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

$$\text{Types } T \quad ::= \quad \mathsf{nat} \mid \mathsf{arr}\, T_1 \; T_2$$

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

Simply typed lambda-calculus

Types and Terms

$$\begin{array}{lcl} \text{Types } T & ::= & \mathsf{nat} \\ & & \mid \mathsf{arr}\, T_1\ T_2 \end{array} \qquad \begin{array}{lcl} \text{Terms M} & ::= & x \\ & & \mid \mathsf{lam}\ x{:}T.M \\ & & \mid \mathsf{app}\ M\ N \end{array}$$

Typing Judgment: $\mathsf{oft}\ M\ T$ read as "M has type T"

Simply typed lambda-calculus

Types and Terms

$$\text{Types } T \quad ::= \quad \text{nat} \mid \text{arr}\, T_1 \; T_2 \qquad\qquad \text{Terms M} \quad ::= \quad x \mid \text{lam } x{:}T.M \mid \text{app } M \; N$$

Typing Judgment: oft M T read as "M has type T"

Typing rules (Gentzen-style, context-free)

$$\dfrac{\begin{array}{c}\overline{\text{oft } x \; T}^{\;u} \\ \vdots \\ \text{oft } M \; S\end{array}}{\text{oft } (\text{lam}\, x{:}T.M) \; (\text{arr } T \; S)} \; \text{t\_lam}^{x,u}$$

Simply typed lambda-calculus

Types and Terms

$$\text{Types } T \quad ::= \quad \text{nat} \mid \text{arr}\, T_1 \;\, T_2 \qquad\qquad \text{Terms M} \quad ::= \quad x \mid \text{lam}\; x{:}T.M \mid \text{app}\; M\; N$$

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

Typing rules (Gentzen-style, context-free)

$$\dfrac{\begin{matrix}\overline{\text{oft}\ x\ \ T}^{\;u} \\ \vdots \\ \text{oft}\ M\ S\end{matrix}}{\text{oft}\ (\text{lam}\, x{:}T.M)\ (\text{arr}\ \ T\ \ S)}\ \text{t\_lam}^{x,u} \qquad \dfrac{\text{oft}\ M\ (\text{arr}\ \ T\ \ S) \qquad \text{oft}\ N\ \ T}{\text{oft}\ (\text{app}\ M\ N)\ S}\ \text{t\_app}$$

Simply typed lambda-calculus

Types and Terms

$$\text{Types } T \quad ::= \quad \text{nat} \mid \text{arr}\, T_1 \; T_2 \qquad\qquad \text{Terms M} \quad ::= \quad x \mid \text{lam } x{:}T.M \mid \text{app } M \; N$$

Typing Judgment: $\quad \text{oft } M \; T \qquad$ read as "M has type T"

Typing rules (Gentzen-style, context-free)

$$\dfrac{\begin{array}{c}\overline{\text{oft } x \; T}\;u \\ \vdots \\ \text{oft } M \; S\end{array}}{\text{oft } (\text{lam}\, x{:}T.M) \; (\text{arr } T \; S)}\;\text{t\_lam}^{x,u} \qquad \dfrac{\text{oft } M \; (\text{arr } T \; S) \quad \text{oft } N \; T}{\text{oft } (\text{app } M \; N) \; S}\;\text{t\_app}$$

$\text{Context } \Gamma \quad ::= \quad \cdot \mid \Gamma, x, \; \text{oft } x \; T$ 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 | lam $x{:}T.M$ | app M N

Typing Judgment: $\Gamma \vdash$ oft M T read as "M has type T in context Γ"

Typing rules

$$\frac{x, u : \text{ oft } x\ T \in \Gamma}{\Gamma \vdash \text{ oft } x\ T}\ u$$

$$\frac{\Gamma, x, u : \text{ oft } x\ T \vdash \text{oft } M\ S}{\Gamma \vdash \text{oft } (\text{lam}\, x{:}T.M)\ (\text{arr } T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft } M\ (\text{arr } T\ S) \quad \Gamma \vdash \text{oft } N\ T}{\Gamma \vdash \text{oft } (\text{app } M\ N)\ S}\ \text{t\_app}$$

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

Talking about derivations

Typing rules

$$\frac{x, u:\ \mathsf{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ T}\ u$$

$$\frac{\Gamma, x, u:\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\, x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \mathsf{oft}\ M\ (\mathsf{arr}\ T\ S) \quad \Gamma \vdash \mathsf{oft}\ N\ T}{\Gamma \vdash \mathsf{oft}\ (\mathsf{app}\ M\ N)\ S}\ \mathsf{t\_app}$$

Talking about derivations

Typing rules

$$\frac{x, u : \text{ oft } x\ T \in \Gamma}{\Gamma \vdash \text{ oft } x\ T}\ u$$

$$\frac{\Gamma, x, u : \text{ oft } x\ T \vdash \text{oft } M\ S}{\Gamma \vdash \text{oft } (\text{lam}\, x{:}T.M)\ (\text{arr } T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft } M\ (\text{arr } T\ S) \quad \Gamma \vdash \text{oft } N\ T}{\Gamma \vdash \text{oft } (\text{app } M\ N)\ S}\ \text{t\_app}$$

- What kinds of variables are used?

Talking about derivations

Typing rules

$$\frac{x, u : \text{ oft } x\ T \in \Gamma}{\Gamma \vdash \text{ oft } x\ T}\ u$$

$$\frac{\Gamma, x, u : \text{ oft } x\ T \vdash \text{oft } M\ S}{\Gamma \vdash \text{oft } (\text{lam}\, x{:}T.M)\ (\text{arr } T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft } M\ (\text{arr } T\ S) \quad \Gamma \vdash \text{oft } N\ T}{\Gamma \vdash \text{oft } (\text{app } M\ N)\ S}\ \text{t\_app}$$

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

Talking about derivations

Typing rules

$$\frac{x, u:\ \text{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \text{oft}\ x\ T}\ u$$

$$\frac{\Gamma, x, u:\ \text{oft}\ x\ T \vdash \text{oft}\ M\ S}{\Gamma \vdash \text{oft}\ (\text{lam}\, x{:}T.M)\ (\text{arr}\ T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft}\ M\ (\text{arr}\ T\ S) \quad \Gamma \vdash \text{oft}\ N\ T}{\Gamma \vdash \text{oft}\ (\text{app}\ M\ N)\ S}\ \text{t\_app}$$

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

Talking about derivations

Typing rules

$$\frac{x, u:\ \text{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \text{oft}\ x\ T}\ u$$

$$\frac{\Gamma, x, u:\ \text{oft}\ x\ T \vdash \text{oft}\ M\ S}{\Gamma \vdash \text{oft}\ (\text{lam}\, x{:}T.M)\ (\text{arr}\ T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft}\ M\ (\text{arr}\ T\ S) \quad \Gamma \vdash \text{oft}\ N\ T}{\Gamma \vdash \text{oft}\ (\text{app}\ M\ N)\ S}\ \text{t\_app}$$

- 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

$$\frac{x, u:\ \text{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \text{oft}\ x\ T}\ u$$

$$\frac{\Gamma, x, u:\ \text{oft}\ x\ T \vdash \text{oft}\ M\ S}{\Gamma \vdash \text{oft}\ (\text{lam}\, x{:}T.M)\ (\text{arr}\ T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft}\ M\ (\text{arr}\ T\ S) \quad \Gamma \vdash \text{oft}\ N\ T}{\Gamma \vdash \text{oft}\ (\text{app}\ M\ N)\ S}\ \text{t\_app}$$

- 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
- What properties do contexts have?

Talking about derivations

Typing rules

$$\frac{x, u:\ \text{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \text{oft}\ x\ T}\ u$$

$$\frac{\Gamma, x, u:\ \text{oft}\ x\ T \vdash \text{oft}\ M\ S}{\Gamma \vdash \text{oft}\ (\text{lam}\, x{:}T.M)\ (\text{arr}\ T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft}\ M\ (\text{arr}\ T\ S) \quad \Gamma \vdash \text{oft}\ N\ T}{\Gamma \vdash \text{oft}\ (\text{app}\ M\ N)\ S}\ \text{t\_app}$$

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

Talking about derivations

Typing rules

$$\frac{x, u : \text{ oft } x\ T \in \Gamma}{\Gamma \vdash \text{ oft } x\ T}\ u$$

$$\frac{\Gamma, x, u : \text{ oft } x\ T \vdash \text{oft } M\ S}{\Gamma \vdash \text{oft } (\text{lam}\, x{:}T.M)\ (\text{arr } T\ S)}\ \text{t\_lam}^{x,u} \qquad \frac{\Gamma \vdash \text{oft } M\ (\text{arr } T\ S) \quad \Gamma \vdash \text{oft } N\ T}{\Gamma \vdash \text{oft } (\text{app } M\ N)\ S}\ \text{t\_app}$$

- 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
- 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.

Type uniqueness

Theorem

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

Type uniqueness

Theorem

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

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

Case 1

$$\mathcal{D} = \cfrac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x, u\text{: oft } x\ T \vdash \text{oft } M\ S\end{array}}{\Gamma \vdash \text{oft (lam } x{:}T.M) \text{ (arr } T\ S)}\ \text{t\_lam} \quad \mathcal{C} = \cfrac{\begin{array}{c}\mathcal{C}_1\\ \Gamma, x, u\text{: oft } x\ T \vdash \text{oft } M\ S'\end{array}}{\Gamma \vdash \text{oft (lam } x{:}T.M) \text{ (arr } T\ S')}\ \text{t\_lam}$$

Type uniqueness

Theorem

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

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

Case 1

$$\mathcal{D} = \cfrac{\overset{\mathcal{D}_1}{\Gamma, x, u{:}\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam} \qquad \mathcal{C} = \cfrac{\overset{\mathcal{C}_1}{\Gamma, x, u{:}\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S'}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S')}\ \mathsf{t\_lam}$$

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

Type uniqueness

Theorem

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

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

Case 1

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1 \\ \Gamma, x, u{:}\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S\end{array}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam} \qquad \mathcal{C} = \frac{\begin{array}{c}\mathcal{C}_1 \\ \Gamma, x, u{:}\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S'\end{array}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S')}\ \mathsf{t\_lam}$$

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

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

Type uniqueness

Theorem

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

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

Case 1

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x, u\text{: oft } x\ T \vdash \mathsf{oft}\ M\ S}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam} \qquad \mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma, x, u\text{: oft } x\ T \vdash \mathsf{oft}\ M\ S'}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S')}\ \mathsf{t\_lam}$$

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

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

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

$$\mathcal{D} = \frac{\begin{array}{c}\mathcal{D}_1\\ \Gamma, x, u\text{: oft } x\ T \vdash \text{oft}\ M\ S\end{array}}{\Gamma \vdash \text{oft (lam } x{:}T.M)\ (\text{arr}\ T\ S)}\ \text{t\_lam} \qquad \mathcal{C} = \frac{\begin{array}{c}\mathcal{C}_1\\ \Gamma, x, u\text{: oft } x\ T \vdash \text{oft}\ M\ S'\end{array}}{\Gamma \vdash \text{oft (lam } x{:}T.M)\ (\text{arr}\ T\ S')}\ \text{t\_lam}$$

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

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

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

Case 2

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

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

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x, u\colon \text{oft } x\ T \vdash \text{oft } M\ S}}{\Gamma \vdash \text{oft (lam } x{:}T.M) \text{ (arr } T\ S)}\ \text{t\_lam} \qquad \mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma, x, u\colon \text{oft } x\ T \vdash \text{oft } M\ S'}}{\Gamma \vdash \text{oft (lam } x{:}T.M) \text{ (arr } T\ S')}\ \text{t\_lam}$$

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

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

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

Case 2

$$\mathcal{D} = \dfrac{x, u\colon \text{oft } x\ T \in \Gamma}{\Gamma \vdash \text{oft } x\ T}\ u \qquad \mathcal{C} = \dfrac{x, v\colon \text{oft } x\ S \in \Gamma}{\Gamma \vdash \text{oft } x\ S}\ v$$

Type uniqueness

Theorem

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

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

Case 1

$$\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma, x, u{:}\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S)}\ \mathsf{t\_lam} \qquad \mathcal{C} = \dfrac{\overset{\mathcal{C}_1}{\Gamma, x, u{:}\ \mathsf{oft}\ x\ T \vdash \mathsf{oft}\ M\ S'}}{\Gamma \vdash \mathsf{oft}\ (\mathsf{lam}\ x{:}T.M)\ (\mathsf{arr}\ T\ S')}\ \mathsf{t\_lam}$$

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

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

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

Case 2

$$\mathcal{D} = \dfrac{x, u{:}\ \mathsf{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ T}\ u \qquad \mathcal{C} = \dfrac{x, v{:}\ \mathsf{oft}\ x\ S \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ S}\ v$$

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 α-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 α-renaming, substitution, adequate representations

Programming proofs [Pientka'08, Pientka,Dunfield'10, Cave,Pientka'12]

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

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 α-renaming, substitution, adequate representations

Programming proofs [Pientka'08, Pientka,Dunfield'10, Cave,Pientka'12]

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

- Contextual types characterize contextual objects [NPP'08]
  $\rightsquigarrow$ 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
  $\rightsquigarrow$ support for α-renaming, substitution, adequate representations

Programming proofs [Pientka'08, Pientka,Dunfield'10, Cave,Pientka'12]

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

- Contextual types characterize contextual objects [NPP'08]
  $\rightsquigarrow$ support well-scoped derivations
- Context variables parameterize computations
  $\rightsquigarrow$ 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 α-renaming, substitution, adequate representations

Programming proofs [Pientka'08, Pientka,Dunfield'10, Cave,Pientka'12]

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

- 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

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

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

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

```
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

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

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

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

$$\frac{\text{oft } M \text{ (arr } T\ S) \quad \text{oft } N\ T}{\text{oft (app } M\ N)\ S}\ \text{t\_app}$$

$$\frac{\begin{array}{c}\overline{\text{oft } x\ T}^{\,u} \\ \vdots \\ \text{oft } M\ S\end{array}}{\text{oft (lam } x{:}T.M) \text{ (arr } T\ S)}\ \text{t\_lam}^{x,u}$$

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

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

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

Typing rules

$$\frac{\text{oft } M \text{ (arr } T \; S) \quad \text{oft } N \; T}{\text{oft (app } M \; N) \; S} \text{ t\_app}$$

$$\frac{\begin{array}{c}\overline{\text{oft } x \; T}^{\,u} \\ \vdots \\ \text{oft } M \; S\end{array}}{\text{oft (lam } x{:}T.M) \text{ (arr } T \; S)} \text{ t\_lam}^{x,u}$$

```
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

Step 2a: Theorem as type

Theorem

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

Step 2a: Theorem as type

Theorem

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

is represented as

Computation-level Type in Beluga

```
(g:ctx) [g.oft (M …) T] → [g.oft (M …) S] → [ .eq T S]
```

Read as: "For all contexts `g` of the schema `ctx`, ...

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 $\mathcal{E} : \text{eq } T \; S$.

is represented as

Computation-level Type in Beluga

```
(g:ctx) [g.oft (M …) T] → [g.oft (M …) S] → [ .eq T S]
```

Read as: "For all contexts `g` of the schema `ctx`, ...

- `[g.oft (M …) T]` and `[ .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

Computation-level Type in Beluga

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

- Parameterize computation over contexts, Distinguish between contexts.

Intrinsic support for contexts

Computation-level Type in Beluga

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

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

Intrinsic support for contexts

Computation-level Type in Beluga

(g:ctx) [g.oft (M ...) T] → [g.oft (M ...) S] → [.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.

Intrinsic support for contexts

Computation-level Type in Beluga

```
(g:ctx) [g.oft (M …) T] → [g.oft (M …) S] → [.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
  `b1:`**block** `x:exp, u:oft x nat, b2:`**block** `y:exp,v:oft y (arr nat nat) .`

Intrinsic support for contexts

Computation-level Type in Beluga

```
(g:ctx) [g.oft (M …) T] → [g.oft (M …) S] → [.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
  `b1:`**block** `x:exp, u:oft x nat, b2:`**block** `y:exp,v:oft y (arr nat nat) .`
- Well-formedness: `b1:`**block** `x:exp,u:oft y nat` is ill-formed.
  `x:exp, y:exp, u:oft x nat` is ill-formed.

Intrinsic support for contexts

Computation-level Type in Beluga

`(g:ctx) [g.oft (M …) T] → [g.oft (M …) S] → [.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
  `b1:`**block** `x:exp, u:oft x nat, b2:`**block** `y:exp,v:oft y (arr nat nat) .`
- 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`

Intrinsic support for contexts

Computation-level Type in Beluga

`(g:ctx) [g.oft (M ...) T] → [g.oft (M ...) S] → [.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
  `b1:`**block** `x:exp, u:oft x nat, b2:`**block** `y:exp,v:oft y (arr nat nat) .`
- 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`

Intrinsic support for contexts

Computation-level Type in Beluga

`(g:ctx) [g.oft (M …) T] → [g.oft (M …) S] → [.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
  `b1:`**block** `x:exp, u:oft x nat, b2:`**block** `y:exp,v:oft y (arr nat nat) .`
- 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 ...) T] → [g.oft (M ...) S] → [.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
  `b1:`**block** `x:exp, u:oft x nat, b2:`**block** `y:exp,v:oft y (arr nat nat) .`
- 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
- 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?

Context	Element
`b:`**block** `x:exp, u:oft x nat`	`b.2` concrete parameter retrieves the second component of `b`

Accessing objects in contexts

- How do we access objects from a context?

Context	Element
`b:`**block** `x:exp, u:oft x nat`	`b.2` concrete parameter retrieves the second component of `b`

`g, b:`**block** `x:exp, u:oft x nat`

Accessing objects in contexts

- How do we access objects from a context?

Context	Element
`b:`**block** `x:exp, u:oft x nat`	`b.2` concrete parameter retrieves the second component of `b`
`g, b:`**block** `x:exp, u:oft x nat`	`#p.2` … parameter variable retrieves the second component of a declaration in `g`

Accessing objects in contexts

- How do we access objects from a context?

Context	Element
`b:`**block** `x:exp, u:oft x nat`	`b.2` concrete parameter retrieves the second component of `b`
`g, b:`**block** `x:exp, u:oft x nat`	`#p.2 …` parameter variable retrieves the second component of a declaration in `g`

- Allow projections on variables and parameter variables only

Accessing objects in contexts

- How do we access objects from a context?

Context	Element
`b:`**block** `x:exp, u:oft x nat`	`b.2` concrete parameter retrieves the second component of `b`
`g, b:`**block** `x:exp, u:oft x nat`	`#p.2` ... parameter variable retrieves the second component of a declaration in `g`

- 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

Step 2b: Proofs as Programs

rec `unique:(g:ctx) [g.oft (M …) T]` $\rightarrow$ `[g.oft (M …) S]` $\rightarrow$ `[.eq 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
```

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]
```

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]
```

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] ;
```

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] ;
```

Recalll:

```
#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 ...)] ⇒                      % 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] ;
```

Recalll:

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

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

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] ;
```

Recalll:

`#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

On paper proof	Implementation in Beluga
Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF

Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

On paper proof	Implementation in Beluga
Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF
Well-scoped derivation	Contextual types and objects

Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

On paper proof	Implementation in Beluga
Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF
Well-scoped derivation	Contextual types and objects
Context	Context schemas

Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

On paper proof	Implementation in Beluga
Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF
Well-scoped derivation	Contextual types and objects
Context	Context schemas
Properties of contexts (weakening, uniqueness)	Typing for schemas

Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

On paper proof	Implementation in Beluga
Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF
Well-scoped derivation	Contextual types and objects
Context	Context schemas
Properties of contexts (weakening, uniqueness)	Typing for schemas

- Compact representation of proofs as functions

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

Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

On paper proof	Implementation in Beluga
Well-formed derivations	Dependent types
Renaming,Substitution	α-renaming, β-reduction in LF
Well-scoped derivation	Contextual types and objects
Context	Context schemas
Properties of contexts (weakening, uniqueness)	Typing for schemas

- Compact representation of proofs as functions

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

Comparison

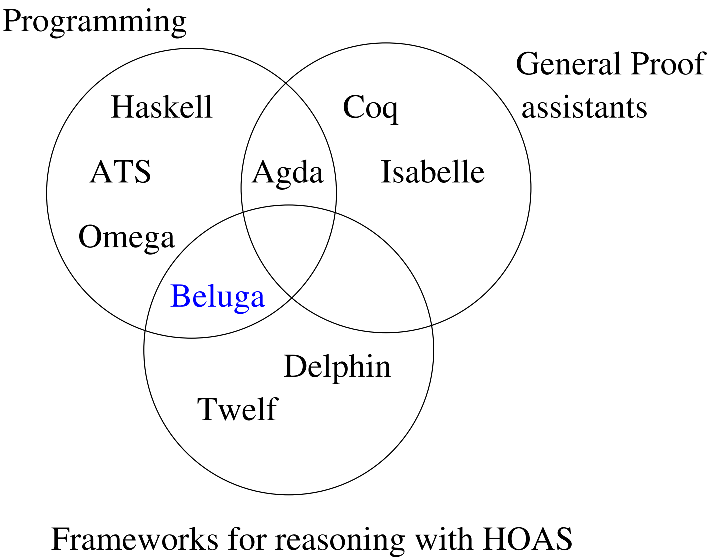
- Twelf [Pf,Sch'99]: Encode proofs as relations
  - Requires lemma to prove injectivity of `arr` constructor.
  - No explicit contexts (cannot express types `T` and `S` and `eq T S` are closed)
  - Parameter case folded into abstraction case
- Delphin [Sch,Pos'08]: Encode proofs as functions
  - Requires 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



Example: Closure conversion

- Translate λ-terms such that bodies only refer to their arguments

Source language		Target language
(lam $y.x + y$) 3	$\Longrightarrow$	(lam $env.env.2 + env.1$) $(3\,,\,x)$

Example: Closure conversion

- Translate λ-terms such that bodies only refer to their arguments

Source language		Target language
$(\mathrm{lam}\ y.x + y)\ 3$	$\Longrightarrow$	$(\mathrm{lam}\ \mathit{env}.\mathit{env}.2 + \mathit{env}.1)\ (3\ ,\ x)$

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

Example: Closure conversion

- Translate λ-terms such that bodies only refer to their arguments

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

- Challenge: Translation translates under binders
- Difficult for HOAS systems such as Twelf or Delphin
- Programming in context in Beluga
  - Distinguish between source language `tm` and target language `ctm`
  - Translate `[`ψ`.tm]` where ψ is a source context
    to `[`ϕ`.ctm]` where ϕ is a target context

Example: Closure conversion

- Translate λ-terms such that bodies only refer to their arguments

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

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

Computation-level Type in Beluga

$\textbf{rec}\ \ \texttt{conv}\ :\texttt{Ctx\_rel}\ [\psi]\ [\phi] \rightarrow [\psi.\ \texttt{tm}] \rightarrow [\phi.\texttt{ctm}]$

Example: Closure conversion

- Translate λ-terms such that bodies only refer to their arguments

Source language — Target language

$$(\text{lam } y.x + y)\ 3 \quad \Longrightarrow \quad (\text{lam } env.env.2 + env.1)\ (3\,,\ x)$$

- Challenge: Translation translates under binders
- Difficult for HOAS systems such as Twelf or Delphin
- Programming in context in Beluga
  - Distinguish between source language `tm` and target language `ctm`
  - Translate `[`ψ`.tm]` where ψ is a source context
    to `[`ϕ`.ctm]` where ϕ is a target context

Computation-level Type in Beluga

rec `conv :Ctx_rel` $[\psi]$ $[\phi]$ $\to$ `[`ψ`. tm]` $\to$`[`ϕ`.ctm]`

Indexed recursive datatype (POPL'12)

- Example: Relating source and target context

Computation-level data types in Beluga

```
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

```
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] ;
```

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

```
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] ;
```

- 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

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 `proj e N` where `e: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 `proj e N` where `e: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 `proj e N` where `e:envr`.
- Guarantee that **all** variables have been replaced.

Computation in Beluga

```
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]
;
```

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 `proj e N` where `e:envr`.
- Guarantee that **all** variables have been replaced.

Computation in Beluga

```
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

```
 datatype envr: type =
| nil : envr
| snoc: envr → ctm → envr
and ctm : type =   ...  ;
```

Computation in Beluga

```
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

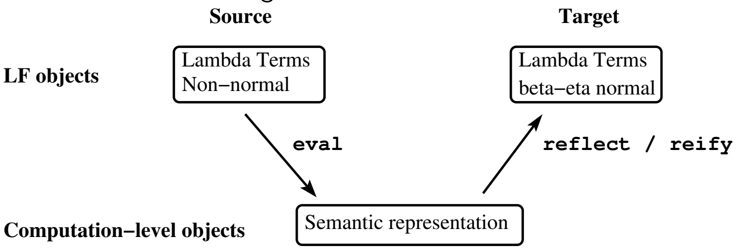
- Introduction
- Example: Type uniqueness
- Writing a proof in Beluga ...
- Wanting more: Programming code transformations
  - Sketching closure conversion
  - Sketching normalization by evaluation
- Conclusion

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

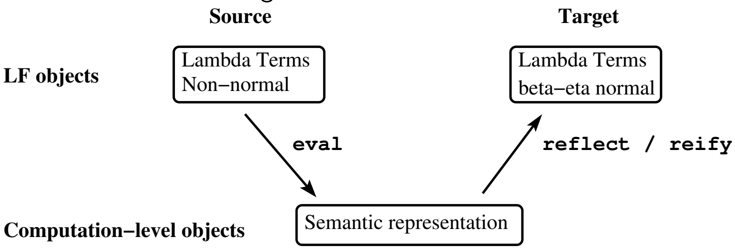


Normalization by evaluation

- 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



- Evaluation is easy, normalization is hard

NBE in context

Source of type T	Target of type T
$\Gamma \vdash T$	$\Gamma \vdash_n T$ – Normal terms $\Gamma \vdash_r T$ – Neutral terms
Semantic Values of type T $\Gamma \vDash T$	

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

$$\begin{array}{rcl} \Gamma \vDash i & \equiv_{def} & \Gamma \vdash_n i \\ \Gamma \vDash S \Rightarrow T & \equiv_{def} & \forall \Gamma' \geq \Gamma.\ (\Gamma' \vDash S) \rightarrow (\Gamma' \vDash T) \end{array}$$

NBE in context

Source of type T	Target of type T
$\Gamma \vdash T$	$\Gamma \vdash_n T$ – Normal terms $\Gamma \vdash_r T$ – Neutral terms
Semantic Values of type T $\Gamma \vDash T$	

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

$$
\begin{array}{rcl}
\Gamma \vDash i & \equiv_{def} & \Gamma \vdash_n i \\
\Gamma \vDash S \Rightarrow T & \equiv_{def} & \forall \Gamma' \geq \Gamma.\ (\Gamma' \vDash S) \rightarrow (\Gamma' \vDash T)
\end{array}
$$

Representation of syntax straightforward

- Source represented in LF using type `tm T`.
- Target represented in LF using type `norm T` and `neut T`.

NBE in context

Source of type T	Target of type T
$\Gamma \vdash T$	$\Gamma \vdash_n T$ – Normal terms $\Gamma \vdash_r T$ – Neutral terms
Semantic Values of type T $\Gamma \vDash T$	

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

$$
\begin{array}{rcl}
\Gamma \vDash i & \equiv_{def} & \Gamma \vdash_n i \\
\Gamma \vDash S \Rightarrow T & \equiv_{def} & \forall \Gamma' \geq \Gamma.\ (\Gamma' \vDash S) \rightarrow (\Gamma' \vDash T)
\end{array}
$$

Representation of syntax straightforward

- Source represented in LF using type `tm T`.
- Target represented in LF using type `norm T` and `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

```
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: → is overloaded.
  - `tm → tm` is the LF function space : binders in the object language are modelled by LF functions
  - `Extends [g] [h] → 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

$$
\begin{array}{rcl}
\Gamma \vDash i & \equiv_{def} & \Gamma \vdash_n i \\
\Gamma \vDash S \Rightarrow T & \equiv_{def} & \forall \Gamma' \geq \Gamma.\ (\Gamma' \vDash S) \rightarrow (\Gamma' \vDash T)
\end{array}
$$

Computation-level data types in Beluga

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

```
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 `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 (evalualte e)
```

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 (evalualte 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
  $\Rightarrow$ 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)

The end

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.