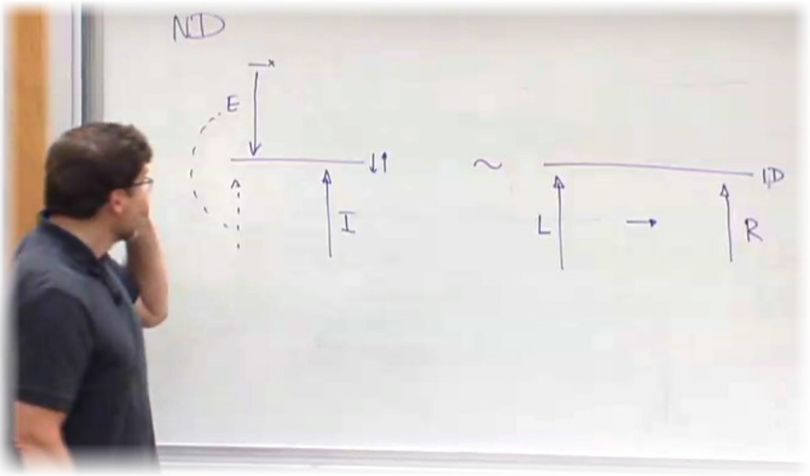
From NJ to LJ by reversing λ-terms

Matthias Puech

September 2012 — Journées PPS

From NJ to LJ



« LJ proofs are "turned-around" NJ proofs »

— Pfenning, Curien @ *OPLSS 2011*

Example: *Barbara*, bidirectionally

$$\dfrac{\dfrac{\dfrac{\dfrac{[\vdash (B \supset C)] \qquad \dfrac{[\vdash (A \supset B)] \qquad [\vdash A]}{\vdash B}\,\textsc{ImpE}}{\vdash C}\,\textsc{ImpE}}{\vdash A \supset C}\,\textsc{ImpI}}{\vdash (B \supset C) \supset A \supset C}\,\textsc{ImpI}}{\vdash (A \supset B) \supset (B \supset C) \supset A \supset C}\,\textsc{ImpI}$$

Example: *Barbara*, bidirectionally

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \longrightarrow A}\,\text{Id} \qquad \dfrac{\dfrac{}{B \longrightarrow B}\,\text{Id} \qquad \dfrac{}{C \longrightarrow C}\,\text{Id}}{B \supset C, B \longrightarrow C}\,\text{ImpL}}{A \supset B, B \supset C, A \longrightarrow C}\,\text{ImpL}}{A \supset B, B \supset C \longrightarrow A \supset C}\,\text{ImpR}}{A \supset B \longrightarrow (B \supset C) \supset A \supset C}\,\text{ImpR}}{\longrightarrow (A \supset B) \supset (B \supset C) \supset A \supset C}\,\text{ImpR}$$

Accumulator-passing style

```
let rec filter p = function
  | [] → []
  | x :: xs →
    if p x
    then x :: filter p xs
    else filter p xs
```

Accumulator-passing style

```
let rec filter p acc = function
  | [] → List.rev acc
  | x :: xs →
    if p x
    then filter p (x :: acc) xs
    else filter p acc xs
```

Accumulator-passing style

```
let rec rev_filter p acc = function
  | [] → acc
  | x :: xs →
    if p x
    then rev_filter p (x :: acc) xs
    else rev_filter p acc xs
```

Accumulator-passing style

```
let rec rev_filter p acc = function
  | [] → acc
  | x :: xs →
    if p x
    then rev_filter p (x :: acc) xs
    else rev_filter p acc xs
```

Remark

We return the *context* of the filtered list (a list too)

Accumulator-passing style

```
type α herd =
| Nil of bool
| Cons of α × α herd

let rec filter p : α herd → α herd = function
  | Nil b → Nil b
  | Cons (x, xs) →
    if p x
    then Cons (x, filter p xs)
    else filter p xs
```

Accumulator-passing style

```
type α dreh = Lin of bool × α body
and α body =
  | Top
  | Snoc of α body × α

let rec rev_filter p acc : α herd → α dreh = function
  | Nil b →  Lin (b, acc)
  | Cons (x, xs) →
    if p x
    then rev_filter p (Snoc (acc, x)) xs
    else rev_filter p acc xs
```

Accumulator-passing style

```
type α dreh = Lin of bool × α body
and α body =
  | Top
  | Snoc of α body × α

let rec rev_filter p acc : α herd → α dreh = function
  | Nil b →  Lin (b, acc)
  | Cons (x, xs) →
    if p x
    then rev_filter p (Snoc (acc, x)) xs
    else rev_filter p acc xs
```

Test

```
# rev_filter ((>=) 2) Top (Cons (1, Cons (2, Cons (3, Nil true))));;
- : int dreh = Lin (true, Snoc (Snoc (Top, 1), 2))
```

In this talk…

filter	rev_filter
NJ	$\mathsf{LJ}_{(\mathsf{T})}$
- systematic translation *natural deduction* → *sequent calculus*
- program transformation on the *canonical* type-checker
- explains bidirectional type-checking
- draw conclusion on focusing in NJ (spoiler)

In this talk...

filter	rev_filter
NJ	$LJ_{(T)}$
- systematic translation *natural deduction* → *sequent calculus*
- program transformation on the *canonical* type-checker
- explains bidirectional type-checking
- draw conclusion on focusing in NJ (spoiler)

Outline of the transformation

1. start with NJ
2. stratify syntax → normal forms
3. write its type-checker (bidirectional)
4. *reverse* atomic term syntax
   4.1 CPS-transform infer
   4.2 defunctionalize
   4.3 isolate reverse pass

1. Starting point: NJ

$$A, B \ ::= \ P \mid A \supset B \mid A \wedge B \mid A \vee B$$

$$M, N \ ::= \ x \mid \lambda x.\, M \mid M\ N \mid M, N \mid \pi_1(M) \mid \pi_2(M)$$
$$\mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \mid \mathrm{case}\ M\ \mathrm{of}\ (x.\, N_1 \mid y.\, N_2)$$

Redexes

are any matching elim ∘ intro *or* any elim ∘ ∨-elim
(commutations, otherwise no subformula property)

$$(\lambda x.\, M)\ N \tag{1}$$
$$\pi_1(M, N) \tag{2}$$
$$\pi_2(M, N) \tag{3}$$
$$\mathrm{case\ inl}(M)\ \mathrm{of}\ (x.\, N_1 \mid y.\, N_2) \tag{4}$$
$$\mathrm{case\ inr}(M)\ \mathrm{of}\ (x.\, N_1 \mid y.\, N_2) \tag{5}$$
$$(\mathrm{case}\ M\ \mathrm{of}\ (x.\, N_1 \mid y.\, N_2))\ M' \tag{6}$$
$$\pi_i(\mathrm{case}\ M\ \mathrm{of}\ (x.\, N_1 \mid y.\, N_2)) \tag{7}$$
$$\mathrm{case}\ (\mathrm{case}\ M\ \mathrm{of}\ (x_1.\, N_1 \mid x_2.\, N_2))\ \mathrm{of}\ (y_1.\, M_1 \mid y_2.\, M_2) \tag{8}$$

2. Enforce normal form

$$\begin{aligned} M, N \ ::= \ & \lambda x.\, M \mid M, N \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \\ & \mid x \mid M\ N \mid \pi_1(M) \mid \pi_2(M) \mid \text{case } M \text{ of } (x.\, N_1 \mid y.\, N_2) \end{aligned}$$

2. Enforce normal form

- no matching elim ∘ intro

$$
\begin{aligned}
M, N \;::=\;& \lambda x.\, M \mid M, N \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \\
&\mid x \mid M\; N \mid \pi_1(M) \mid \pi_2(M) \mid \mathrm{case}\; M \;\mathrm{of}\; (x.\, N_1 \mid y.\, N_2)
\end{aligned}
$$

2. Enforce normal form

- no matching elim ∘ intro

$$\begin{aligned} M, N &::= \lambda x.\, M \mid M, N \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \mid R \\ R &::= x \mid R\ M \mid \pi_1(R) \mid \pi_2(R) \mid \mathrm{case}\ R\ \mathrm{of}\ (x.\, M \mid y.\, N) \end{aligned}$$

2. Enforce normal form

- no matching elim $\circ$ intro
- no any-elim $\circ$ $\vee$-elim

$$
\begin{aligned}
M, N &::= \lambda x.\, M \mid M, N \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \mid R \\
R &::= x \mid R\ M \mid \pi_1(R) \mid \pi_2(R) \mid \text{case } R \text{ of } (x.\, M \mid y.\, N)
\end{aligned}
$$

2. Enforce normal form

- no matching elim ∘ intro
- no any-elim ∘ ∨-elim

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

2. Enforce normal form

- no matching elim ∘ intro
- no any-elim ∘ ∨-elim

$$
\begin{aligned}
M, N &::= \lambda x.\, M \mid M, N \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \mid R \\
&\quad \mid \text{case } R \text{ of } (x.\, M \mid y.\, N) && \text{Canonical terms} \\
R &::= x \mid R\ M \mid \pi_1(R) \mid \pi_2(R) && \text{Atomic terms}
\end{aligned}
$$

2. Enforce normal form

$$
\begin{aligned}
M, N &::= \lambda x.\, M \mid M, N \mid \mathrm{inl}(M) \mid \mathrm{inr}(M) \mid R \mid \text{case } R \text{ of } (x.\, M \mid y.\, N) \\
R &::= x \mid R\ M \mid \pi_1(R) \mid \pi_2(R)
\end{aligned}
$$

Lemma

Only 2 syntactic categories needed.

Proof.

Each connective can only be introduced or eliminated. □

Lemma

R have a list-like structure.

Proof.

Each elimination has only one principal premise. □

3. Write the type-checker

Lemma (Bidirectional type-checking)

1. *Given* Γ *and* R, *we can infer* A *s.t.* $\Gamma \vdash R : A$
2. *Given* Γ, M *and* A *we can check that* $\Gamma \vdash M : A$

Proof.

1. By induction on R:
   - x is inferrable,
   - the type B of the principal premise of R is inferrable (it's an R). A is a subterm of B so it's inferrable.
2. By induction on M:
   - premises of introductions are subterms of conclusion,
   - an R is inferrable, so it is checkable,
   - commuting eliminations: case-by-case.

□

3. Write the 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
```

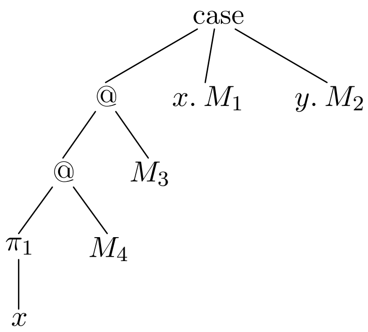
3. Write the type-checker

```
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
```

Inefficiency

```
(* ... *)
  | 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



4.1. 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)
```

4.2. 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); (*SCase(x,m,y,n)*)
                       check ((y, b) :: env) (n, c))
  | 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) *)
```

4.2. Defunctionalization

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

4.2. 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 : a × a × s → unit = function
    | c, And (a, _), SPil s → apply env (c, a, s)
    | c, And (_, b), SPir s → apply env (c, b, s)
    | c, Arr (a, b), SApp (m,s) → check env (m, a); apply env (c, b, s)
    | c, Or (a, b), SCase (x, m, y, n) → check ((x, a) :: env) (m, c);
                                         check ((y, b) :: env) (n, c)
    | Nat, Nat, SNil → ()
```

4.2. 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 : a × a × s → unit = function
    | c, And (a, _), SPil s → spine env (c, a, s)
    | c, And (_, b), SPir s → spine env (c, b, s)
    | c, Arr (a, b), SApp (m,s) → check env (m, a); spine env (c, b, s)
    | c, Or (a, b), SCase (x, m, y, n) → check ((x, a) :: env) (m, c);
                                         check ((y, b) :: env) (n, c)
    | Nat, Nat, SNil → ()
```

4.3. Commutation of the rev pass

$$\boxed{\mathsf{check} \circ \mathsf{rev\_spine} \circ \mathsf{spine} \quad \Longrightarrow \quad \mathsf{rev} \circ \mathsf{check} \circ \mathsf{spine}}$$

4.3. Commutation of the rev pass

$$\text{check} \circ \text{rev\_spine} \circ \text{spine} \quad \Longrightarrow \quad \text{rev} \circ \text{check} \circ \text{spine}$$

```
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 : a × a × s → unit = function
  | c, And (a, _), SPil s → spine env (c, a, s)
  | c, And (_, b), SPir s → spine env (c, b, s)
  | c, Arr (a, b), SApp (m,s) → check env (m, a); spine env (c, b, s)
  | c, Or (a, b), SCase (x, m, y, n) →
      check ((x, a) :: env) (m, c); check ((y, b) :: env) (n, c)
  | Nat, Nat, SNil → ()
```

CPS ∘ defunctionalization reverses a data structure

S are the *contexts/zippers* of R (see Danvy & Nielsen, 2001)

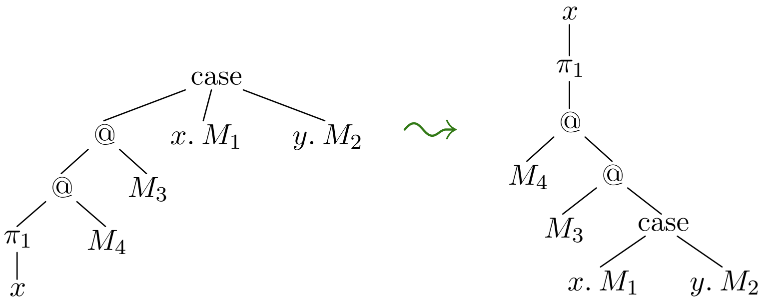
```
type m =
  | Lam of string × m
  | Inl of m
  | Inr of m
  | Pair of m × m
  | Case of r × string × m × string × m
  | Atom of r
and r =
  | Var of string
  | App of r × m
  | Pil of r
  | Pir of r
```

```
type v =
  | Lam of string × v
  | Inl of v
  | Inr of v
  | Pair of v × v
  | Var of string × s
and s =
  | SPir of s
  | SPil of s
  | SApp of v × s
  | SCase of string × v × string × v
  | SNil
```

CPS ∘ defunctionalization reverses a data structure

S are the *contexts*/*zippers* of R (see Danvy & Nielsen, 2001)

Example



What is this system?

$$
\begin{array}{rcl}
V, W & ::= & \lambda x.\, V \mid V, W \mid \mathrm{inl}(V) \mid \mathrm{inr}(V) \mid x(S) \\
S & ::= & V; S \mid \pi_1; S \mid \pi_2; S \mid \mathrm{case}(x.\, V \mid y.\, W) \mid \cdot
\end{array}
$$

What is this system? LJT/$\bar{\lambda}$ (Herbelin, 1995)

$$V, W \;::=\; \lambda x.\, V \mid V, W \mid \mathrm{inl}(V) \mid \mathrm{inr}(V) \mid x(S)$$
$$S \;::=\; V; S \mid \pi_1; S \mid \pi_2; S \mid \mathrm{case}(x.\, V \mid y.\, W) \mid \cdot$$

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

...

FOCUS
$$\frac{x : A \in \Gamma \qquad \Gamma; A \vdash S : C}{\Gamma \vdash x(S) : C}$$

$\boxed{\Gamma; A \vdash S : C}$ Focused left rules

IMPL
$$\frac{\Gamma \vdash V : A \qquad \Gamma; B \vdash S : C}{\Gamma; A \supset B \vdash V; S : C}$$

CONJL1
$$\frac{\Gamma; A \vdash S : C}{\Gamma; A \wedge B \vdash \pi_1; S : C}$$

DISJL
$$\frac{\Gamma, x : A \vdash V : C \qquad \Gamma, y : B \vdash W : C}{\Gamma; A \vee B \vdash \mathrm{case}(x.\, V \mid y.\, W) : C}$$

ID
$$\frac{}{\Gamma; P \vdash \cdot : P}$$

Moral of the story

Theorem

- *types* NJ.m *and* LJT.m *are isomorphic*
- NJ.check env m *iff* LJT.check env (rev m)

Proof.

by construction. □

Moral of the story

Theorem

- *types* NJ.m *and* LJT.m *are isomorphic*
- NJ.check env m *iff* LJT.check env (rev m)

Proof.

by construction. □

Lessons learned

- LJT type-checkers have no infer mode
- reversed NJ is LJT, not LJ
- so NJ was already "focused"

Moral of the story

Theorem

- *types* NJ.m *and* LJT.m *are isomorphic*
- NJ.check env m *iff* LJT.check env (rev m)

Proof.

by construction. □

Lessons learned

- LJT type-checkers have no infer mode
- reversed NJ is LJT, not LJ
- so NJ was already "focused"

Open questions

- does it scale to your favorite N.D.-style calculus?
- in particular NK? (adding e.g. call/cc)
- what is an unfocused NJ?