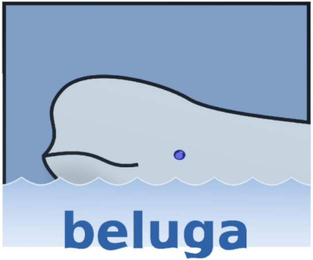
Beluga$^{\mu}$: Programming proofs in context ...

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

"The limits of my language mean the limits of my world."
- L. Wittgenstein

This talk

Design and implementation of Beluga

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

"The limits of my language mean the limits of my world."

- L. Wittgenstein

Simply typed lambda-calculus

Types and Terms

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

Simply typed lambda-calculus

Types and Terms

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

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

Simply typed lambda-calculus

Types and Terms

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

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

Typing rules (Gentzen-style, context-free)

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

Simply typed lambda-calculus

Types and Terms

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

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

Typing rules (Gentzen-style, context-free)

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

Simply typed lambda-calculus

Types and Terms

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

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

Typing rules (Gentzen-style, context-free)

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

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

Simply typed lambda-calculus

Types and Terms

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

Typing Judgment: $\Gamma \vdash \text{oft } M\ T$ read as "M has type T in context 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 $\Gamma \;::=\; \cdot \mid \Gamma, x,\ \text{oft } x\ T$ 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

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

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

- 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

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

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

- What kinds of variables are used? Bound variables, Schematic variables in particular:Meta-variables, Parameter variables, Context variables
- What operations on variables are needed? Substitution for bound variable, Renaming of bound variables, Substitution for schematic variables
- How should we represent contexts? What properties do contexts have?

Talking about derivations

Typing rules

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

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

- What kinds of variables are used? Bound variables, Schematic variables in particular:Meta-variables, Parameter variables, Context variables
- What operations on variables are needed? Substitution for bound variable, Renaming of bound variables, Substitution for schematic variables
- How should we represent contexts? What properties do contexts have? (Structured) Sequences, Every declaration is unique, weakening, substitution lemma, etc.

Talking about derivations

Typing rules

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

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

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

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

Type uniqueness

Theorem

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

Type uniqueness

Theorem

If $\mathcal{D} : \Gamma \vdash \mathsf{oft}\ M\ T$ 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

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

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

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

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

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

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

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

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

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

$\mathcal{E} :\ \text{eq } S\ S'$ 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 eq (arr $T\ S$) (arr $T\ S'$) by 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

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

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

$$\mathcal{D} = \dfrac{x, u\!:\ \mathsf{oft}\ x\ T \in \Gamma}{\Gamma \vdash\ \mathsf{oft}\ x\ T}\ u$$

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

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

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

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

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

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

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

Beluga$^\mu$: two level approach

Logical framework LF [HHP'93]

- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types

Beluga$^\mu$: two level approach

Logical framework LF [HHP'93]

- Compact representation of formal systems and derivations
- Higher-order abstract syntax and dependent types
  $\leadsto$ support for 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]

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

- Contextual LF: Contextual types characterize contextual objects [NPP'08]
  $\rightsquigarrow$ support well-scoped derivations
  $\rightsquigarrow$ abstract notion of contexts and substitution

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]

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

- Contextual LF: Contextual types characterize contextual objects [NPP'08]
  $\leadsto$ support well-scoped derivations
  $\leadsto$ abstract notion of contexts and substitution
- Recursive definitions = Indexed Recursive Types [Cave,Pientka'12]

Beluga$^\mu$: two level approach

Logical framework LF [HHP'93]

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

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

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

- Contextual LF: Contextual types characterize contextual objects [NPP'08]
  $\rightsquigarrow$ support well-scoped derivations
  $\rightsquigarrow$ abstract notion of contexts and substitution
- Recursive definitions = Indexed Recursive Types [Cave,Pientka'12]

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

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

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

Step 1: Represent types and lambda-terms in LF

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

LF representation in Beluga

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

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

Typing rules

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

Step 1: Represent types and lambda-terms in LF

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

LF representation in Beluga

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

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

Typing rules

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

```
datatype oft: tm → tp → type =
| t_app: oft M (arr T S) → oft N T
    → oft (app M N) S
```

```
| t_lam: (Π x:tm.oft x T → oft (M x) S)
    → oft (lam T M) (arr T S);
```

Step 2a: Theorem as type

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

```
(Γ:ctx) [Γ ⊢ oft (M …) T] → [Γ ⊢ oft (M …) S] → [ ⊢ eq T S]
```

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

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

`(Γ:ctx) [Γ ⊢ oft (M …) T] → [Γ ⊢ oft (M …) S] → [ ⊢ eq T S]`

Read as: "For all contexts Γ of the schema `ctx`, …

- `[Γ ⊢ 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 Γ.
- Contexts are structured sequences and are classified by context schemas

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

```
(Γ:ctx) [Γ ⊢ oft (M …) T] → [Γ ⊢ oft (M …) S] → [ ⊢ eq T S]
```

Read as: "For all contexts Γ of the schema `ctx`, …

- `[Γ ⊢ 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 Γ.
- Contexts are structured sequences and are classified by context schemas
  schema `ctx = some [T:tp]` **block** `x:tm, u:oft x T.`

Step 2b: Proofs as Programs

Step 2b: Proofs as Programs

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

Step 2b: Proofs as Programs

```
rec unique:(Γ:ctx)[Γ ⊢oft (M …)T] →[Γ ⊢oft (M …)S] →[ ⊢eq T S] =
fn d ⇒ fn c ⇒ case d of
```

Step 2b: Proofs as Programs

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

fn  d  ⇒  fn  c  ⇒  case  d  of

| [Γ ⊢ t_app  (D1 …)  (D2 …)]  ⇒                    % Application Case
  let   [Γ ⊢ t_app  (C1 …)  (C2 …)]  = c  in
  let   [  ⊢  e_ref]  = unique  [Γ ⊢ D1 …]  [Γ ⊢ C1 …]  in
    [ ⊢  e_ref]
```

Step 2b: Proofs as Programs

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

fn d ⇒ fn c ⇒ case d of

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


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

Step 2b: Proofs as Programs

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

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

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

Step 2b: Proofs as Programs

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

fn  d  ⇒  fn  c  ⇒  case  d  of

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

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

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

Recalll:

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

Step 2b: Proofs as Programs

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

fn d ⇒ fn c ⇒ case d of

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


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


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

Recalll:

`#q:`**`block`**` x:tm, u:oft x T`

`#r:`**`block`**` x:tm, u:oft x S`

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

Step 2b: Proofs as Programs

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

fn d ⇒ fn c ⇒ case d of

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

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

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

Recalll:

`#q:`**`block`**` x:tm, u:oft x T`

`#r:`**`block`**` x:tm, u:oft x S`

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

Therefore: `T` = `S`

Revisiting the design of Beluga

- Compact adequate representation of derivations and contexts

On paper proof	Implementation in Beluga [IJCAR'10]
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 [IJCAR'10]
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 [IJCAR'10]
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 [IJCAR'10]
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 [IJCAR'10]
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 [POPL'08,PPDP08]

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 [IJCAR'10]
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 [POPL'08,PPDP08]

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 based on proof theory
    - Equality built-into the logic
    - Contexts are represented as lists
    - Requires lemmas about these lists (for example that all assumptions occur uniquely)

This talk

Design and implementation of Beluga

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

Example: Evaluator using closures

- Lambda-terms and closures

$$\begin{array}{llll} \text{Terms} & M, N & := & x \mid \lambda x.M \mid M\ N \\ \text{Closures} & C & := & \mathsf{Cl}(x.M, \rho) \\ \text{Environment} & \rho & := & \cdot \mid \rho,\ (x, C) \end{array}$$

- Meaning of $\mathsf{Cl}(x.M, \rho)$: ρ provides instantiations for all the free variables in $x.M$.
- Environment ρ is a mapping from variables to closures

Example: Evaluator using closures

- Lambda-terms and closures

$$\begin{array}{llcl} \text{Terms} & M, N & := & x \mid \lambda x.M \mid M\ N \\ \text{Closures} & C & := & \mathsf{Cl}(x.M, \rho) \\ \text{Environment} & \rho & := & \cdot \mid \rho,\ (x, C) \end{array}$$

- Meaning of $\mathsf{Cl}(x.M, \rho)$: ρ provides instantiations for all the free variables in $x.M$.
- Environment ρ is a mapping from variables to closures
- Evaluation : $(M,\ \rho) \Downarrow C$

$$\frac{\text{lookup } x\ \rho = C}{(x\ ,\ \rho) \Downarrow C} \qquad \frac{}{(\lambda x.M\ ,\ \rho) \Downarrow \mathsf{Cl}(x.M,\ \rho)}$$

$$\frac{(M_1,\ \rho) \Downarrow \mathsf{Cl}(x.N,\ \rho') \quad (M_2,\ \rho) \Downarrow C \quad (N,\ \rho', (x, C)) \Downarrow C'}{(M_1\ M_2,\ \rho) \Downarrow C'}$$

Representing terms, contexts and closures

LF representation in Beluga

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

Representing terms, contexts and closures

LF representation in Beluga

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

Computation-level data types in Beluga

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

Representing terms, contexts and closures

LF representation in Beluga

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

Computation-level data types in Beluga

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

Note: → is overloaded.

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

Representing terms, contexts and closures (revised)

LF representation in Beluga

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

Computation-level data types in Beluga

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

Representing terms, contexts and closures (revised)

LF representation in Beluga

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

Computation-level data types in Beluga

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

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

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

Evaluation using closures

Evaluation using closures

Define recursive program parametric in context

rec `eval:` $(\psi$`:ctx) [`$\psi \vdash$`tm]` $\rightarrow$ `(Var [`ψ`]` $\rightarrow$ `Clos)` $\rightarrow$ `Clos =`

Evaluation using closures

```
rec eval: (ψ:ctx) [ψ ⊢tm] → (Var [ψ] → Clos) → Clos =
fn e ⇒ fn env ⇒ case e of
```

Evaluation using closures

```
rec  eval: (ψ:ctx) [ψ ⊢tm] → (Var [ψ] → Clos)  → Clos =
fn  e ⇒ fn  env ⇒ case  e of
| [ψ ⊢#p …] ⇒env (V [ψ ⊢#p …])
```

Evaluation using closures

```
rec eval: (ψ:ctx) [ψ ⊢tm] → (Var [ψ] → Clos) → Clos =
fn e ⇒ fn env ⇒ case e of
| [ψ ⊢#p ...] ⇒env (V [ψ ⊢#p ...])
| [ψ ⊢lam λx. E ...x] ⇒ Cl [ψ,x:tm ⊢E ...x] env
```

Evaluation using closures

```
rec eval: (ψ:ctx) [ψ ⊢tm] → (Var [ψ] → Clos) → Clos =
fn e ⇒ fn env ⇒ case e of
| [ψ ⊢#p ...] ⇒env (V [ψ ⊢#p ...])
| [ψ ⊢lam λx. E ...x] ⇒ Cl [ψ,x:tm ⊢E ...x] env

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

Evaluation using closures

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

Features

- Pattern matching on contextual objects **and** computation-level data constructors
- Matching on contexts to lookup variables

Weak Normalization

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

Weak Normalization

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

Theorem

If $\vdash M : A$ then M halts.

Proof.

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

□

Representing terms and evaluation in LF

Revisiting our definition of lambda-terms

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

Operational semantics

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

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

```
datatype halts : tm A → type =
| h/value : mstep M M’ → value M’ → halts M;
```

Reducibility Candidates

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

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

Reducibility Candidates

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

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

Computation-level data types in Beluga

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

- Not strictly positive definition, but stratified.

Reducibility Candidates

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

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

Computation-level data types in Beluga

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

- Not strictly positive definition, but stratified.

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

```
datatype RedSub : (Γ:ctx){σ: ⊢ Γ} ctype =
| Nil : RedSub [ ⊢ ^ ]
| Cons : RedSub [ ⊢ σ] → Reduce [ ⊢ A] [ ⊢ M] → RedSub [ ⊢ σ M ];
```

Generalization of Fundamental Lemma

Lemma (Main lemma)

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

Proof.

Case $\dfrac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$

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

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

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

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

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

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

Hence $[\sigma](\lambda x.M) \in \mathcal{R}_{A\to B}$ by definition

☐

Theorems as Computation-level Types

Lemma (Backward closed)

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

Computation-level Type in Beluga

```
rec closed : [⊢mstep M M'] → Reduce [⊢A] [⊢M'] → Reduce [⊢A] [⊢M] = ? ;
```

Lemma (Main lemma)

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

Computation-level Type in Beluga

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

Fundamental Lemma

Fundamental Lemma

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

Fundamental Lemma

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

Fundamental Lemma

```
rec closed : [ ⊢mstep M M'] →Reduce [ ⊢A] [ ⊢M'] →Reduce [ ⊢A] [ ⊢M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢tm A]} RedSub [ ⊢σ] →Reduce [ ⊢A] [ ⊢M σ] =
mlam Γ⇒mlam M ⇒fn rs ⇒ case [Γ ⊢M ...] of
| [Γ ⊢#p ...] ⇒lookup [Γ] [Γ ⊢#p ...] rs
| [Γ ⊢ lam (λx. M1 ... x)] ⇒
  Arr [ ⊢ h/value s/refl v/lam]
   (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
                                  (main [Γ,x:tm _] [Γ,x ⊢ M1 ... x] (Cons rs rN)))
```

Fundamental Lemma

```
rec closed : [ ⊢mstep M M’] →Reduce [ ⊢A] [ ⊢M’] →Reduce [ ⊢A] [ ⊢M] = ? ;

rec main : {Γ:ctx}{M:[Γ ⊢tm A]} RedSub [ ⊢ σ] →Reduce [ ⊢A] [ ⊢M σ] =

mlam Γ⇒mlam M ⇒fn rs ⇒ case [Γ ⊢M …] of

| [Γ ⊢#p …] ⇒lookup [Γ] [Γ ⊢#p …] rs

| [Γ ⊢ lam (λx. M1 … x)] ⇒
  Arr [ ⊢ h/value s/refl v/lam]
   (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
                              (main [Γ,x:tm _] [Γ,x ⊢ M1 … x] (Cons rs rN)))


| [Γ ⊢ app (M1 …) (M2 …)] ⇒
  let Arr ha f = main [Γ] [Γ ⊢ M1 …] rs in
  f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2 …] rs)
```

Fundamental Lemma

```
rec closed : [ ⊢mstep M M'] →Reduce [ ⊢A] [ ⊢M'] →Reduce [ ⊢A] [ ⊢M] = ? ;
rec main : {Γ:ctx}{M:[Γ ⊢tm A]} RedSub [ ⊢σ] →Reduce [ ⊢A] [ ⊢M σ] =
mlam Γ⇒mlam M ⇒fn rs ⇒ case [Γ ⊢M …] of
| [Γ ⊢#p …] ⇒lookup [Γ] [Γ ⊢#p …] rs
| [Γ ⊢ lam (λx. M1 … x)] ⇒
  Arr [ ⊢ h/value s/refl v/lam]
   (mlam N ⇒ fn rN ⇒ closed [ ⊢ s/beta]
                                     (main [Γ,x:tm _] [Γ,x ⊢ M1 … x] (Cons rs rN)))


| [Γ ⊢ app (M1 …) (M2 …)] ⇒
  let Arr ha f = main [Γ] [Γ ⊢ M1 …] rs in
  f [ ⊢ _ ] (main [Γ] [Γ ⊢ M2 …] rs)


| [Γ ⊢ c] ⇒ I [ ⊢ h/value s/refl v/c];
```

- Direct encoding of on-paper proof
- Equations about substitution properties automatically discharged (amounts to roughly a dozen lemmas about substitution and weakening)
- Total encoding about 75 lines of Beluga code

Other examples and comparison

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

$\Longrightarrow$ Sufficient evidence that Beluga is ideally suited to support such advanced proofs

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

What have we achieved?

- Foundation for programming proofs in context (joint work with A. Cave [POPL'12])
  - Proof term language for first-order logic over contextual LF as domain
  - Uniform treatment of contextual types, context, ...
  - Modular foundation for dependently-typed programming with phase-distinction ⇒ Generalization of DML and ATS
  - Non-termination or effects are allowed, although we often want to concentrate on pure total programs.
- Extending contextual LF with first-class substitutions and their equational theory (joint work with A. Cave [LFMTP'13])
- Rich set of examples
  - Type-preserving compiler for simply typed lambda-calculus (joint work with O. Savary Belanger, S. Monnier [CPP'13])
  - (Weak) Normalization proofs (A. Cave)
- Latest release in Jan'14: Support for indexed data types, first-class substitutions, equational theory behind substitutions

This talk

Design and implementation of Beluga

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

This talk

Design and implementation of Beluga

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

Conclusion

Beluga$^{\mu}$: programming proofs in context

- Level 1: Contextual LF
  - Supports for specifying formal systems in LF
  - Embed contexts and contextual LF objects into computations and types
  - First-class substitution and contexts together with rich equational theory
- Level 2: Functional programming language supporting indexed types
  - Pattern match and manipulate contextual LF objects
  - Proof terms language for first-order logic over contextual LF
  - Supports indexed recursive types

$\Longrightarrow$ Elegant and compact framework for programming proofs.

"A language that doesn't affect the way you think about programming, is not worth knowing." *- Alan Perlis*

Current work

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

Current work

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

The end

Thank you!

Download prototype and examples at

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

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