Proofs in Continuation Passing Style

Normalization of System T Enriched with Sums and Delimited Control

Danko Ilik, Inria, France

September 8, 2014; Canterbury, UK
Tutorial Goals

1. Motivate usefulness of proofs in CPS

2. Enable participants to practice the technique “in the privacy of one's own mind” (Barendregt)
   - doing a sequence of exercises
   - guided by the Agda proof assistant
Why Write Proofs in CPS

- Allows to use classical-logic-like proofs inside intuitionistic systems (ex. Coq, Agda proof assistants)
  - This may actually allow you to prove statements you do not know how to prove directly
- Does not lose constructivity
  - Proofs are still pure functional programs
Is Classical Logic Constructive?

Not really!

As soon as you go out of pure logic by adding axioms, like the Induction Axiom (Peano Arithmetic), or Choice Axiom (Classical Analysis), problems appear

Ex. you can refute Church's Thesis
The Purely Existential Formulas

- **Double negation translation** translates *all* classical formulas/proofs to intuitionistic ones

\[ A^\perp := (A \Rightarrow \bot) \Rightarrow \bot \]

\[
\begin{align*}
(A \land B)^\perp & := A^\perp \land B^\perp \\
(A \lor B)^\perp & := A^\perp \lor B^\perp \\
(A \Rightarrow B)^\perp & := A^\perp \Rightarrow B^\perp \\
(\forall x A)^\perp & := \forall x A^\perp \\
(\exists x A)^\perp & := \exists x A^\perp
\end{align*}
\]

- But *not all* formulas are left unchanged by the translation

- Formulas that do not contain \( \Rightarrow \) and \( \forall \) are **resistant** to the double negation translation, for them we can derive

\[ A^\perp \Rightarrow A \]

- These are called the **purely existential** or \( \Sigma-0-1 \) formulas
CPS–, A– and Double Negation Translation

- The double negation translation showed is the same as the CPS translation of types, mapping a type to its continuation monad.

- The meta-proof technique for \( A \Downarrow \Rightarrow A \) is known as A–translation but the same as defining run for a continuation monad.

Running programs in CPS is the same as combining double negation– and A–translation.
When to Use Proofs in CPS

When your goal is a data type

i.e.

Not a function or $\Pi$-type

For example,

an inductive type in Coq/Agda
Case study:

Gödel's System T extended with sums and delimited control
Why System T

• The pure fragment of Functional Programming
  – Extends simply typed lambda calculus with datatypes and recursion
• The target language for extracting programs in Foundations of Mathematics
  – Higher-type primitive recursion
    • Ex. allowing to define the diagonal Ackermann function
Why System T

- Elegant proof of Normalization of simply typed lambda calculus
  - known as Normalization by Evaluation (NBE) [Berger-Schwichtenberg]
- But extending the proof to sums, recursion, and delimited control does not seem direct
  - does exist as a program: Type Directed Partial Evaluation (TDPE) [Danvy]
NBE for $\Lambda \rightarrow$ (Simply Typed Lambda Calculus)

data Formula : Set where
  $\_$ : Proposition $\rightarrow$ Formula
  _$\Rightarrow$ _ : Formula $\rightarrow$ Formula $\rightarrow$ Formula

data _$\vdash$ _ : List Formula $\rightarrow$ Formula $\rightarrow$ Set where
  hyp : $\forall \{\Gamma A\} \rightarrow (A :: \Gamma) \vdash A$
  wkn : $\forall \{\Gamma A B\} \rightarrow \Gamma \vdash A \rightarrow (B :: \Gamma) \vdash A$
  $\supset e : \forall \{\Gamma A B\} \rightarrow \Gamma \vdash (A \supset B) \rightarrow \Gamma \vdash A \rightarrow \Gamma \vdash B$
  $\supset i : \forall \{\Gamma A B\} \rightarrow (A :: \Gamma) \vdash B \rightarrow \Gamma \vdash (A \supset B)$

Example. $\lambda x. \lambda y. x$ is represented by $\supset i \ (\supset i \ (wkn \ hyp))$
The goal is to bring lambda terms into the following (eta-long beta-normal) form, mutually defined by normal and neutral terms:

**mutual**

```lean
data _⊢ʳ_ : List Formula → Formula → Set where
  ⊢i : ∀ {Γ A B} → (A :: Γ) ⊢ʳ B → Γ ⊢ʳ (A ⊃ B)
  e2r : ∀ {Γ A} → Γ ⊢ᵉ A → Γ ⊢ʳ A

data _⊢ᵉ_ : List Formula → Formula → Set where
  ⊢e : ∀ {Γ A B} → Γ ⊢ᵉ (A ⊃ B) → Γ ⊢ʳ A → Γ ⊢ᵉ B
  hyp : ∀ {Γ A} → (A :: Γ) ⊢ᵉ A
  wkn : ∀ {Γ A B} → Γ ⊢ᵉ A → (B :: Γ) ⊢ᵉ A
```

**Example.** At type (τ→τ)→(τ→τ) the term λx. x (⊢i hyp) is not in normal form, but λx. λy. x y (⊢i (⊢i (e2r (⊢e (wkn hyp) hyp)))) is.

**Note.** The constructor e2r is an explicit conversion from a neutral to normal term.
The method is to write an evaluator and an inverse to the evaluator and then compose them:

\[
\text{nbe} : \forall \{\Gamma \ A\} \rightarrow \Gamma \vdash A \rightarrow \Gamma \vdash^r A
\]
\[
\text{nbe} \{\Gamma\} \ p = \text{reify} \ (\text{soundness} \ p \ (\text{reflect-context} \ \Gamma))
\]

soundness : \forall \{\Gamma \ A\} \\
\rightarrow \Gamma \vdash A \\
\rightarrow \{w : K\} \rightarrow w \vdash \Gamma \\
\rightarrow w \vdash A

mutation : 
\[
\text{reify} : \forall \{A \ \Gamma\} \\
\rightarrow \Gamma \vdash A \rightarrow \Gamma \vdash^r A
\]
\[
\text{reflect} : \forall \{A \ \Gamma\} \\
\rightarrow \Gamma \vdash^e A \rightarrow \Gamma \vdash A
\]
NBE for $\Lambda \imp$

- The technique will be the same for all extension of NBE beyond $\Lambda \imp$
- The only thing that will change is the **forcing relation** $\models$ that defines the semantics target of the evaluator
- For $\Lambda \imp$, forcing is defined as

  \[
  \models_\_ : K \to \text{Formula} \to \text{Set}
  \]

  \[
  w \models (A \supset B) = \{w' : K\} \to w' \geq w \to w' \models A \to w' \models B \]

  \[
  w \models ($ P) = w \models^a P
  \]

  which is the clause for forcing of implication in Kripke models
We now turn to live proving

Agda keystroke memento:

- Ctrl+C+L to load a file
- Ctrl+C+R to refine a proof hole
- Ctrl+C+T to show type of proof hole
- Ctrl+C+N to normalize a term within a hole
- Ctrl+C+D to show type of a term written within a hole
- Ctrl+C+E to show typing environment in a hole
- Ctrl+C+X+D to disable hole focus
NBE for $\Lambda \Rightarrow$ in Continuation Passing Style

- We make the forcing relation a continuation monad:

\[
\text{mutual}
\]
\[
\begin{align*}
_\models^- : K \to \text{Formula} \to \text{Set} \\
w \models^- (P) &= w \models^a P \\
w \models^- (A \supset B) &= \{w' : K\} \to w' \geq w \to w' \models A \to w' \models B
\end{align*}
\]
\[
_\models^- : K \to \text{Formula} \to \text{Set} \\
w \models A = (C : \text{Formula}) \to \forall \{w_1\} \to w_1 \geq w \\
\to (\forall \{w_2\} \to w_2 \geq w_1 \to w_2 \models^- A \to w_2 \models^a C) \\
\to w_1 \models^a C
\]
NBE for $\Lambda \Rightarrow$ in Continuation Passing Style

- Monadic operations help to structure proofs:
  
  - **return**: $\forall \{A \ w\} \rightarrow w \vdash^s A \rightarrow w \vdash A$
  
  - **bind**: $\forall \{A \ B \ w\} \rightarrow w \vdash A$
    
    $\rightarrow (\forall \{w'\} \rightarrow w' \geq w \rightarrow w' \vdash^s A \rightarrow w' \vdash B)$
    
    $\rightarrow w \vdash B$
  
  - **run**: $\forall \{w\} \rightarrow w \vdash (\$ P) \rightarrow w \vdash^s (\$ P)$
Exercise 1 – Adding sums and products

- The target language of normal forms:
  
  \texttt{mutual}

\begin{verbatim}
data _\vdash_r_ : \text{List Formula} \to \text{Formula} \to \text{Set} \ where
  \text{\_i} : \forall \{\Gamma \ A \ B\} \to (A :: \Gamma) \vdash_r B \to \Gamma \vdash_r (A \supset B)
  \text{\_i1} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_r A \to \Gamma \vdash_r (A \lor B)
  \text{\_i2} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_r B \to \Gamma \vdash_r (A \lor B)
  \text{\_e2r} : \forall \{\Gamma \ A\} \to \Gamma \vdash_e A \to \Gamma \vdash_r A
  \text{\_i} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_r A \to \Gamma \vdash_r B \to \Gamma \vdash_r (A \land B)

data _\vdash_e_ : \text{List Formula} \to \text{Formula} \to \text{Set} \ where
  \text{\_hyp} : \forall \{\Gamma \ A\} \to (A :: \Gamma) \vdash_e A
  \text{\_e} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_e (A \supset B) \to \Gamma \vdash_r A \to \Gamma \vdash_e B
  \text{\_ve} : \forall \{\Gamma \ A \ B \ C\} \to \Gamma \vdash_e (A \lor B) \to (A :: \Gamma) \vdash_r C \to (B :: \Gamma) \vdash_r C \to \Gamma \vdash_e C
  \text{\_wkn} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_r A \to (B :: \Gamma) \vdash_e A
  \text{\_le1} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_e (A \land B) \to \Gamma \vdash_e A
  \text{\_le2} : \forall \{\Gamma \ A \ B\} \to \Gamma \vdash_e (A \land B) \to \Gamma \vdash_e B
\end{verbatim}

\end{verbatim}
Exercise 1 – Adding sums and products

- We modify the forcing relation by inserting double negation translation and A-translation:

\[
\text{mutual} \quad \_ \vdash^s _{-} : K \rightarrow \text{Formula} \rightarrow \text{Set} \\
_{w} \vdash^s (\$ P) = w \vdash^a \$ P \\
_{w} \vdash^s (A \supset B) = \{w' : K \rightarrow w' \geq w \rightarrow w' \vdash A \rightarrow w' \vdash B\} \\
_{w} \vdash^s (A \lor B) = w \vdash A \lor w \vdash B \\
_{w} \vdash^s (A \land B) = w \vdash A \land w \vdash B
\]

\[
\_ \vdash^- _{-} : K \rightarrow \text{Formula} \rightarrow \text{Set} \\
w \vdash^- A = (C : \text{Formula}) \rightarrow \forall \{w_{1}\} \rightarrow w_{1} \geq w \\
\rightarrow (\forall \{w_{2}\} \rightarrow w_{2} \geq w_{1} \rightarrow w_{2} \vdash^s A \rightarrow w_{2} \vdash^a C) \\
\rightarrow w_{1} \vdash^a C
\]
Exercise 1 – Adding sums and products

• The call-by-value variant of forcing:

\[
\text{mutual}
\]

\[
\vdash_S^w : K \rightarrow \text{Formula} \rightarrow \text{Set}
\]

\[
w \vdash_S (\$ P) = w \vdash_A \$ P
\]

\[
w \vdash_S (A \supset B) = \{w' : K\} \rightarrow w' \geq w \rightarrow w' \vdash_S A \rightarrow w' \vdash B
\]

\[
w \vdash_S (A \lor B) = w \vdash_S A \cup w \vdash_S B
\]

\[
w \vdash_S (A \land B) = w \vdash_S A \times w \vdash_S B
\]

\[
\vdash_\_ : K \rightarrow \text{Formula} \rightarrow \text{Set}
\]

\[
w \vdash A = (C : \text{Formula}) \rightarrow \forall \{w_1\} \rightarrow w_1 \geq w
\]

\[
\rightarrow (\forall \{w_2\} \rightarrow w_2 \geq w_1, \rightarrow w_2 \vdash_S A \rightarrow w_2 \vdash_A C)
\]

\[
\rightarrow w_1 \vdash_A C
\]
Exercise 1

- You are given the complete NBE proof in CPS for \( \Lambda \rightarrow^{+\times} \) i.e. simply typed lambda calculus extended with product and sum types;
- This proof uses the call-by-name continuation monad;
- Finish the missing goals of Exercise1.agda in order to prove the same result, but this time using the call-by-value continuation monad;
- Try to use monadic operators (return, bind) as much as possible;
- To check if your implementation is right, normalize the test cases at the end of the file, and compare with the expected output.
Exercise 2 – Adding Primitive Recursion

- Target language:

```
mutual
data _⊢ʳ_ : K → Formula → Set where
    ...
    zero : ∀ {Γ} → Γ ⊢ʳ ℕ
    succ : ∀ {Γ} → Γ ⊢ʳ ℕ → Γ ⊢ʳ ℕ

data _⊢ᵉ_ : K → Formula → Set where
    ...
    rec : ∀ {Γ C} → Γ ⊢ᵉ ℕ → Γ ⊢ʳ C
        → Γ ⊢ʳ (ℕ ⊆⁺ (C ⊆⁺ C)) → Γ ⊢ᵉ C
```
Exercise 2 – Adding Primitive Recursion

- Major difference: evaluation is no longer into an arbitrary Kripke model, but must use the universal model.

- Because evaluation of recursion presupposes reify/reflect pair is already defined!!

- Forcing relation stays the same, but we are no longer abstract, ex. we put:

  \[ w \models^s \mathbb{N} = w \models^r \mathbb{N} \]
Exercise 2

- You are given the file `Exercise2.agda` containing an incomplete proof of NBE for simply typed lambda calculus with sums, products, and higher type primitive recursion;

- Complete the missing goals, and check how your proof computes the test cases.
Exercise 3 – Adding Control Delimited at $\mathbb{N}$

- We need to know if a term is annotated or not:

```plaintext
data Annot : Set where
    - : Annot
    + : Annot

data _⊢_!_ : K → Formula → Annot → Set where ...

mutual
data _⊢ʳ_!_ : K → Formula → Annot → Set where ...
    ...
data _⊢ₑ_!_ : K → Formula → Annot → Set where ...
    ...
```
Exercise 3 – Adding Control Delimited at $\mathbb{N}$

- Target language:

\[
\text{mutual}\ 
\begin{array}{l}
\text{data } _\vdash^r_! : K \to \text{Formula} \to \text{Annot} \to \text{Set} \quad \text{where}
\left.\right.
\begin{array}{l}
\vdots
\end{array}
\end{array}
\]

\[
\begin{array}{l}
\text{data } _\vdash^e_! : K \to \text{Formula} \to \text{Annot} \to \text{Set} \quad \text{where}
\left.\right.
\begin{array}{l}
\vdots
\end{array}
\end{array}
\]

\begin{align*}
\text{reset} : & \forall \{\Gamma\} \to \Gamma \vdash^e \mathbb{N} ! + \to \Gamma \vdash^e \mathbb{N} ! - \\
\text{shift} : & \forall \{\Gamma A\} \to (A \supset \mathbb{N} :: \Gamma) \vdash^e \mathbb{N} ! + \to \Gamma \vdash^e A ! +
\end{align*}
Exercise 3 – Adding Control Delimited at $\mathbb{N}$

- Strong forcing of implication accounts for annotations $\mathbf{a}$:

$$w \models^s (A \supset B)! \mathbf{a} =
\{w' : K\} \rightarrow w' \geq w
\rightarrow \{\mathbf{a}' : \text{Annot}\} \rightarrow \mathbf{a}' \succeq \mathbf{a}
\rightarrow w' \models^s A! \mathbf{a}' \rightarrow w' \models B! \mathbf{a}'$$

- And so does the answer type of the monad:

$$\_ \vdash^r \_ \!: K \rightarrow \text{Formula} \rightarrow \text{Annot} \rightarrow \text{Set}
w \vdash^r C! - = w \vdash^r C! -
w \vdash^r C! + = w \vdash^r \mathbb{N}! +$$
You are given the file `Exercise3.agda` containing an incomplete proof of NBE for simply typed lambda calculus with sums, products, and higher type primitive recursion;

Complete the missing goals, and check how your proof computes the test cases.
Solutions to Exercises

Will be posted on

www.lix.polytechnique.fr/~danko/PPDP-2014-tutorial/

(on Thursday morning)