From Natural Deduction to the Sequent Calculus

by passing an accumulator

Matthias Puech

AARHUS UNIVERSITY

DANSAS'13

Odense, August 23, 2013

Logic can explain programs . . .

Logic can explain programs …

… and programs can explain logic

Logic can explain programs …

… and programs can explain logic

Goal of this talk: *understand the relationship between two calculi by means of functional program transformations*

From natural deduction …

$$\textsc{ImpI}\quad \dfrac{\begin{array}{c}[\vdash A]\\ \vdots \\ \vdash B\end{array}}{\vdash A \supset B} \qquad\qquad \textsc{ImpE}\quad \dfrac{\vdash A \supset B \qquad \vdash A}{\vdash B}$$

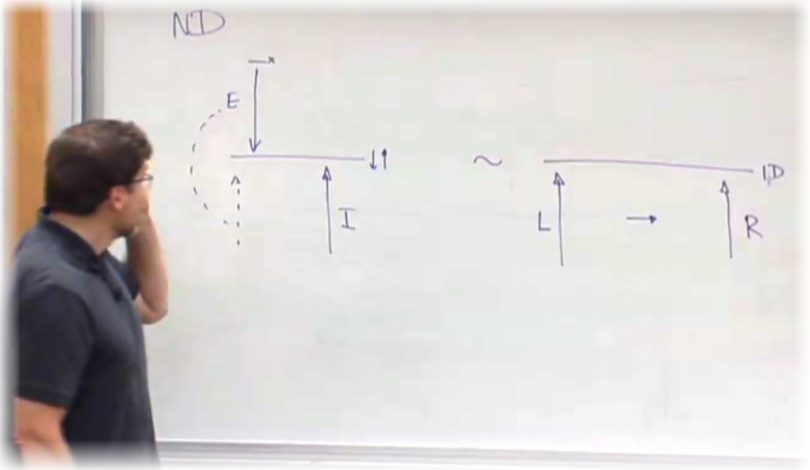
- “natural” reasoning steps
- inferences change the goal, hypotheses and “hanging”
- bidirectional reading, difficult proof search

… to the sequent calculus

$$\textsc{ImpR}\ \frac{\Gamma, A \longrightarrow B}{\Gamma \longrightarrow A \supset B} \qquad \textsc{ImpL}\ \frac{\Gamma \longrightarrow A \qquad \Gamma, B \longrightarrow C}{\Gamma, A \supset B \longrightarrow C}$$

- "fine-grained" reasoning steps
- left inferences change hypotheses
- bottom-up reading, easy proof search

Intuition



Natural deductions are “reversed” sequent calculus proofs

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

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

The message

- S.C. is an accumulator-passing N.D.
- there is a systematic *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.

In this talk

The method

Go through term assignments and reason on the type checker:

$$
\begin{tikzcd}
\text{natural deduction} \arrow[d] & \text{sequent calculus} \\
\text{canonical } \lambda\text{-calculus} \arrow[d] & \text{canonical } \bar{\lambda}\text{-calculus} \arrow[u] \\
\text{type-checker} \arrow[r, "\text{transformation}"] & \text{type-checker} \arrow[u]
\end{tikzcd}
$$

Outline

The intuition

The transformation

Some extensions

Starting point: the canonical λ-calculus [Pfenning, 2001]

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

$$R \;::=\; R\, M \;\big|\; \pi_1(R) \;\big|\; \pi_2(R) \;\big|\; x$$

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

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

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

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

Starting point: the canonical λ-calculus [Pfenning, 2001]

```
let rec check env : m × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, (x, m), (y, n)), c → let (Or (a, b)) = infer env r in
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | Atom r, Nat → let Nat = infer env r in ()
and infer env : r → a = function
  | Var x → List.assoc x env
  | App (r, m) → let (Arr (a, b)) = infer env r in
     check env (m, a); b
  | Pil r → let (And (a, _)) = infer env r in a
  | Pir r → let (And (_, b)) = infer env r in b
```

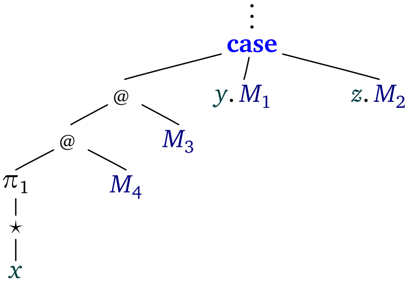
Starting point: the canonical λ-calculus [Pfenning, 2001]

```
let rec check env : m × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, (x, m), (y, n)), c → let (Or (a, b)) = infer env r in
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | Atom r, Nat → let Nat = infer env r in ()
and infer env : r → a = function
  | Var x → List.assoc x env
  | App (r, m) → let (Arr (a, b)) = infer env r in
     check env (m, a); b
  | Pil r → let (And (a, _)) = infer env r in a
  | Pir r → let (And (_, b)) = infer env r in b
```

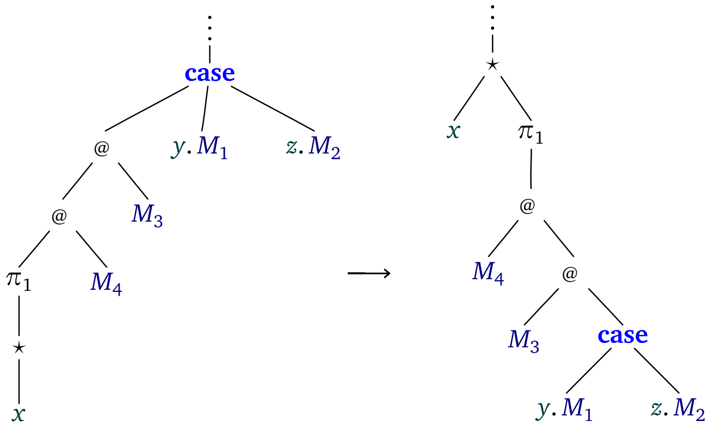
Inefficiency: no tail recursion

```
(* ... *)
  | Case (r, (x, m), (y, n)), c → let (Or (a, b)) = infer env r in
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
and infer env : r → a = function
  | Var x → List.assoc x env
  | App (r, m) → let (Arr (a, b)) = infer env r in check env (m, a); b
  | Pil r → let (And (a, _)) = infer env r in a
  | Pir r → let (And (_, b)) = infer env r in b
```

Example



Solution: reverse atomic terms



Solution: reverse atomic terms

$$
\begin{aligned}
M, N \; &::= \; \lambda x.M \;\big|\; \mathbf{inl}(M) \;\big|\; \mathbf{inr}(M) \;\big|\; M, N \;\big|\; \mathbf{case}\ R\ \mathbf{of}\ \langle x.M \mid x.M \rangle \;\big|\; R \\
R \; &::= \; R\,M \;\big|\; \pi_1(R) \;\big|\; \pi_2(R) \;\big|\; x
\end{aligned}
$$

$$\downarrow$$

$$
\begin{aligned}
V, W \; &::= \; \lambda x.V \;\big|\; \mathbf{inl}(V) \;\big|\; \mathbf{inr}(V) \;\big|\; V, W \;\big|\; x\,(S) \\
S \; &::= \; \cdot \;\big|\; V, S \;\big|\; \pi_1, S \;\big|\; \pi_2, S \;\big|\; \mathbf{case}\langle x.V \mid y.W \rangle
\end{aligned}
$$

Solution: reverse atomic terms (and introduce an accumulator)

```
let rec check env : v × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Var (x, s), c → spine env (c, List.assoc x env, s)
and spine env c : a × s → unit = function
  | And (a, _), SPil s → spine env (c, a, s)
  | And (_, b), SPir s → spine env (c, b, s)
  | Arr (a, b), SApp (m,s) → check env (m, a); spine env (c, b, s)
  | Or (a, b), SCase (x, m, y, n) →
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | c', SNil when c=c' → ()
```

Solution: reverse atomic terms (and introduce an accumulator)

```
let rec check env : v × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Var (x, s), c → spine env (c, List.assoc x env, s)
and spine env c : a × s → unit = function
  | And (a, _), SPil s → spine env (c, a, s)
  | And (_, b), SPir s → spine env (c, b, s)
  | Arr (a, b), SApp (m,s) → check env (m, a); spine env (c, b, s)
  | Or (a, b), SCase (x, m, y, n) →
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | c', SNil when c=c' → ()
```

End result: the $\bar{\lambda}$-calculus [Herbelin, 1994]

a.k.a. *spine calculus*, or LJT, or n-ary application

$$
\begin{aligned}
V, W &::= \lambda x. V \mid V, W \mid \mathbf{inl}(V) \mid \mathbf{inr}(V) \mid x\,(S) \\
S &::= \cdot \mid V, S \mid \pi_1, S \mid \pi_2, S \mid \mathbf{case}\langle x. V \mid y. W \rangle
\end{aligned}
$$

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

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

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

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

Outline

The transformation

A new application for Danvy and Nielsen [2001]'s framework:

- (partial) CPS-transformation
- defunctionalization
- reforestation

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

Step 0. the initial type-checker

```
let rec check env : m × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, (x, m), (y, n)), c → let (Or (a, b)) = infer env r in
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | Atom r, Nat → let Nat = infer env r in ()
and infer env : r → a = function
  | Var x → List.assoc x env
  | App (r, m) → let (Arr (a, b)) = infer env r in
     check env (m, a); b
  | Pil r → let (And (a, _)) = infer env r in a
  | Pir r → let (And (_, b)) = infer env r in b
```

Step 1. CPS-transformation of `infer`

```
let rec check env : m × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, (x, m), (y,n)), c → infer env r (fun (Or (a, b)) →
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c))
  | Atom r, Nat → infer env r (fun Nat → ())
and infer env : r → (a → unit) → unit = fun r s → match r with
  | Var x → s (List.assoc x env)
  | App (r, m) → infer env r (fun (Arr (a, b)) →
     check env (m, a); s b)
  | Pil r → infer env r (fun (And (a, _)) → s a)
  | Pir r → infer env r (fun (And (_, b)) → s b)
```

Step 2. Defunctionalization

```
let rec check env : m × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Case (r, (x, m), (y,n)), c → infer env r (fun (Or (a, b)) →
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)) (*SCase(x,
  | Atom r, Nat → infer env r (fun Nat → ()) (* SNil *)
and infer env : r → (a → unit) → unit = fun r s → match r with
  | Var x → s (List.assoc x env)
  | App (r, m) → infer env r (fun (Arr (a, b)) →
     check env (m, a); s b) (* SApp(m,s) *)
  | Pil r → infer env r (fun (And (a, _)) → s a) (* SPil(s) *)
  | Pir r → infer env r (fun (And (_, b)) → s b) (* SPir(s) *)
```

Step 2. Defunctionalization

```
(* spines *)
type s =
  | SPil of s
  | SPir of s
  | SApp of m × s
  | SCase of string × m × string × m
  | SNil
```

Step 2. Defunctionalization

```
let rec check env : m × a → unit = function
  (* ... *)
  | Case (r, (x, m), (y,n)), c → infer env c (SCase (x, m, y, n)) r
  | Atom r, Nat → infer env Nat SNil r
and infer env c : s → r → unit = fun s → function
  | Var x → apply env (c, List.assoc x env, s)
  | App (r, m) → infer env c (SApp (m, s)) r
  | Pil r → infer env c (SPil s) r
  | Pir r → infer env c (SPir s) r
and apply env c : a × s → unit = function
  | And (a, _), SPil s → apply env (c, a, s)
  | And (_, b), SPir s → apply env (c, b, s)
  | Arr (a, b), SApp (m,s) → check env (m, a); apply env (c, b, s)
  | Or (a, b), SCase (x, m, y, n) → check ((x, a) :: env) (m, c);
                                      check ((y, b) :: env) (n, c)
  | c', SNil when c=c' → ()
```

Step 2. Defunctionalization

```
let rec check env : m × a → unit = function
  (* ... *)
  | Case (r, (x, m), (y,n)), c → rev_spine env c (SCase (x, m, y, n)) r
  | Atom r, Nat → rev_spine env Nat SNil r
and rev_spine env c : s → r → unit = fun s → function
  | Var x → spine env (c, List.assoc x env, s)
  | App (r, m) → rev_spine env c (SApp (m, s)) r
  | Pil r → rev_spine env c (SPil s) r
  | Pir r → rev_spine env c (SPir s) r
and spine env c : a × s → unit = function
  | And (a, _), SPil s → spine env (c, a, s)
  | And (_, b), SPir s → spine env (c, b, s)
  | Arr (a, b), SApp (m,s) → check env (m, a); spine env (c, b, s)
  | Or (a, b), SCase (x, m, y, n) → check ((x, a) :: env) (m, c);
                                    check ((y, b) :: env) (n, c)
  | c’, SNil when c=c’ → ()
```

Step 3. Reforestation

Goal

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

$$\begin{array}{c}\texttt{check} \circ \texttt{rev\_spine} \circ \texttt{spine} \\ \downarrow \\ \texttt{rev\_term} \circ \texttt{check} \circ \texttt{spine}\end{array}$$

let `final_check env (m, a) = check env (rev_term m, a)`

Step 3. Reforestation

```
let rec rev_term : m → v = function
  | Lam (x, m) → VLam (x, rev_term m)
  | Pair (m, n) → VPair (rev_term m, rev_term n)
  | Inl m → VInl (rev_term m)
  | Inr m → VInr (rev_term m)
  | Case (r, x, m, y, n) →
   VHead (rev_spine r (SCase (x, rev_term m, y, rev_term n)))
  | Atom r → VHead (rev_spine r SAtom)
and rev_spine : r → s → h = fun r s → match r with
  | Var x → HVar (x, s)
  | App (r, m) → rev_spine r (SApp (rev_term m, s))
  | Pil r → rev_spine r (SPil s)
  | Pir r → rev_spine r (SPir s)
```

Step 3. Reforestation

```
let rec check env : v × a → unit = function
  | Lam (x, m), Arr (a, b) → check ((x, a) :: env) (m, b)
  | Inl m, Or (a, _) → check env (m, a)
  | Inr m, Or (_, b) → check env (m, b)
  | Pair (m, n), And (a, b) → check env (m, a); check env (n, b)
  | Var (x, s), c → spine env (c, List.assoc x env, s)
and spine env c : a × s → unit = function
  | And (a, _), SPil s → spine env (c, a, s)
  | And (_, b), SPir s → spine env (c, b, s)
  | Arr (a, b), SApp (m,s) → check env (m, a); spine env (c, b, s)
  | Or (a, b), SCase (x, m, y, n) →
     check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | c', SNil when c=c' → ()
```

Outline

Example 1. Multiplicative connectives

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

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

Example 1. Multiplicative connectives

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

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

Term assignment:

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

Example 1. Multiplicative connectives

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

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

Term assignment:

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

Reversed terms:

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

Example 2. A modal logic of necessity

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

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

Example 2. 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:

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

Example 2. 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:

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

Conclusion

- a systematic derivation of S.C.-style calculi from N.D.-style calculi, using off-the-shelf program transformations
- data type + checker $\rightarrow$ data type + reversal + checker
- works for non-canonical λ-calculus
  (but it has to be bidirectional)
- works for unfocused sequent calculus
  ($\lambda_{\mathsf{Nat}}/\lambda^{\mathsf{Gtz}}$ calculi of Espírito Santo [2007])

Conclusion

- a systematic derivation of S.C.-style calculi from N.D.-style calculi, using off-the-shelf program transformations
- data type + checker → data type + reversal + checker
- works for non-canonical λ-calculus
  (but it has to be bidirectional)
- works for unfocused sequent calculus
  ($\lambda_{\mathsf{Nat}}/\lambda^{\mathsf{Gtz}}$ calculi of Espírito Santo [2007])

Further work

- justification of bidirectional type checking
- what about Moggi's monadic calculus, a.k.a. LJQ?
- what about classical logic?

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.

Frank Pfenning. Logical frameworks. pages 1063–1147. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

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