g.\text{t_lam} (\lambda x.\lambda u. \ D \ldots x u)] \Rightarrow % \text{Abstraction Case} \]

\[ \text{let} \ [g.\text{t_lam} (\lambda x.\lambda u. \ C \ldots x u)] = c \ \text{in} \]

\[ \text{let} \ [.\text{e_ref}] = \text{unique} \ [g,b:\text{block} \ x:exp, u:\text{oft} x \ T . \ D \ldots b.1 b.2] \]

\[ [g,b . \ C \ldots b.1 b.2] \ \text{in} \]

\[ [.\text{e_ref}] \]

\[ | \ [g.#q.2 \ldots] \Rightarrow % \ d : \text{oft} (#q.1 \ldots) T % \text{Assumption Case} \]

\[ \text{let} \ [g.#r.2 \ldots] = c \ \text{in} % c : \text{oft} (#r.1 \ldots) S \]

\[ [.\text{e_ref}] \]

Recall:

\#q: \text{block} \ x:exp, u:\text{oft} x \ T

\#r: \text{block} \ x:exp, u:\text{oft} x \ S
Step 2b: Proofs as Programs

```
rec unique:(g:ctx) [g.oft (M ...) T] → [g.oft (M ...) S] → [.eq T S] =
fn d ⇒ fn c ⇒ case d of
| [g.t_app (D1 ...) (D2 ...)] ⇒ % Application Case
  let [g.t_app (C1 ...) (C2 ...)] = c in
  let [.e_ref] = unique [g.D1 ...] [g.C1 ...] in
  [.e_ref]

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

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

Recall:

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

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

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

fn d ⇒ fn c ⇒ case d of
  | [g.t_app (D1 ...) (D2 ...)] ⇒
    let [g.t_app (C1 ...) (C2 ...)] = c in
    let [.e_ref] = unique [g.D1 ...] [g.C1 ...] in
    [.e_ref]

  | [g.t_lam (λx.λu. D ... x u)] ⇒
    let [g.t_lam (λx.λu. C ... x u)] = c in
    let [.e_ref] = unique [g,b:block x:exp, u:oft x _ . D ... b.1 b.2] [g,b . C ... b.1 b.2] in
    [.e_ref]

  | [g.#q.2 ...] ⇒
    let [g.#r.2 ...] = c in
    [.e_ref]

Recall:

#q:block x:exp, u:oft x T
#r:block x:exp, 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</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>
</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</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</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</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</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

<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</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

<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
  - 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: Type uniqueness
- Writing a proof in Beluga . . .
- Wanting more: Programming code transformations
  - Sketching closure conversion
  - Sketching normalization by evaluation
- Conclusion
Three solitudes

Programming

- Haskell
- ATS
- Omega
- Delphin
- Twelf

General Proof assistants

- Coq
- Agda
- Isabelle

Frameworks for reasoning with HOAS
Example: Closure conversion

- Translate $\lambda$-terms such that bodies only refer to their arguments

Source language | Target language
--- | ---
$(\text{lam } y.x + y)\ 3$ | $(\text{lam } env.env.2 + env.1)\ (3, x)$
Example: Closure conversion

- Translate $\lambda$-terms such that bodies only refer to their arguments

  Source language          | Target language
  -------------------------|----------------------
  $(\text{lam } y.x + y) \ 3$ | $(\text{lam } env.env.2 + env.1) \ (3, x)$

- Challenge: Translation translates under binders
- Difficult for HOAS systems such as Twelf or Delphin
Example: Closure conversion

- Translate $\lambda$-terms such that bodies only refer to their arguments

Source language | Target language
--- | ---
$(\text{lam } y.x + y) \ 3$ | $(\text{lam } env.env.2 + env.1) (3, x)$

- Challenge: Translation translates under binders
- Difficult for HOAS systems such as Twelf or Delphin
- Programming in context in Beluga
  - Distinguish between source language $\text{tm}$ and target language $\text{ctm}$
  - Translate $[\psi.\text{tm}]$ where $\psi$ is a source context
    to $[\phi.\text{ctm}]$ where $\phi$ is a target context
Example: Closure conversion

• Translate $\lambda$-terms such that bodies only refer to their arguments

Source language | Target language
--- | ---
$(\text{lam } y.x + y) \, 3$ | $(\text{lam } env.env.2 + env.1) \, (3, \, x)$

• Challenge: Translation translates under binders

• Difficult for HOAS systems such as Twelf or Delphin

• Programming in context in Beluga
  • Distinguish between source language $\text{tm}$ and target language $\text{ctm}$
  • Translate $[\psi.\text{tm}]$ where $\psi$ is a source context
to $[\phi.\text{ctm}]$ where $\phi$ is a target context

Computation-level Type in Beluga

$$\text{rec conv : Ctx_rel } [\psi] [\phi] \rightarrow [\psi.\text{tm}] \rightarrow [\phi.\text{ctm}]$$
Example: Closure conversion

- Translate $\lambda$-terms such that bodies only refer to their arguments

  Source language                     Target language
  $(\text{lam } y.x + y) \ 3$        $\Rightarrow$  $(\text{lam } env.env.2 + env.1) \ (3, \ x)$

- Challenge: Translation translates under binders

- Difficult for HOAS systems such as Twelf or Delphin

- Programming in context in Beluga
  - Distinguish between source language $\text{tm}$ and target language $\text{ctm}$
  - Translate $[\psi.\text{tm}]$ where $\psi$ is a source context
    to $[\phi.\text{ctm}]$ where $\phi$ is a target context

**Computation-level Type in Beluga**

```plaintext
rec conv : Ctx_rel [ψ] [φ] → [ψ. tm] →[φ.ctm]
```
Indexed recursive datatype (POPL’12)

- Example: Relating source and target context

**Computation-level data types in Beluga**

```haskell
datatype Ctx_rel : {g:ctx}{h:cctx} ctype =
| Rnil : Ctx_rel [] []
| Rsnoc : Ctx_rel [g] [h]
   → Ctx_rel [g, x:tm] [h, x:ctm];
```
Indexed recursive datatype (POPL’12)

- Example: Type preserving context relation

Computation-level data types in Beluga

```haskell
datatype Ctx_trel : {g:tctx}{h:tcctx} ctype =
| Rnil : Ctx_trel [] []
| Rsnoc : Ctx_trel [g] [h] →Tp_rel [. T] [. S]
   →Ctx_trel [g, x:tm T] [h,x:ctm S] ;
```

Example: Wrapper for contextual objects.

```haskell
datatype TmVar : {g:tctx} .tp →ctype =
| TmVar : {#p:[g.tm T]} TmVar [g] [.T]
```

datatype CtxObj : {h:cctx} ctype
```haskell
| Ctx : {h:cctx} CtxObj [h] ;
```
Indexed recursive datatype (POPL’12)

- Example: Type preserving context relation

Computation-level data types in Beluga

```
datatype Ctx_trel : {g:tctx}{h:tcctx} ctype =
| Rnil : Ctx_trel [] []
| Rsnoc : Ctx_trel [g] [h] → Tp_rel [. T] [. S]
  → Ctx_trel [g, x:tm T] [h,x:ctm S]
```

- Example: Wrapper for contextual objects.

```
datatype TmVar : {g:tctx} [.tp] → ctype =
| TmVar : {#p:[g.tm T]} TmVar [g] [.T]
;

datatype CtxObj : {h:cctx} ctype =
| Ctx : {h:cctx} CtxObj [h]
```
Indexed recursive datatype (POPL’12)

- Example: Type preserving context relation

Computation-level data types in Beluga

```plaintext
datatype Ctx_trel : {g:tctx}{h:tcctx} ctype =
| Rnil : Ctx_trel [] []
| Rsnoc : Ctx_trel [g] [h] \rightarrow Tp_rel [\cdot T] [\cdot S]
  \rightarrow Ctx_trel [g, x:tm T] [h, x:ctm S]
```

- Example: Wrapper for contextual objects.

```plaintext
datatype TmVar : {g:tctx} [\cdot tp] \rightarrow ctype =
| TmVar : {#p:[g.tm T]} TmVar [g] [\cdot T]
;

datatype CtxObj : {h:cctx} ctype =
| Ctx : {h:cctx} CtxObj [h]
```

- Choice how much to push to the computation level
Replacing variables with their projections

- Traverse term in target language by pattern matching on the context

\[
\text{Guarantee that all variables have been replaced.}
\]

\[
\text{Terminates since context decreases}
\]
Replacing variables with their projections

- Traverse term in target language by pattern matching on the context
- Use built-in substitutions to replace \( x \) with its corresponding projection \( \text{proj } e \ N \) where \( e : \text{envr} \).
Replacing variables with their projections

- Traverse term in target language by pattern matching on the context

- Use built-in substitutions to replace $x$ with its corresponding projection $\text{proj } e \ N$ where $e: \text{envr}$.

- Guarantee that all variables have been replaced.
Replacing variables with their projections

- Traverse term in target language by pattern matching on the context

- Use built-in substitutions to replace $x$ with its corresponding projection $\text{proj}_e N$ where $e: \text{envr}$.

- Guarantee that all variables have been replaced.

**Computation in Beluga**

```plaintext
rec addProjs : (g:cctx) [.nat] → [g, e:envr . ctm] → [e:envr . ctm] =
fn n ⇒ fn m ⇒ case m of
  | [ e:envr . M e ] ⇒ [e:envr . M e]
  | [ g, x:ctm , e:envr . M .. x e ] ⇒
    let [.N] = n in addProjs [.s N] [g, e:envr . M .. (proj e N) e]
;```

Terminates since context decreases.
Replacing variables with their projections

- Traverse term in target language by pattern matching on the context
- Use built-in substitutions to replace \( x \) with its corresponding projection \( \text{proj} \ e \ N \) where \( e:envr \).
- Guarantee that all variables have been replaced.

Computation in Beluga

```haskell
rec addProjs : (g:cctx) [.nat] → [g, e:envr . ctm] → [e:envr . ctm] =
fn n ⇒ fn m ⇒ case m of
| [ e:envr . M e ] ⇒ [e:envr . M e]
| [ g, x:ctm , e:envr . M .. x e ] ⇒
  let [.N] = n in addProjs [.s N] [g, e:envr . M .. (proj e N) e] ;
```

- Terminates since context decreases
Converting context to environment

LF representation in Beluga

```haskell
datatype envr : type =
| nil : envr
| snoc: envr → ctm → envr

and ctm : type = ... ;
```

Computation in Beluga

```haskell
rec ctxToEnv : CtxObj [h] → [h . envr] =
fn ctx ⇒ case ctx of
| Ctx [] ⇒ [ . nil]
| Ctx [h,x:ctm] ⇒
  let [h’ . Env .. ] = ctxToEnv (Ctx [h]) in
    [h’, x:ctm . snoc (Env ..) x]
;
```

- Convert context to list
- Pattern matching on context
Example: Closure conversion

- Naive Closure conversion [Cave, Pientka’12]

- Type-preserving closure conversion [O. Savary Belanger, M. Boespflug, S. Monnier, B. Pientka]
  - Compact elegant representation
  - Only abstract over the free variables in an expression
  - Enforces also scope preservation
  - Almost proof-less

- Lessons learned:
  - Programming in context requires a new look at existing algorithms
  - Distinguishing between different context natural
  - Indexed data types are key to finding elegant solutions
This talk

Design and implementation of Beluga

- Introduction
- Example: Type uniqueness
- Writing a proof in Beluga . . .
- Wanting more: Programming code transformations
  - Sketching closure conversion
  - Sketching normalization by evaluation
- Conclusion
Normalization by evaluation

- Reuse evaluation of computation language to normalize terms in the object language [Berger, Schwichtenberg 91]

- Good benchmark
  - Twelf, Delphin are too weak (to do it directly)
  - Licata and Harper [ICFP’09] cannot express type preservation
  - Coq/Agda lack support for substitutions and binders
Normalization by evaluation

- Reuse evaluation of computation language to normalize terms in the object language [Berger, Schwichtenberg 91]

- Good benchmark
  - Twelf, Delphin are too weak (to do it directly)
  - Licata and Harper [ICFP’09] cannot express type preservation
  - Coq/Agda lack support for substitutions and binders

- General idea of NBE in Beluga

```
Source
LF objects
Lambda Terms
Non-normal

Target
Lambda Terms
beta-eta normal

Computation-level objects
Semantic representation

eval
reflect / reify
```

Evaluation is easy, normalization is hard
Normalization by evaluation

- Reuse evaluation of computation language to normalize terms in the object language [Berger, Schwichtenberg 91]

- Good benchmark
  - Twelf, Delphin are too weak (to do it directly)
  - Licata and Harper [ICFP’09] cannot express type preservation
  - Coq/Agda lack support for substitutions and binders

- General idea of NBE in Beluga

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>LF objects</td>
<td>Lambda Terms beta–eta normal</td>
</tr>
<tr>
<td>Lambda Terms Non–normal</td>
<td>Lambda Terms beta–eta normal</td>
</tr>
<tr>
<td>eval</td>
<td>reflect / reify</td>
</tr>
</tbody>
</table>

- Evaluation is easy, normalization is hard
## NBE in context

<table>
<thead>
<tr>
<th>Source of type $T$</th>
<th>Target of type $T$</th>
</tr>
</thead>
</table>
| $\Gamma \vdash T$ | $\Gamma \vdash_n T$ – Normal terms  
$\Gamma \vdash_r T$ – Neutral terms |

### Semantic Values of type $T$

$\Gamma \models T$

- **Types**: $T, S ::= T \Rightarrow S \mid i$
- **Definition of semantic values**

\[
\begin{align*}
\Gamma \models i & \equiv_{def} \Gamma \vdash_n i \\
\Gamma \models S \Rightarrow T & \equiv_{def} \forall \Gamma' \geq \Gamma. (\Gamma' \models S) \rightarrow (\Gamma' \models T)
\end{align*}
\]
NBE in context

<table>
<thead>
<tr>
<th>Source of type $T$</th>
<th>Target of type $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash T$</td>
<td>$\Gamma \vdash_{n} T$ – Normal terms</td>
</tr>
<tr>
<td></td>
<td>$\Gamma \vdash_{r} T$ – Neutral terms</td>
</tr>
</tbody>
</table>

Semantic Values of type $T$

$\Gamma \models T$

- **Types:** $T, S ::= T \Rightarrow S \mid i$
- **Definition of semantic values**

$$
\begin{align*}
\Gamma \models i & \equiv_{\text{def}} \Gamma \vdash_{n} i \\
\Gamma \models S \Rightarrow T & \equiv_{\text{def}} \forall \Gamma' \geq \Gamma. (\Gamma' \models S) \rightarrow (\Gamma' \models T)
\end{align*}
$$

Representation of syntax straightforward

- Source represented in LF using type $\text{tm } T$.
- Target represented in LF using type $\text{norm } T$ and $\text{neut } T$. 

---

Beluga: Design and implementation

B. Pientka

Beluga$\mu$: Programming proofs in context...
NBE in context

<table>
<thead>
<tr>
<th>Source of type $T$</th>
<th>Target of type $T$</th>
</tr>
</thead>
</table>
| $\Gamma \vdash T$ | $\Gamma \vdash_n T$ – Normal terms  
|                  | $\Gamma \vdash_r T$ – Neutral terms |

<table>
<thead>
<tr>
<th>Semantic Values of type $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vDash T$</td>
</tr>
</tbody>
</table>

- Types: $T, S ::= T \Rightarrow S | i$
- Definition of semantic values
  \[
  \Gamma \vdash i \equiv_{def} \Gamma \vdash_n i \\
  \Gamma \vdash S \Rightarrow T \equiv_{def} \forall \Gamma' \geq \Gamma. (\Gamma' \vDash S) \rightarrow (\Gamma' \vDash T)
  \]

Representation of syntax straightforward
- Source represented in LF using type $\text{tm} T$.
- Target represented in LF using type $\text{norm} T$ and $\text{neut} T$.

How to represent semantic values and context relations?
Defining context extensions using indexed types

- Context $g$ is a prefix of context $h$

## Computation-level data types in Beluga

```haskell
datatype Extends : {g:ctx} {h:ctx} ctype =
| Zero : Extends [g] [g]
| Succ : Extends [g] [h] → Extends [g] [h,x:neut A]
```

- Use indexed types - keyword: `ctype`
- Note: $\rightarrow$ is overloaded.
  - $tm \rightarrow tm$ is the LF function space: binders in the object language are modelled by LF functions
  - Extends $[g] [h] \rightarrow$ Extends $[g] [h,x:neut A]$ is a computation-level function
Representing target semantic values using indexed types

- Represenation of semantics using computation-level functions

\[ \Gamma \models i \quad \equiv_{\text{def}} \quad \Gamma \vdash_n i \]
\[ \Gamma \models S \Rightarrow T \quad \equiv_{\text{def}} \quad \forall \Gamma' \geq \Gamma. (\Gamma' \models S) \rightarrow (\Gamma' \models T) \]

Computation-level data types in Beluga

```
datatype Sem : \{g:ctx\} [. tp] \rightarrow ctype =
  | Syn : \{g . neut (atomic P)\} \rightarrow Sem [g] [.atomic P]
  | Slam : (\{h:ctx\} Extends [g] [h] \rightarrow Sem [h] [.S] \rightarrow Sem [h] [.T])
    \rightarrow Sem [g] [. arr S T]
;```

- Not a positive definition - we are making no claims regarding strong normalization.
Sketch of normalization by evaluation

- Define mutual recursive functions `reflect` and `reify`

```plaintext
rec reflect : [g. neut T] → Sem [g] [ .T] % Recursion on T
and reify : Sem [g] [ .T] → [g.norm T] % Recursion on T
```
Sketch of normalization by evaluation

- Define mutual recursive functions \texttt{reflect} and \texttt{reify}

\begin{verbatim}
rec reflect : [g. neut T] → Sem [g] [ .T] \% Recursion on T
and reify : Sem [g] [ .T] → [g.norm T] \% Recursion on T
\end{verbatim}

- Map between vars in the source language and their semantic values

\begin{verbatim}
datatype TmVar : {g:tctx} [.tp] → ctype =
| TmVar : {#p:[g.tm T]} TmVar [g] [ .T];
typedef Map : {g:tctx}{h:ctx} ctype = {T:[.tp]} TmVar [g] [ .T] → Sem [h] [ .T];
\end{verbatim}

- Generalized evaluation and normalization followed by reification

\begin{verbatim}
rec eval : Map [g] [h] → [g. tm S] → Sem [h] [.S] = ...
rec evaluate : [. tm S] → Sem [ ] [.S] = fn t ⇒ (eval initialMap t)
rec nbe : [. tm T] → [. norm T] = fn e ⇒ reify (evalualte e)
\end{verbatim}
Sketch of normalization by evaluation

- Define mutual recursive functions `reflect` and `reify`

```
rec reflect : [g. neut T] → Sem [g] [ .T]  % Recursion on T
and reify : Sem [g] [ .T] → [g.norm T]  % Recursion on T
```

- Map between vars in the source language and their semantic values

```
datatype TmVar : {g:tctx} [.tp] → ctype =
| TmVar : {#p:[g.tm T]} TmVar [g] [.T];
typedef Map : {g:tctx}{h:ctx} ctype = {T:[.tp]} TmVar [g] [.T] → Sem [h] [.T];
```

- Generalized evaluation and normalization followed by reification

```
rec eval : Map [g] [h] → [g. tm S] → Sem [h] [.S] = ...
rec evaluate : [. tm S] → Sem [ ] [.S] = fn t ⇒ (eval initialMap t)
rec nbe : [. tm T] → [. norm T] = fn e ⇒ reify (evaluate e)
```

- Almost a consistency proof! Currently no termination or positivity checking.
What have we achieved?

- Revised foundation for programming with contexts and contextual LF (joint work with A. Cave [POPL’12])
- 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
- Effectively write programs to manipulate rich abstract syntax trees and express properties about them
- Release in Sept’12: Support for indexed data types; coverage; type reconstruction; environment-based interpreter; support for holes (partial programs)

Result:

Compact and elegant programming (with) inductive proofs in context
Current work

- Prototype in OCaml (ongoing)
- Extension to coinduction (D. Thibodeau, A. Abel)
- Termination checking (C. Badescu)
- Mixing computations in computation-level types (A. Cave)
- Case study: Certified compiler (O. Savary Belanger)
- Compiling contexts and contextual objects (F. Ferreira)
Thank you!

Download prototype and examples at

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

Current Belugians: Brigitte Pientka, Mathieu Boespflug, Costin Badescu, Olivier Savary Belanger, Andrew Cave, Francisco Ferreira, Stefan Monnier, David Thibodeau

Interested? - Talk to me! We have funded postdoc and funded PhD positions.