Proofs, upside down

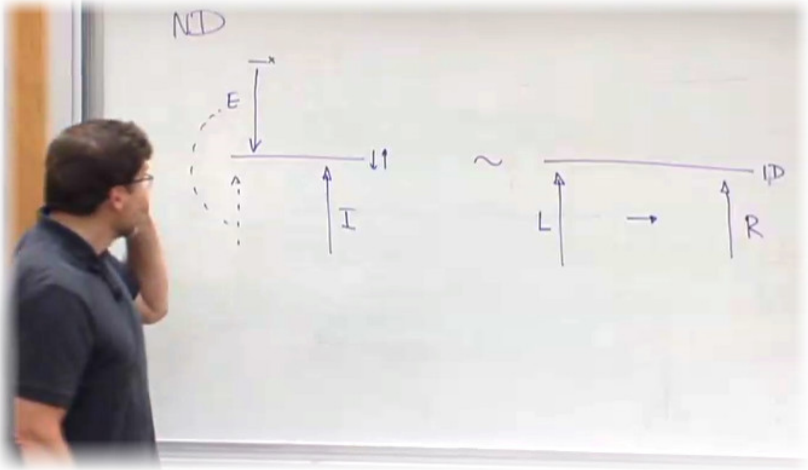
A functional correspondence between
natural deduction and the *sequent calculus*

Matthias Puech

AARHUS UNIVERSITY

APLAS'13
Melbourne, December 11, 2013

An intuition



Natural deductions are "reversed" sequent calculus proofs

An intuition

Problem

How to make this intuition formal?

- how to define “reversal” generically?
- from N.D., how to *derive* S.C.?

and now, for something completely different. . .

Accumulator-passing style

A well-known programmer trick to save stack space

Accumulator-passing style

A well-known programmer trick to save stack space

- a function in direct style:

```
let rec tower1 = function
  | [] → 1
  | x :: xs → x ** tower1 xs
```

Accumulator-passing style

A well-known programmer trick to save stack space

- a function in direct style:

```
let rec tower1 = function
  | [] → 1
  | x :: xs → x ** tower1 xs
```

- the same in accumulator-passing style:

```
let rec tower2 acc = function
  | [] → acc
  | x :: xs → tower2 (x ** acc) xs
```

Accumulator-passing style

A well-known programmer trick to save stack space

- a function in direct style:

```
let rec tower1 = function
  | [] → 1
  | x :: xs → x ** tower1 xs
```

- the same in accumulator-passing style:

```
let rec tower2 acc = function
  | [] → acc
  | x :: xs → tower2 (x ** acc) xs

(* don't forget to reverse the input list *)
let tower xs = tower2 1 (List.rev xs)
```

In this talk

$$\frac{\text{sequent calculus}}{\text{natural deduction}} = \frac{\texttt{tower2}}{\texttt{tower1}}$$

In this talk

$$\frac{\text{sequent calculus}}{\text{natural deduction}} = \frac{\texttt{tower2}}{\texttt{tower1}}$$

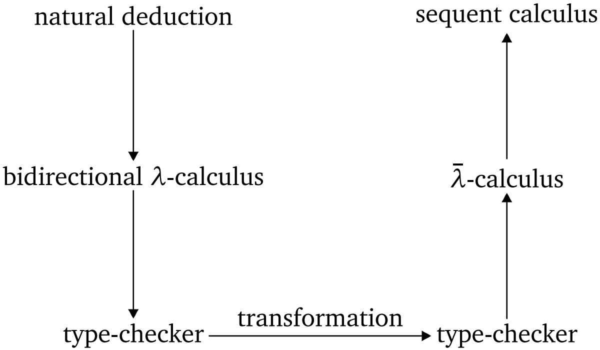
The message

- S.C. is an accumulator-passing N.D.
- there is a systematic, off-the-shelf transformation from N.D.-style systems to S.C.-style systems
- it is modular, i.e., it applies to variants of N.D./S.C.
- a programmatic explanation of a proof-theoretical artifact

In this talk

The medium

Go through term assignments and reason on the type checker:



Outline

The transformation

Some extensions

Outline

Starting point: the Bidirectional λ-calculus

a.k.a. intercalations, normal forms+annotation [Pierce and Turner, 2000]

$$A ::= \mathsf{p} \mid A \supset A \qquad \text{Types}$$

$$M ::= \lambda x. M \mid R \qquad \text{Terms}$$

$$R ::= R\,M \mid x \mid (M : A) \qquad \text{Atoms}$$

$\boxed{\Gamma \vdash R \Rightarrow A}$ Inference

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{App}\ \frac{\Gamma \vdash R \Rightarrow A \supset B \qquad \Gamma \vdash M \Leftarrow A}{\Gamma \vdash R\,M \Rightarrow B} \qquad \textsc{Annot}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash (M : A) \Rightarrow A}$$

$\boxed{\Gamma \vdash M \Leftarrow A}$ Checking

$$\textsc{Lam}\ \frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x. M \Leftarrow A \supset B} \qquad \textsc{Atom}\ \frac{\Gamma \vdash R \Rightarrow C}{\Gamma \vdash R \Leftarrow C}$$

Starting point: the Bidirectional λ-calculus

```
type a = Base | Imp of a × a
type m = Lam of string × m | Atom of r
and r = App of r × m | Var of string | Annot of m × a

let rec check env c : m → unit =
 let rec infer : r → a = fun r → match r with
 | Var x → List.assoc x env
 | Annot (m, a) → check env a m; a
 | App (r, m) → let Imp (a, b) = infer r in check env a m; b
 in fun m → match m, c with
  | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
  | Atom r, _ → match infer r with c' when c=c' → ()
```

Starting point: the Bidirectional λ-calculus

```
type a = Base | Imp of a × a
type m = Lam of string × m | Atom of r
and r = App of r × m | Var of string | Annot of m × a

let rec check env c : m → unit =
 let rec infer : r → a = fun r → match r with
 | Var x → List.assoc x env
 | Annot (m, a) → check env a m; a
 | App (r, m) → let Imp (a, b) = infer r in check env a m; b
 in fun m → match m, c with
  | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
  | Atom r, _ → match infer r with c' when c=c' → ()
```

Remarks

- inference in constant environment → infer λ-dropped

Starting point: the Bidirectional λ-calculus

```
type a = Base | Imp of a × a
type m = Lam of string × m | Atom of r
and r = App of r × m | Var of string | Annot of m × a

let rec check env c : m → unit =
 let rec infer : r → a = fun r → match r with
  | Var x → List.assoc x env
  | Annot (m, a) → check env a m; a
  | App (r, m) → let Imp (a, b) = infer r in check env a m; b
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → match infer r with c' when c=c' → ()
```

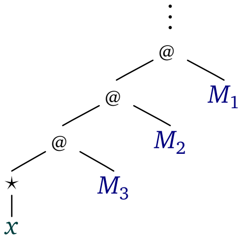
Remarks

- inference in constant environment → infer λ-dropped
- infer is head-recursive

Inefficiency: no tail recursion

```
(* ... *)
 let rec infer : r → a = fun r → match r with
  | Var x → List.assoc x env
  | Annot (m, a) → check env a m; a
  | App (r, m) → let Imp (a, b) = infer r in check env a m; b
(* ... *)
```

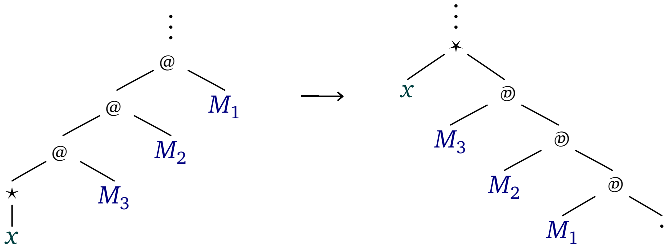
Example



Solution: reverse atomic terms

```
(* ... *)
 let rec infer : r → a = fun r → match r with
 | Var x → List.assoc x env
 | Annot (m, a) → check env a m; a
 | App (r, m) → let Imp (a, b) = infer r in check env a m; b
(* ... *)
```

Example



The transformation

An application of Danvy and Nielsen [2001]'s framework:

- (partial) *CPS transformation*
- (lightweight) *defunctionalization*
- *reforestation* (= deforestation^{-1})

Turns *direct style* into *accumulator-passing style*

Step 1. CPS transformation of infer (call-by-value)

```
let rec check env c : m → unit =
 let rec infer : r → a = fun r → match r with
   | Var x → List.assoc x env
   | Annot (m, a) → check env a m; a
   | App (r, m) → let Imp (a, b) = infer r in check env a m; b
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → match infer r with c' when c=c' → ()
```

Step 1. CPS transformation of infer (call-by-value)

```
type k = a → unit
let rec check env c : m → unit =
  let rec infer : r → k → unit = fun r k → match r with
    | Var x → k (List.assoc x env)
    | Annot (m, a) → check env a m; k a
    | App (r, m) → infer r (fun (Imp (a, b)) → check env a m; k b)
  in fun m → match m, c with
    | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
    | Atom r, _ → infer r (function c' when c=c' → ())
```

Step 1. CPS transformation of `infer` (call-by-value)

```
type k = a → unit
let rec check env c : m → unit =
  let rec infer : r → k → unit = fun r k → match r with
    | Var x → k (List.assoc x env)
    | Annot (m, a) → check env a m; k a
    | App (r, m) → infer r (fun (Imp (a, b)) → check env a m; k b)
  in fun m → match m, c with
    | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
    | Atom r, _ → infer r (function c' when c=c' → ())
```

Step 1. CPS transformation of infer (call-by-value)

```
type k = a → unit
let rec check env c : m → unit =
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → k (List.assoc x env)
   | Annot (m, a) → check env a m; k a
   | App (r, m) → infer r (fun (Imp (a, b)) → check env a m; k b)
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r (function c' when c=c' → ())
```

Step 1. CPS transformation of infer (call-by-value)

```
type k = a → unit
let rec check env c : m → unit =
  let rec infer : r → k → unit = fun r k → match r with
    | Var x → k (List.assoc x env)
    | Annot (m, a) → check env a m; k a
    | App (r, m) → infer r (fun (Imp (a, b)) → check env a m; k b)
  in fun m → match m, c with
    | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
    | Atom r, _ → infer r (function c’ when c=c’ → ())
```

Step 2. (lightweight) Defunctionalization

```
type k = a → unit
let rec check env c : m → unit =
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → k (List.assoc x env)
   | Annot (m, a) → check env a m; k a                 (* KCons *)
   | App (r, m) → infer r (fun (Imp (a, b)) → check env a m; k b)
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r (function c' when c=c' → ()) (* KNil *)
```

Step 2. (lightweight) Defunctionalization

```
type k = a → unit
let rec check env c : m → unit =
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → k (List.assoc x env)
   | Annot (m, a) → check env a m; k a
   | App (r, m) → infer r (KCons (m, k))
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r KNil
```

Step 2. (lightweight) Defunctionalization

```
type k = KNil | KCons of m × k
let rec check env c : m → unit =
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → k (List.assoc x env)
   | Annot (m, a) → check env a m; k a
   | App (r, m) → infer r (KCons (m, k))
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r KNil
```

Step 2. (lightweight) Defunctionalization

```
type k = KNil | KCons of m × k
let rec check env c : m → unit =



  let rec infer : r → k → unit = fun r k → match r with
    | Var x → k (List.assoc x env)
    | Annot (m, a) → check env a m; k a
    | App (r, m) → infer r (KCons (m, k))
  in fun m → match m, c with
    | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
    | Atom r, _ → infer r KNil
```

Step 2. (lightweight) Defunctionalization

```
type k = KNil | KCons of m × k
let rec check env c : m → unit =
 let rec apply : k → a → unit = fun k a → match k, a with
   | KNil, c’ when c=c’ → ()
   | KCons (m, k), Imp (a, b) → check env a m; k b in
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → k (List.assoc x env)
   | Annot (m, a) → check env a m; k a
   | App (r, m) → infer r (KCons (m, k))
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r KNil
```

Step 2. (lightweight) Defunctionalization

```
type k = KNil | KCons of m × k
let rec check env c : m → unit =
 let rec apply : k → a → unit = fun k a → match k, a with
   | KNil, c' when c=c' → ()
   | KCons (m, k), Imp (a, b) → check env a m; apply k b in
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → apply k (List.assoc x env)
   | Annot (m, a) → check env a m; apply k a
   | App (r, m) → infer r (KCons (m, k))
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r KNil
```

Step 2. (lightweight) Defunctionalization

```
type k = KNil | KCons of m × k
let rec check env c : m → unit =
 let rec apply : k → a → unit = fun k a → match k, a with
   | KNil, c' when c=c' → ()
   | KCons (m, k), Imp (a, b) → check env a m; apply k b in
 let rec infer : r → k → unit = fun r k → match r with
   | Var x → apply k (List.assoc x env)
   | Annot (m, a) → check env a m; apply k a
   | App (r, m) → infer r (KCons (m, k))
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → infer r KNil
```

Step 2. (lightweight) Defunctionalization

```
type k = KNil | KCons of m × k
let rec check env c : m → unit =
 let rec cont : k → a → unit = fun k a → match k, a with
   | KNil, c' when c=c' → ()
   | KCons (m, k), Imp (a, b) → check env a m; cont k b in
 let rec rev_atom : r → k → unit = fun r k → match r with
   | Var x → cont k (List.assoc x env)
   | Annot (m, a) → check env a m; cont k a
   | App (r, m) → rev_atom r (KCons (m, k))
 in fun m → match m, c with
   | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
   | Atom r, _ → rev_atom r KNil
```

Step 3. Reforestation

Goal

Introduce intermediate data structure of *reversed term* V
to decouple *reversal* from *checking*:

$$\texttt{check} \circ \texttt{rev\_atom} \circ \texttt{cont}$$
$$\downarrow$$
$$\texttt{rev} \circ \texttt{check} \circ \texttt{cont}$$

Step 3. Reforestation

```
(* intermediate data structure *)
type v = VLam of string × v | VHead of h
and h =
  | HVar of string × k
  | HAnnot of v × a × k
and k = KNil | KCons of v × k
```

Step 3. Reforestation

```
(* intermediate data structure *)
type v = VLam of string × v | VHead of h
and h =
  | HVar of string × k
  | HAnnot of v × a × k
and k = KNil | KCons of v × k

(* term reversal *)
let rec rev : m → v = fun m → match m with
  | Lam (x, m) → VLam (x, rev m)
  | Atom r → VHead (rev_atom r KNil)
and rev_atom : r → k → h = fun r k → match r with
  | Var x → HVar (x, k)
  | Annot (m, a) → HAnnot (rev m, a, k)
  | App (r, m) → rev_atom r (KCons (rev m, k))
```

Step 3. Reforestation

```
(* reversed term checking *)
let rec check env c : v → unit =
  let rec cont : k → a → unit = fun k a → match k, a with
    | KNil, c' when c=c' → ()
    | KCons (m, k), Imp (a, b) → check env a m; cont k b in
  let head h = match h with
    | HVar (x, k) → cont k (List.assoc x env)
    | HAnnot (m, a, k) → check env a m; cont k a in
  fun v → match v, c with
    | VLam (x, m), Imp (a, b) → check ((x, a) :: env) b m
    | VHead h, _ → head h

(* main function *)
let check env c m = check env c (rev m)
```

End result: the $\bar{\lambda}$-calculus

a.k.a. *spine calculus*, or LJT, or n-ary application [Herbelin, 1994]

$$V ::= \lambda x.V \mid H \qquad \text{Values}$$

$$H ::= x(S) \mid (V:A)(S) \qquad \text{Heads}$$

$$S ::= \cdot \mid V, S \qquad \text{Spines}$$

$\boxed{\Gamma \mid A \longrightarrow S : C}$ Focused left rules

$$\frac{\Gamma \longrightarrow V : A \qquad \Gamma \mid B \longrightarrow S : C}{\Gamma \mid A \supset B \longrightarrow V, S : C}\ \textsc{SApp} \qquad \frac{}{\Gamma \mid C \longrightarrow \cdot : C}\ \textsc{SAtom}$$

$\boxed{\Gamma \longrightarrow V : A}$ Right rules

$$\frac{\Gamma, x : A \longrightarrow V : B}{\Gamma \longrightarrow \lambda x.M : A \supset B}\ \textsc{VLam} \qquad \frac{x : A \in \Gamma \qquad \Gamma \mid A \longrightarrow S : C}{\Gamma \longrightarrow x(S) : C}\ \textsc{HVar}$$

$$\frac{\Gamma \longrightarrow V : A \qquad \Gamma \mid A \longrightarrow S : C}{\Gamma \longrightarrow (V:A)(S) : C}\ \textsc{HAnnot}$$

End result: the $\bar{\lambda}$-calculus

Theorem

Initial.`check env a m` $= ()$ *iff* *Final.*`check env a m` $= ()$

Proof.

By composition of the soundness of the transformations □

End result: the $\bar{\lambda}$-calculus

Theorem

$\Gamma \vdash M \Leftarrow A \qquad \textit{iff} \qquad \Gamma \longrightarrow (\texttt{rev}\, M) : A$

Proof.

By composition of the soundness of the transformations □

End result: the $\bar{\lambda}$-calculus

Theorem

$\Gamma \vdash A \qquad \textit{iff} \qquad \Gamma \longrightarrow A$

Proof.

By composition of the soundness of the transformations □

End result: the $\bar{\lambda}$-calculus

Theorem

$\Gamma \vdash A$ *iff* $\Gamma \longrightarrow A$

Proof.

By composition of the soundness of the transformations □

Remark

we derived the rules of LJT

Outline

The transformation

Some extensions

Extension 1. Full propositional intuitionistic N.D.

It scales to full NJ [Herbelin, 1995]:

$$A \; ::= \; \mathtt{p} \;\Big|\; A \supset A \;\Big|\; A \wedge A \;\Big|\; A \vee A$$

Extension 1. Full propositional intuitionistic N.D.

It scales to full NJ [Herbelin, 1995]:

$$A \ ::= \ \mathsf{p} \ \big| \ A \supset A \ \big| \ A \wedge A \ \big| \ A \vee A$$

Term assignment:

$$M \ ::= \ \lambda x.M \ \big| \ \langle M, M \rangle \ \big| \ \mathbf{inl}(M) \ \big| \ \mathbf{inr}(M) \ \big| \ \mathbf{case}\ R\ \mathbf{of}\ \langle x.M \mid x.M \rangle \ \big| \ R$$
$$R \ ::= \ x \ \big| \ R\,M \ \big| \ \pi_1(R) \ \big| \ \pi_2(R) \ \big| \ (M : A)$$

Extension 1. Full propositional intuitionistic N.D.

It scales to full NJ [Herbelin, 1995]:

$$A \ ::= \ \mathsf{p} \ \big| \ A \supset A \ \big| \ A \wedge A \ \big| \ A \vee A$$

Term assignment:

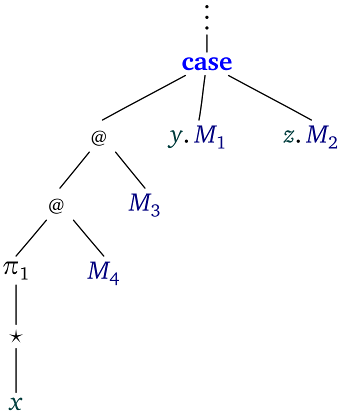
$$M \ ::= \ \lambda x.M \ \big| \ \langle M, M \rangle \ \big| \ \mathbf{inl}(M) \ \big| \ \mathbf{inr}(M) \ \big| \ \mathbf{case}\ R\ \mathbf{of}\ \langle x.M \mid x.M \rangle \ \big| \ R$$
$$R \ ::= \ x \ \big| \ R\,M \ \big| \ \pi_1(R) \ \big| \ \pi_2(R) \ \big| \ (M : A)$$

Reversed terms:

$$V \ ::= \ \lambda x.V \ \big| \ \langle V, V \rangle \ \big| \ \mathbf{inl}(V) \ \big| \ \mathbf{inr}(V) \ \big| \ x\,(S) \ \big| \ (M : A)\,(S)$$
$$S \ ::= \ V, S \ \big| \ \pi_1, S \ \big| \ \pi_2, S \ \big| \ \mathbf{case}\langle x.V \mid y.V \rangle \ \big| \ \cdot$$

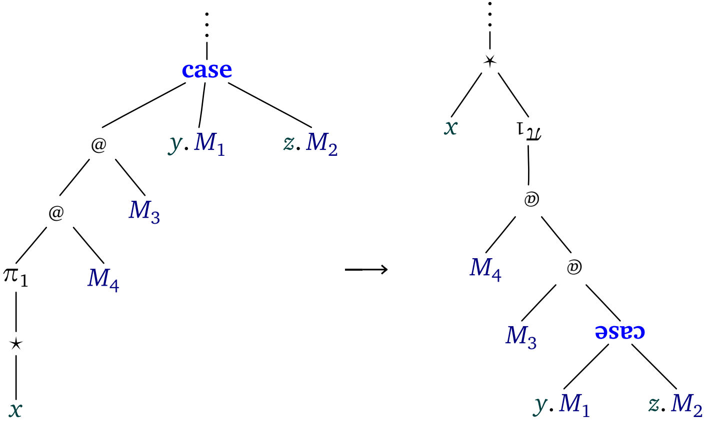
Extension 1. Full propositional intuitionistic N.D.

Example



Extension 1. Full propositional intuitionistic N.D.

Example



Extension 2. Multiplicative connectives

We can define conjunction multiplicatively [Girard et al., 1989]:

$$\dfrac{\vdash A \wedge B \downarrow \qquad \begin{matrix} [\vdash A \downarrow] \quad [\vdash B \downarrow] \\ \vdots \\ \vdash C \uparrow \end{matrix}}{\vdash C \uparrow}\ \textsc{ConjE'}$$

Extension 2. Multiplicative connectives

We can define conjunction multiplicatively [Girard et al., 1989]:

$$\dfrac{\vdash A \wedge B \downarrow \qquad \begin{matrix} [\vdash A \downarrow] \quad [\vdash B \downarrow] \\ \vdots \\ \vdash C \uparrow \end{matrix}}{\vdash C \uparrow}\ \textsc{ConjE'}$$

Term assignment:

$$\begin{aligned} M &::= \lambda x. M \;\big|\; \langle M, M \rangle \;\big|\; \textbf{let}\ \langle x, y \rangle = R\ \textbf{in}\ M \;\big|\; R \\ R &::= x \;\big|\; R\, M \end{aligned}$$

Extension 2. Multiplicative connectives

We can define conjunction multiplicatively [Girard et al., 1989]:

$$\dfrac{\vdash A \wedge B \downarrow \qquad \begin{matrix}[\vdash A\downarrow] \quad [\vdash B\downarrow] \\ \vdots \\ \vdash C \uparrow\end{matrix}}{\vdash C \uparrow}\ \textsc{ConjE'}$$

Term assignment:

$$\begin{aligned} M &::= \lambda x.M \;\big|\; \langle M, M\rangle \;\big|\; \textbf{let}\ \langle x, y\rangle = R\ \textbf{in}\ M \;\big|\; R \\ R &::= x \;\big|\; R\,M \end{aligned}$$

Reversed terms:

$$\begin{aligned} V &::= \lambda x.V \;\big|\; \langle V, V\rangle \;\big|\; x\,(S) \;\big|\; R \\ S &::= \cdot \;\big|\; V, S \;\big|\; \langle x, y\rangle.V \end{aligned}$$

Extension 2. Multiplicative connectives

We can define conjunction multiplicatively [Girard et al., 1989]:

$$\dfrac{\vdash A \wedge B \downarrow \qquad \begin{matrix} [\vdash A \downarrow] \quad [\vdash B \downarrow] \\ \vdots \\ \vdash C \uparrow \end{matrix}}{\vdash C \uparrow}\ \textsc{ConjE'}$$

$$\textsc{ConjL'}\ \dfrac{\Gamma, x : A, y : B \longrightarrow V : B}{\Gamma \mid A \wedge B \longrightarrow \langle x, y \rangle . V : C}$$

Term assignment:

$$\begin{aligned} M &::= \lambda x.M \mid \langle M, M \rangle \mid \textbf{let } \langle x, y \rangle = R \textbf{ in } M \mid R \\ R &::= x \mid R\, M \end{aligned}$$

Reversed terms:

$$\begin{aligned} V &::= \lambda x.V \mid \langle V, V \rangle \mid x\,(S) \mid R \\ S &::= \cdot \mid V, S \mid \langle x, y \rangle . V \end{aligned}$$

Extension 3. Unfocused sequent calculus

Let us add a *cut* rule to N.D. [Espírito Santo, 2007]:

$$\dfrac{\vdash A \downarrow \qquad \begin{array}{c}[\vdash A \downarrow] \\ \vdots \\ \vdash B \uparrow\end{array}}{\vdash B \uparrow}\ \textsc{Cut}$$

Extension 3. Unfocused sequent calculus

Let us add a *cut* rule to N.D. [Espírito Santo, 2007]:

$$\dfrac{\vdash A\downarrow \qquad \begin{array}{c}[\vdash A\downarrow]\\ \vdots \\ \vdash B\uparrow\end{array}}{\vdash B\uparrow}\ \textsc{Cut}$$

Term assignment:

$$\begin{aligned} M &::= x \;\big|\; \lambda x.M \;\big|\; M[x/R] \\ R &::= (M:A) \;\big|\; R\,M \end{aligned}$$

Extension 3. Unfocused sequent calculus

Let us add a *cut* rule to N.D. [Espírito Santo, 2007]:

$$\dfrac{\vdash A\downarrow \qquad \begin{array}{c}[\vdash A\downarrow]\\ \vdots \\ \vdash B\uparrow\end{array}}{\vdash B\uparrow}\ \textsc{Cut}$$

Term assignment:

$$\begin{aligned} M &::= x \;\big|\; \lambda x.M \;\big|\; M[x/R] \\ R &::= (M:A) \;\big|\; R\,M \end{aligned}$$

Reversed terms:

$$\begin{aligned} V &::= x \;\big|\; \lambda x.V \;\big|\; (V:A)(S) \\ S &::= V,S \;\big|\; x.V \end{aligned}$$

Extension 3. Unfocused sequent calculus

Let us add a *cut* rule to N.D. [Espírito Santo, 2007]:

$$\dfrac{\vdash A \downarrow \qquad \begin{array}{c}[\vdash A \downarrow]\\ \vdots \\ \vdash B \uparrow\end{array}}{\vdash B \uparrow}\ \textsc{Cut}$$

$$\textsc{Unfocus}\ \dfrac{\Gamma, x : A \longrightarrow V : B}{\Gamma \,|\, A \longrightarrow x.V : B}$$

Term assignment:

$$\begin{aligned} M &::= x \;\big|\; \lambda x.M \;\big|\; M[x/R] \\ R &::= (M : A) \;\big|\; R\,M \end{aligned}$$

Reversed terms:

$$\begin{aligned} V &::= x \;\big|\; \lambda x.V \;\big|\; (V : A)(S) \\ S &::= V, S \;\big|\; x.V \end{aligned}$$

Conclusion

- a systematic derivation of S.C.-style calculi from N.D.-style calculi, using “algebraic” CPS $\circ$ reforestation
- N.D. terms + checker $\longrightarrow$ S.C. terms + reversal + checker
- explains proof theory with compilation

Conclusion

- a systematic derivation of S.C.-style calculi from N.D.-style calculi, using "algebraic" CPS $\circ$ reforestation
- N.D. terms + checker $\longrightarrow$ S.C. terms + reversal + checker
- explains proof theory with compilation

$\rightsquigarrow$ *Gentzen was a functional programmer!*

Conclusion

- a systematic derivation of S.C.-style calculi from N.D.-style calculi, using "algebraic" CPS $\circ$ reforestation
- N.D. terms + checker $\longrightarrow$ S.C. terms + reversal + checker
- explains proof theory with compilation

$\rightsquigarrow$ *Gentzen was a functional programmer!*

Further work

- what justification for the bidirectional λ-calculus?
- what about Moggi's monadic calculus, a.k.a. LJQ?
- what about classical logic?

Conclusion

- a systematic derivation of S.C.-style calculi from N.D.-style calculi, using "algebraic" CPS $\circ$ reforestation
- N.D. terms + checker $\longrightarrow$ S.C. terms + reversal + checker
- explains proof theory with compilation

$\rightsquigarrow$ *Gentzen was a functional programmer!*

Further work

- what justification for the bidirectional λ-calculus?
- what about Moggi's monadic calculus, a.k.a. LJQ?
- what about classical logic?

Thank you!

Backup slides

Extension 4. A modal logic of necessity

We can introduce a *necessity operator*: [Pfenning and Davies, 2001]

BoxI

$$\frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A}$$

BoxE

$$\frac{\Delta; \Gamma \vdash \Box A \qquad \Delta, A; \Gamma \vdash C}{\Delta; \Gamma \vdash C}$$

Extension 4. A modal logic of necessity

We can introduce a *necessity operator*: [Pfenning and Davies, 2001]

$$\frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A}\ \text{BoxI} \qquad \frac{\Delta; \Gamma \vdash \Box A \qquad \Delta, A; \Gamma \vdash C}{\Delta; \Gamma \vdash C}\ \text{BoxE}$$

Term assignment:

$$\begin{aligned} M &::= \lambda x.M \mid \mathbf{box}(M) \mid \mathbf{let\ box}\ X = R\ \mathbf{in}\ M \mid R \\ R &::= x \mid X \mid R\,M \end{aligned}$$

Extension 4. A modal logic of necessity

We can introduce a *necessity operator*: [Pfenning and Davies, 2001]

$$\text{BoxI}\ \frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A} \qquad \text{BoxE}\ \frac{\Delta; \Gamma \vdash \Box A \qquad \Delta, A; \Gamma \vdash C}{\Delta; \Gamma \vdash C}$$

Term assignment:

$$\begin{aligned} M &::= \lambda x. M \mid \mathbf{box}(M) \mid \mathbf{let\ box}\ X = R\ \mathbf{in}\ M \mid R \\ R &::= x \mid X \mid R\, M \end{aligned}$$

Reversed terms:

$$\begin{aligned} V &::= \lambda x. V \mid \mathbf{box}(V) \mid x\,(S) \mid X\,(S) \\ S &::= \cdot \mid M, S \mid X. M \end{aligned}$$

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *PPDP*, pages 162–174. ACM, 2001. ISBN 1-58113-388-X.

José Espírito Santo. Completing Herbelin's programme. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 118–132. Springer, 2007. ISBN 978-3-540-73227-3.

Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

Hugo Herbelin. A λ-calculus structure isomorphic to Gentzen-style sequent calculus structure. In Leszek Pacholski and Jerzy Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 61–75, Kazimierz, Poland, September 1994. Springer. ISBN 3-540-60017-5.

Hugo Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*. PhD thesis, Université Paris-Diderot—Paris VII, 1995.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11 (4):511–540, 2001.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.