Distilled Tutorial – PPDP 2014

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

$$A^{\perp} := (A_{\perp} \supset \perp) \supset \perp$$

$$(A \wedge B)_{\perp} := A^{\perp} \wedge B^{\perp}$$
$$(A \vee B)_{\perp} := A^{\perp} \vee B^{\perp}$$
$$(A \supset B)_{\perp} := A^{\perp} \supset B^{\perp}$$
$$(\forall x A)_{\perp} := \forall x A^{\perp}$$
$$(\exists x A)_{\perp} := \exists x A^{\perp}$$

- But ***not all*** formulas are left unchanged by the translation
- Formulas that do not contain $\supset$ and $\forall$ are ***resistant*** to the double negation translation, for them we can derive

$$A^{\perp} \supset A$$

- These are called the **purely existential** or **∑-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

$$A^{\perp} \supset A$$

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 Π-type

For example,

an inductive type in Coq/Agda

Proofs in CPS

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 <u>proof</u> to sums, recursion, and delimited control does not seem direct
  - does exist as a <u>program</u>: Type Directed Partial Evaluation (**TDPE**) [Danvy]

NBE for $\Lambda^{\to}$ (Simply Typed Lambda Calculus)

```
data Formula : Set where
  $_  : Proposition → Formula
  _⊃_ : Formula → Formula → Formula

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

Example. λx. λy. x is represented by `⊃i (⊃i (wkn hyp))`

NBE for $\Lambda^{\to}$

The goal is to bring lambda terms into the following (eta-long beta-normal) form, mutually defined by *normal* and *neutral* terms:

```
mutual
  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 $(\tau\to\tau)\to(\tau\to\tau)$ the term $\lambda x.\ x$ (`⊃i hyp`) is not in normal form, but $\lambda x.\ \lambda y.\ x\ y$ (`⊃i (⊃i (e2r (⊃e (wkn hyp) hyp)))`) is.

Note. The constructor `e2r` is an explicit conversion from a neutral to normal term.

NBE for $\Lambda^{\to}$

The method is to write an evaluator and an inverse to the evaluator and then compose them:

```
nbe : ∀ {Γ A} → Γ ⊢ A → Γ ⊢ʳ A
nbe {Γ} p = reify (soundness p (reflect-context Γ))
```

```
soundness : ∀ {Γ A}
→ Γ ⊢ A → {w : K} → w ⊩ Γ
→ w ⊩ A
```

```
mutual
  reify : ∀ {A Γ}
     → Γ ⊩ A → Γ ⊢ʳ A
  reflect : ∀ {A Γ}
     → Γ ⊢ᵉ A → Γ ⊩ A
```

NBE for $\Lambda^{\rightarrow}$

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

```
_⊩_ : K → Formula → Set
w ⊩ (A ⊃ B) = {w' : K} → w' ≥ w → w' ⊩ A → w' ⊩ B
w ⊩ ($ P) = w ⊩ᵃ P
```

which is the clause for forcing of implication in Kripke models

NBE for $\Lambda^{\rightarrow}$ in Agda

- 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 $\Lambda^{\to}$ in Continuation Passing Style

- We make the forcing relation a continuation monad:

```
mutual
 _⊩ˢ_ : K → Formula → Set
 w ⊩ˢ ($ P) = w ⊩ᵃ $ P
 w ⊩ˢ (A ⊃ B) = {w' : K} → w' ≥ w → w' ⊩ A → w' ⊩ B

 _⊩_ : K → Formula → Set
 w ⊩ A = (C : Formula) → ∀ {w₁} → w₁ ≥ w
           → (∀ {w₂} → w₂ ≥ w₁ → w₂ ⊩ˢ A → w₂ ⊩ᵃ C)
           → w₁ ⊩ᵃ C
```

NBE for $\Lambda^{\rightarrow}$ in Continuation Passing Style

- Monadic operations help to structure proofs:
- `return : ∀ {A w} → w ⊩ˢ A → w ⊩ A`
- ```
  bind : ∀ {A B w} → w ⊩ A
    → (∀ {w'} → w' ≥ w → w' ⊩ˢ A → w' ⊩ B)
    → w ⊩ B
  ```
- `run : ∀ {w} → w ⊩ ($ P) → w ⊩ˢ ($ P)`
  ```

Exercise 1 – Adding sums and products

- The target language of normal forms:

```
mutual
  data _⊢ʳ_ : List Formula → Formula → Set where
    ⊃i  : ∀ {Γ A B} → (A ∷ Γ) ⊢ʳ B → Γ ⊢ʳ (A ⊃ B)
    ∨i1 : ∀ {Γ A B} → Γ ⊢ʳ A → Γ ⊢ʳ (A ∨ B)
    ∨i2 : ∀ {Γ A B} → Γ ⊢ʳ B → Γ ⊢ʳ (A ∨ B)
    e2r : ∀ {Γ A} → Γ ⊢ᵉ A → Γ ⊢ʳ A
    ∧i  : ∀ {Γ A B} → Γ ⊢ʳ A → Γ ⊢ʳ B → Γ ⊢ʳ (A ∧ B)

  data _⊢ᵉ_ : List Formula → Formula → Set where
    hyp : ∀ {Γ A} → (A ∷ Γ) ⊢ᵉ A
    ⊃e  : ∀ {Γ A B} → Γ ⊢ᵉ (A ⊃ B) → Γ ⊢ʳ A → Γ ⊢ᵉ B
    ∨e  : ∀ {Γ A B C} → Γ ⊢ᵉ (A ∨ B) → (A ∷ Γ) ⊢ʳ C → (B ∷
Γ) ⊢ʳ C → Γ ⊢ᵉ C
    wkn : ∀ {Γ A B} → Γ ⊢ʳ A → (B ∷ Γ) ⊢ᵉ A
    ∧e1 : ∀ {Γ A B} → Γ ⊢ᵉ (A ∧ B) → Γ ⊢ᵉ A
    ∧e2 : ∀ {Γ A B} → Γ ⊢ᵉ (A ∧ B) → Γ ⊢ᵉ B
```

Exercise 1 – Adding sums and products

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

```
mutual
 _⊩ˢ_ : K → Formula → Set
 w ⊩ˢ ($ P) = w ⊩ᵃ $ P
 w ⊩ˢ (A ⊃ B) = {w' : K} → w' ≥ w → w' ⊩ A → w' ⊩ B
 w ⊩ˢ (A ∨ B) = w ⊩ A ⊎ w ⊩ B
 w ⊩ˢ (A ∧ B) = w ⊩ A × w ⊩ B

 _⊩_ : K → Formula → Set
 w ⊩ A = (C : Formula) → ∀ {w₁} → w₁ ≥ w
           → (∀ {w₂} → w₂ ≥ w₁ → w₂ ⊩ˢ A → w₂ ⊩ᵃ C)
           → w₁ ⊩ᵃ C
```

Exercise 1 – Adding sums and products

- The call-by-value variant of forcing:

```
mutual
 _⊩ˢ_ : K → Formula → Set
 w ⊩ˢ ($ P) = w ⊩ᵃ $ P
 w ⊩ˢ (A ⊃ B) = {w' : K} → w' ≥ w → w' ⊩ˢ A → w' ⊩ B
 w ⊩ˢ (A ∨ B) = w ⊩ˢ A ⊎ w ⊩ˢ B
 w ⊩ˢ (A ∧ B) = w ⊩ˢ A × w ⊩ˢ B

 _⊩_ : K → Formula → Set
 w ⊩ A = (C : Formula) → ∀ {w₁} → w₁ ≥ w
          → (∀ {w₂} → w₂ ≥ w₁ → w₂ ⊩ˢ A → w₂ ⊩ᵃ C)
          → w₁ ⊩ᵃ C
```

Exercise 1

- You are given the complete NBE proof in CPS for $\Lambda^{\to + \times}$ 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:

$$w \Vdash^s \mathbb{N} = w \vdash^r \mathbb{N}$$

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 ℕ

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

```
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 ℕ

- Target language:

```
mutual
  data _⊢ʳ_!_ : K → Formula → Annot → Set where
    …
  data _⊢ᵉ_!_ : K → Formula → Annot → Set where
    …
    reset : ∀ {Γ} → Γ ⊢ᵉ ℕ ! + → Γ ⊢ᵉ ℕ ! –
    shift : ∀ {Γ A} → (A ⊃ ℕ ∷ Γ) ⊢ᵉ ℕ ! + → Γ ⊢ᵉ A ! +
```

Exercise 3 – Adding Control Delimited at $\mathbb{N}$

- Strong forcing of implication accounts for annotations a:

```
w ⊩ˢ (A ⊃ B) ! a =
   {w' : K} → w' ≥ w
→ {a' : Annot} → a' ≳ a
→ w' ⊩ˢ A ! a' → w' ⊩ B ! a'
```

- And so does the answer type of the monad:

```
_⊢-ʳ_!_ : K → Formula → Annot → Set
w ⊢-ʳ C ! ₋ = w ⊢ʳ C ! ₋
w ⊢-ʳ C ! ₊ = w ⊢ʳ ℕ ! ₊
```

Exercise 3

- 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)