From Natural Deduction to the Sequent Calculus by passing an accumulator

Matthias Puech

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 . . .

\[
\text{IMP}_I \quad \frac{[\vdash A]}{\vdash B} \\
\text{IMP}_E \quad \frac{\vdash A \supset B \quad \vdash A}{\vdash B}
\]

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

\[
\text{IMPR} \quad \frac{\Gamma, A \rightarrow B}{\Gamma \rightarrow A \supset B} \quad \text{IMPL} \quad \frac{\Gamma \rightarrow A \quad \Gamma, B \rightarrow C}{\Gamma, A \supset B \rightarrow C}
\]

- “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:

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

• the same in accumulator-passing style

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

(* don't forget to reverse the input list *)

```ocaml
let tower xs = tower2 1 (List.rev xs)
```
Accumulator-passing style

A well-known programmer trick to save stack space

• a function in direct style:

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

• the same in accumulator-passing style

```ocaml
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:

```ocaml
define tower1 = function
| [] -> 1
| x :: xs -> x ** tower1 xs
```

- the same in accumulator-passing style

```ocaml
define tower2 acc = function
| [] -> acc
| x :: xs -> tower2 (x ** acc) xs

(* don’t forget to reverse the input list *)
define tower xs = tower2 1 (List.rev xs)
```
In this talk

\[
\frac{\text{tower1}}{\text{tower2}} = \frac{\text{natural deduction}}{\text{sequent calculus}}
\]

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:

- **natural deduction**
  - canonical $\lambda$-calculus
    - type-checker
- **sequent calculus**
  - canonical $\bar{\lambda}$-calculus
    - type-checker

Transformation
Outline

The intuition

The transformation

Some extensions
Starting point: the canonical $\lambda$-calculus [Pfenning, 2001]

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

$$R ::= R M \mid \pi_1(R) \mid \pi_2(R) \mid x$$

$\Gamma \vdash M \Leftarrow A$ Checking

$\frac{\text{LAM}}{
\Gamma, x : A \vdash M \Leftarrow B
}{\Gamma \vdash \lambda x. M \Leftarrow A \supset B}$

$\frac{\text{АТОМ}}{
\Gamma \vdash R \Rightarrow C
}{\Gamma \vdash R \Leftarrow C}$

$\Gamma \vdash R \Rightarrow A$ Inference

$\frac{\text{VAR}}{
x : A \in \Gamma
}{\Gamma \vdash x \Rightarrow A}$

$\frac{\text{APP}}{
\Gamma \vdash R \Rightarrow A \supset B \quad \Gamma \vdash M \Leftarrow A
}{\Gamma \vdash R M \Rightarrow B}$

$\Gamma \vdash R \Rightarrow A \quad \Gamma \vdash M \Leftarrow A \quad \Gamma \vdash R \Rightarrow C$
Starting point: the canonical $\lambda$-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
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

```
\text{case} \rightarrow \star \quad \pi_1 \rightarrow \pi_1
```

```
x \quad \pi_1 \rightarrow x
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```

```
\pi_1 \rightarrow \pi_1
```

```
\ast \rightarrow \star
```
Solution: reverse atomic terms

\[ M, N ::= \lambda x. M | \text{inl}(M) | \text{inr}(M) | M, N | \text{case } R \text{ of } \langle x. M | x. M \rangle | R \]

\[ R ::= R M | \pi_1(R) | \pi_2(R) | x \]

\[ V, W ::= \lambda x. V | \text{inl}(V) | \text{inr}(V) | V, W | x(S) \]

\[ S ::= \cdot | V, S | \pi_1, S | \pi_2, S | \text{case}\langle x. V | y. W \rangle \]
Solution: reverse atomic terms (and introduce an accumulator)

```ml
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)

```haskell
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

\[
V, W ::= \lambda x. V | V, W | \text{inl}(V) | \text{inr}(V) | x(S)
\]

\[
S ::= \cdot | V, S | \pi_1, S | \pi_2, S | \text{case}(x. V | y. W)
\]

\[\Gamma \rightarrow V : A\] \hspace{1cm} \text{Right rules}

**Right rules**

**VLAM**

\[
\begin{array}{c}
\Gamma, x : A \rightarrow M : B \\
\Gamma \rightarrow \lambda x. M : A \supset B
\end{array}
\]

**HVar**

\[
\begin{array}{c}
x : A \in \Gamma \\
\Gamma | A \rightarrow S : C
\end{array}
\]

\[
\Gamma \rightarrow x(S) : C
\]

\[\Gamma | A \rightarrow S : C\] \hspace{1cm} \text{Focused left rules}

**Focused left rules**

**SApp**

\[
\begin{array}{c}
\Gamma \rightarrow V : A \\
\Gamma | B \rightarrow S : C
\end{array}
\]

\[
\Gamma | A \supset B \rightarrow V, S : C
\]

**SAtom**

\[
\begin{array}{c}
\Gamma | C \rightarrow \cdot : C
\end{array}
\]

\[
\cdots
\]
Outline

The intuition

The transformation

Some extensions
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
```
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

\[
\text{let rec check env : } m \times a \rightarrow \text{unit} = \text{function}
\]

\[
| \text{Lam} \ (x, m), \text{Arr} \ (a, b) \rightarrow \text{check} \ ((x, a) :: \text{env}) \ (m, b) \\
| \text{Inl} \ m, \text{Or} \ (a, _) \rightarrow \text{check env} \ (m, a) \\
| \text{Inr} \ m, \text{Or} \ (_, b) \rightarrow \text{check env} \ (m, b) \\
| \text{Pair} \ (m, n), \text{And} \ (a, b) \rightarrow \text{check env} \ (m, a); \text{check env} \ (n, b) \\
| \text{Case} \ (r, (x, m), (y,n)), c \rightarrow \text{infer env} \ r \ (\text{fun} \ (\text{Or} \ (a, b))) \rightarrow \\
\]  

\[
\text{check} \ ((x, a) :: \text{env}) \ (m, c); \text{check} \ ((y, b) :: \text{env}) \ (n, c)) \ (*\text{SCase}(x, y, m, n, c, r, c)*)
\]

\[
| \text{Atom} \ r, \text{Nat} \rightarrow \text{infer env} \ r \ (\text{fun} \ \text{Nat} \rightarrow () \ (*\text{SNil} *)) \\
\]

\[\text{and infer env : } r \rightarrow (a \rightarrow \text{unit}) \rightarrow \text{unit} = \text{fun} \ r \ s \rightarrow \text{match} \ r \ \text{with}
\]

\[
| \text{Var} \ x \rightarrow s \ (\text{List}.\text{assoc} \ x \ \text{env}) \\
| \text{App} \ (r, m) \rightarrow \text{infer env} \ r \ (\text{fun} \ (\text{Arr} \ (a, b))) \rightarrow \\
\]  

\[
\text{check env} \ (m, a); \text{s} \ b \ (*\text{SApp}(m,s) *)
\]

\[
| \text{Pil} \ r \rightarrow \text{infer env} \ r \ (\text{fun} \ (\text{And} \ (a, _))) \rightarrow \text{s} \ a \ (*\text{SPil}(s) *) \\
| \text{Pir} \ r \rightarrow \text{infer env} \ r \ (\text{fun} \ (\text{And} \ (_, b))) \rightarrow \text{s} \ b \ (*\text{SPir}(s) *)
\]
Step 2. Defunctionalization

(* spines *)

def type s =
    | SPil of s
    | SPir of s
    | SApp of m × s
    | SCase of string × m × string × m
    | SNil
Step 2. Defunctionalization

```haskell
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' → ()
```
Goal
Introduce intermediate data structure of reversed term $V$ to decouple reversal from checking:

\[
\text{check} \circ \text{rev\_spine} \circ \text{spine}
\]

\[
\downarrow
\]

\[
\text{rev\_term} \circ \text{check} \circ \text{spine}
\]

let $\text{final\_check} \; \text{env} \; (m, a) = \text{check} \; \text{env} \; (\text{rev\_term} \; m, a)$
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

The intuition

The transformation

Some extensions
Example 1. Multiplicative connectives

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

\[
\begin{array}{c}
[\vdash A] \\
\vdash A \wedge B \\
\vdash C \\
\vdash C
\end{array}
\]

\[\text{CONJE'}\]
Example 1. Multiplicative connectives

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

\[
\begin{array}{c}
\vdash A \\
\vdash B \\
\vdash C
\end{array}
\]

\[\vdash A \land B \quad \text{CONJE'}
\]

Term assignment:

\[
M, N ::= \lambda x.M \mid M, N \mid \text{let } x, y = R \text{ in } M \mid R
\]

\[
R ::= x \mid RM
\]
Example 1. Multiplicative connectives

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

\[
\begin{align*}
&\vdash A \\ &\vdash B \\
&\vdash A \land B & \vdash C
\end{align*}
\]

\[\text{ConjE'}\]

Term assignment:

\[
M, N ::= \lambda x. M \mid M, N \mid \text{let } x, y = R \text{ in } M \mid R \\
R ::= x \mid RM
\]

Reversed terms:

\[
V, W ::= \lambda x. V \mid V, W \mid x(S) \mid R \\
S ::= \cdot \mid M, S \mid (x,y).M
\]
Example 2. A modal logic of necessity

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

\[
\begin{align*}
\text{BoxI} & \quad \frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A} \\
\text{BoxE} & \quad \frac{\Delta; \Gamma \vdash \Box A \quad \Delta, A; \Gamma \vdash C}{\Delta; \Gamma \vdash C}
\end{align*}
\]
Example 2. A modal logic of necessity

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

\[
\begin{align*}
\text{BoxI} & \quad \Delta; \cdot \vdash A \\
& \quad \Delta; \Gamma \vdash \Box A \\
\text{BoxE} & \quad \Delta; \Gamma \vdash \Box A \\
& \quad \Delta, A; \Gamma \vdash C \\
& \quad \Delta; \Gamma \vdash C
\end{align*}
\]

Term assignment:

\[
\begin{align*}
M & ::= \lambda x.M \mid \text{box}(M) \mid \text{let box } X = R \text{ in } M \mid R \\
R & ::= x \mid X \mid RM
\end{align*}
\]
Example 2. A modal logic of necessity

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

\[
\begin{align*}
\text{BoxI} & \quad \frac{\Delta; \cdot \vdash A}{\Delta; \Gamma \vdash \Box A} \\
\text{BoxE} & \quad \frac{\Delta; \Gamma \vdash \Box A \quad \Delta, A; \Gamma \vdash C}{\Delta; \Gamma \vdash C}
\end{align*}
\]

Term assignment:

\[
\begin{align*}
M & ::= \lambda x. M \mid \text{box}(M) \mid \text{let box } X = R \text{ in } M \mid R \\
R & ::= x \mid X \mid R M
\end{align*}
\]

Reversed terms:

\[
\begin{align*}
V & ::= \lambda x. V \mid \text{box}(V) \mid x (S) \mid X (S) \\
S & ::= \cdot \mid M, S \mid X . M
\end{align*}
\]
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 $\lambda$-calculus
  (but it has to be bidirectional)
- works for unfocused sequent calculus
  ($\lambda_{Nat} / \lambda^{Gtz}$ 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 ($\lambda_{Nat}/\lambda^{Gtz}$ 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?


Frank Pfenning and Rowan Davies. A judgmental reconstruction of