Proofs, upside down

A functional correspondence between natural deduction and the sequent calculus

Matthias Puech

Department of Computer Science, Aarhus University, Denmark*

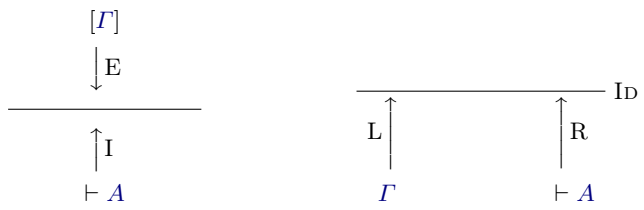
Abstract. It is well known in proof theory that sequent-calculus proofs differ from natural deduction proofs by "reversing" elimination rules upside down into left introduction rules. It is also well known that to each recursive, functional program corresponds an equivalent iterative, accumulator-passing program, where the accumulator stores the continuation of the iteration, in "reversed" order. Here, we compose these remarks and show that a restriction of the intuitionistic sequent calculus, LJT, is exactly an accumulator-passing version of intuitionistic natural deduction NJ. More precisely, we obtain this correspondence by applying a series of off-the-shelf program transformations *à la* Danvy et al. on a type checker for the bidirectional λ-calculus, and get a type checker for the $\bar{\lambda}$-calculus, the proof term assignment of LJT. This functional correspondence revisits the relationship between natural deduction and the sequent calculus by systematically deriving the rules of the latter from the former, and allows us to derive new sequent calculus rules from the introduction and elimination rules of new logical connectives.

1 Introduction

A typical introductory course to proof theory starts by presenting the two calculi introduced by Gentzen [9]: first *natural deduction*, that defines the meaning of each logical connective by its introduction and elimination rules, and then the *sequent calculus*, an equivalent refinement of the latter that makes it easier to search for proofs. Natural deductions admit a *bidirectional* reading: introduction rules are read bottom-up, from the conclusion, and elimination rules are read top-down, from the hypotheses. Sequent calculus is then presented as a response to this cumbersome bidirectionality, by turning all elimination subproofs upside down (Fig. 1): introductions are renamed "right rules", upside-down eliminations become "left rules", and they operate directly on formulae in the environment Γ, instead of operating on the goal of their premises.

This kind of inversion of control is performed routinely by functional programmers: a piece of data traversed recursively might be turned upside down

* This work was carried out while the author was at Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, PiR2, INRIA Paris-Rocquencourt, F-75205 Paris, France.



(a) natural deduction (b) sequent calculus

Fig. 1: From natural deductions to sequent-calculus proofs

to be traversed iteratively. For instance, a recursive function computing the exponent tower of a list (Fig. 2) can be transformed in a tail-recursive function carrying an accumulator; only, in this case, the list must be reversed first (because exponentiation is not commutative) and the value for the base case, here 1, must be passed at the top level. Not only are these two programs equivalent, but one can always derive the second from the first.

The analogy is even clearer from the other side of the Curry-Howard looking glass. It is well known since Herbelin's work [10,11] that a restriction of intuitionistic sequent calculus LJ named LJT can be viewed as a type system for the $\bar{\lambda}$- or *spine*-calculus, a language in which consecutive eliminations are reversed with respect to the usual λ-calculus, the variable case being accessible at the top level. Recently, Espírito Santo lifted the restriction [7], and presented two isomorphic calculi corresponding to full LJ. Both authors posed a pair of calculi, respectively in natural-deduction and sequent-calculus style, and showed how to translate a given term from one to the other.

We propose here a method to systematically derive, not a particular proof, but the *inference system* of an intuitionistic sequent calculus itself from the rules of an intuitionistic natural deduction, by means of only off-the-shelf program transformations, in the style of Danvy and colleagues [1]: take NJ, presented as a recursive type-checking program for the bidirectional λ-calculus, turn it into an equivalent accumulator-passing style program, and you will get a type checker for $\bar{\lambda}$-terms, which are notations for LJT proofs. In other words, we show that LJT is precisely to NJ what `tower_acc` is to `tower_rec`. We conclude that, in the light of functional-programming techniques, we can reinterpret Gentzen's discovery of sequent calculus as a "compilation" of natural deduction.

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

(a) a recursive function

```
let rec tower_acc acc = function
  | [] → acc
  | x :: xs → tower_acc (x ** acc) xs
let tower xs = tower_acc 1 (List.rev xs)
```

(b) in accumulator-passing style

Fig. 2: From a recursive function to its accumulator-passing equivalent

In Section 2, we present this transformation step-by-step, in OCaml:

- *CPS transformation*, showing that eliminations are head recursive,
- *lightweight defunctionalization*, reifying the continuations into *spines*,
- *reforestation*, decoupling the checking of a term from its reversal, and introducing an intermediate data structure: $\bar{\lambda}$-terms.

For the sake of conciseness, this transformation is performed in NJ with a restricted set of connectives; in Section 3, we show that it is modular, i.e., that it applies to richer situations, by exhibiting example extensions.

2 The Transformation

2.1 NJ and the Bidirectional λ-calculus

The starting point of our transformation is a standard type checking algorithm for NJ proofs of propositions built out of the following connectives (this choice is discussed in the next section):

$$A, B \ ::= \ A \supset B \mid A \vee B \mid A \wedge B \mid \mathsf{p}$$

The terms we assign to NJ proofs are not however those of the usual λ-calculus, but of a *bidirectional* extension of it [16,3] (Fig. 3). Bidirectional typing was devised initially as a method for partial type inference. The idea is to judge differently two classes of λ-terms, those *checkable* (whose type is supposed to be an input of the type-checking algorithm) and those *inferable* (whose type can be synthesized by the algorithm).

This distinction can be reflected back syntactically by *stratifying* the syntax of terms (as in, e.g., [14]) into two categories: general *terms* M, N, whose checking requires to know their type, and *atomic terms* R, whose type can be synthesized. An atomic term is a term, since if we can infer its type, we can check that it is equal to a given type, hence the coercion from M to R and rule Atom.[1] Dually, every term can be made atomic provided we are given its expected type, hence the typing annotation construct $(M : A)$ and rule Annot. Variables are inferable since their type can be read off the environment. Eliminations are too, provided their principal premise is inferable, and its conclusion is a subterm of it; all other constructs are "only" general terms. Since the λ-abstractions are checked, they do not require type annotations (rule Lam; this omission was the original motivation of bidirectional type checking).

This stratification has another interpretation: one can see the bidirectional calculus as a reorganization of the syntax of the λ-calculus concentrating on *redexes*. In the λ-calculus, redexes are the combination of matching introductions and eliminations. Here, we restrict principal premises of eliminations to

[1] Often, you will find this rule restricted to atomic types, e.g., in [3], which ensures η-long canonicity. We are not concerned by this restriction here.

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

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

$$\textsc{Lam}\ \frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow A \supset B} \qquad \textsc{Inl}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash \mathbf{inl}(M) \Leftarrow A \vee B}$$

$$\textsc{Inr}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash \mathbf{inr}(M) \Leftarrow A \vee B} \qquad \textsc{Pair}\ \frac{\Gamma \vdash M \Leftarrow A \qquad \Gamma \vdash N \Leftarrow B}{\Gamma \vdash \langle M, N\rangle \Leftarrow A \wedge B} \qquad \textsc{Atom}\ \frac{\Gamma \vdash R \Rightarrow C}{\Gamma \vdash R \Leftarrow C}$$

$$\textsc{Case}\ \frac{\Gamma \vdash R \Rightarrow A \vee B \qquad \Gamma, x : A \vdash M \Leftarrow C \qquad \Gamma, y : B \vdash N \Leftarrow C}{\Gamma \vdash \mathbf{case}\ R\ \mathbf{of}\ \langle x.\, M \mid y.\, N\rangle \Leftarrow C}$$

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

$$\textsc{Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Pil}\ \frac{\Gamma \vdash R \Rightarrow A \wedge B}{\Gamma \vdash \pi_1(R) \Rightarrow A} \qquad \textsc{Pir}\ \frac{\Gamma \vdash R \Rightarrow A \wedge B}{\Gamma \vdash \pi_2(R) \Rightarrow B}$$

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

Fig. 3: The bidirectional NJ/λ-calculus

be other eliminations or variables, creating no redexes, or type annotation.[2] Consequently, to construct a bidirectional term with a redex, we *must* use the annotation, for instance $(\lambda x.\, M : A \to B)\ N$. Conversely, a term that does not use this construct is canonical; such a term can be seen as a notation for *intercalations* [19]. Note that there are more non-canonical bidirectional terms than there are equivalent λ-terms [6], since we can always add type annotations, e.g., $(x\ M : A \supset B)\ N$ instead of $x\ M\ N$. Note also that an atomic term has no more than one direct atomic subterm, since an elimination has no more than one principal premise; hence, we will sometimes call them *chains of eliminations*. A (general) term which has a direct atomic subterm, i.e., a coercion R or an elimination $\mathbf{case}\ R\ \mathbf{of}\ \langle x.\, M \mid y.\, N\rangle$, will be called a *full chain*.

Figure 4 is a transliteration of this algorithm into OCaml, a metalanguage more suitable for program transformations. For concision, we use pattern-matching

[2] Note that $\vee$-eliminations are *not* allowed as principal subterms of an elimination, since they could "hide" a redex (a *commutative cut*). The same remark would apply to e.g., $\bot$ or $\exists$.

```
type a = At | Imp of a × a | And of a × a | Or of a × a
type var = string
type env = (var × a) list
type m = Lam of var × m | Pair of m × m | Inl of m | Inr of m
 | Case of r × var × m × var × m | Atom of r
and r = App of r × m | Pil of r | Pir of r | Var of var | Annot of m × a

let rec check env c : m → unit =
  let rec infer : r → a = function
  | 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
  | Pil r → let (And (a, _)) = infer r in a
  | Pir r → let (And (_, b)) = infer r in b
  in fun m → match m, c with
  | Lam (x, m), Imp (a, b) → check ((x, a) :: env) b m
  | Pair (m, n), And (a, b) → check env a m; check env b n
  | Inl m, Or (a, _) → check env a m
  | Inr n, Or (_, b) → check env b n
  | Case (r, x, m, y, n), c → let (Or (a, b)) = infer r in
    check ((x, a) :: env) c m; check ((y, b) :: env) c n
  | Atom r, c → match infer r with c' when c=c' → ()
```

Fig. 4: Initial program, i.e., Fig. 3 in OCaml (module Initial)

failure to signal a typing error. Function `infer` is written in lambda-dropped form, to emphasize that its recursive calls are in the scope of the same environment `env` and expected type `c`. It is only called in non tail-recursive position: its code begins by recursively descending all the way to the bottom of a chain of eliminations. Only then does it synthesize the type of the atomic term, "on the way back". Alternatively, we could traverse this chain in reverse order, accumulating the synthesized types. It is precisely what we embark on doing.

2.2 CPS Transformation

The first two steps of our transformation could be considered a unique, compound one called "algebraic CPS transform" since its popularization by Danvy et al. [5,1]. Its goal is to turn the recursive program above—it needs a stack, implicit in the metalanguage, to store intermediate results—into a deterministic state transition system, a simpler metalanguage where this stack is *reified*. Here however, we only perform this transformation selectively, on function `infer` but not on `check`, since we are only interested in reversing atomic terms.

The first step is to turn every call to `infer` into a tail-recursive one, by applying Plotkin's standard call-by-value CPS transformation [17] (we show only the modified lines).

```
let rec check env c : m → unit =
  let rec infer : r → (a → unit) → unit = fun r s → match r with
  | Var x → s (List.assoc x env)
  | Annot (m, a) → check env a m; s a
  | App (r, m) → infer r (fun (Imp (a, b)) → check env a m; s b)
  | Pil r → infer r (fun (And (a, _)) → s a)
  | Pir r → infer r (fun (And (_, b)) → s b)
  in fun m → (* ... *)
  | Case (r, x, m, y, n), c → infer r (fun (Or (a, b)) →
      check ((x, a) :: env) c m; check ((y, b) :: env) c n)
  | Atom r, c → infer r (function c' when c=c' → ())
```

We add an extra functional argument `s` to `infer`, which is called with its result; recursive calls "chain up" the computation to be done at return time. Consequently, all calls to `infer` are tail calls. Note the answer type of `infer`: it is fixed by the return type of `check`, which is `unit`. All calls to `infer` are done directly after pattern-matching: the function is *head recursive* (doing all the work "on the way back").

The CPS transformation trades one feature of the metalanguage—the ability to store intermediate results on a stack—into another—the ability to have functions as first-class values. Yet, in what follows, we map back such a higher-order program into a first-order one.

2.3 Lightweight defunctionalization

The second step is a variant of *defunctionalization*, as showcased by Danvy and Nielsen [5], which takes a program with first-class functions and returns an equivalent one where these functions have been reified into purely first-order data. The idea is to replace every such inner function by a unique identifier (a type constructor in our case), and all application of a functional variable `f` by `apply f`, where `apply` is a "dictionary" mapping identifiers to the function they stand for. Each inner function can have free variables, so each constructor needs to be parameterized by the values of these free variables. *Lightweight* defunctionalization [2] restricts this set of parameters: free variables that are in scope of both introduction and elimination of functions do not need to be parameters. In our case, both `env` and `c` are "constant" throughout all recursive calls to `infer`, and need not be saved in constructors.

We thus introduce the type `s` of *spines*[3]: we call SCase, SAtom, SApp, SPil and SPir the respective continuations of the previous program. Then we transform our program accordingly, introducing function `apply`:

```
type s =
  | SAtom
  | SPil of s
  | SPir of s
```

[3] We motivate the choice of this name in Section 2.5.

```
  | SCase of var × m × var × m
  | SApp of m × s

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

We uncurried `apply` on-the-fly for legibility. Note that defunctionalization preserves tail calls: function `infer` is still tail-recursive.

Because all inner functions transformed stemmed from CPS, `s` is the type of *reified continuations*. Because the original type checker traverses the whole term structure, these can be seen as the type of *zippers* [12] or *contexts* of atomic terms: a pair $(r, s) : r \times s$ determines uniquely an atomic position inside an atomic term $S[R]$. CPS and defunctionalization decomposed the recursive process in two parts: what is done "on the way down" of an atomic term traversal (function `infer`), accumulating a continuation `s`, and what is done "on the way back", (function `apply`), reading off this continuation in reverse order. Since `infer` was head recursive, our transformed `infer` is a simple *reversal* function that takes an atom to a spine, and eventually calls `apply` with this spine. Function `apply` now actually performs the type synthesis; the impatient reader can already interpret this function as the second judgment of Fig. 7, but a final step is needed to reach our target.

2.4 Reforestation

This type checker is a strange hybrid: given a term `m`, it checks its type until arriving to a full chain (`check`), which it reverses into a spine `s` (`infer`), which in turn is type-checked (`apply`). In the last step, we decouple completely reversal and checking so that, given a term `m`, we can first reverse it completely, and only then check its type. The transformation comprises two *reforestation* steps. Reforestation is the inverse of Wadler's *deforestation* [20]: instead of eliminating intermediate data structure for efficiency by "chaining up" function calls, we

reintroduce an intermediate data structure of reversed terms from "chained up" function calls.

The first reforestation concerns `infer`: it "lifts up" the computation done in its base cases (Var and Annot) outside it, at its call sites. To this end, `infer` needs to return an intermediate data structure, that we call a *head* `h`, representing algebraically the computation to be done in these two base cases, each parameterized by their free variables:

```
type h =
  | HVar of var × s
  | HAnnot of m × a × s
```

A new `head` function is introduced, that plays the same role as `apply` in the defunctionalization step: it maps an "algebraic base case" `h` to the computation it stands for. Previous call sites to `infer` now perform the composition of the new `infer` and `head` functions. It reads:

```
(* ... *)
  let head : h → unit = function
    | HVar (x, s) → apply (s, List.assoc x env)
    | HAnnot (m, a, s) → check env a m; apply (s, a) in
  let rec infer : r → s → h = fun r s → match r with
    | Var x → HVar (x, s)
    | Annot (m, a) → HAnnot (m, a, s)
(* ... *) in fun m → (* ... *)
    | Case (r, x, m, y, n), c → head (infer r (SCase (x, m, y, n)))
    | Atom r, c → head (infer r SAtom)
```

In Wadler's words, this program is not in "treeless form", because of these function compositions. Applying deforestation to it, we would get back the program of the last section. The type `h` of heads represents *reversed full chains*: if we were to construct a full chain out of thread and pearls (Fig. 5), reversing it would amount to hold it, not by its top-level node (a Case or an Atom) but by its bottom node (a Var or an Annot) and letting all nodes hang loose underneath; the whole atomic spine would be reversed, top-level nodes becoming bottom nodes (SCase and SAtom) and bottom nodes becoming top-level nodes (HVar and HAnnot).

Interleaved checking and reversal are still not completely decoupled, so we perform one final reforestation, on function `check`. Again, we "lift up" the calls to `infer` in the base cases (Case and Atom) outside of `check`, at the top level. To this end, we introduce the intermediate data structure `v` of reversed terms, on the model of `m` but replacing constructors Case and Atom by a unique VHead constructor. Since `check`, `head` and `spine` are mutually recursive, so are the final types `v`, `h` and `s`. Function `check` is decomposed in two passes, one taking an `m` to an intermediate `v` (`rev`), and one actually checking the resulting `v` (`check`). The resulting code is shown on Fig. 6. For better readability, we renamed `infer` into `rev_spine`, and `apply` into `spine`. Again, if we deforest this program, the

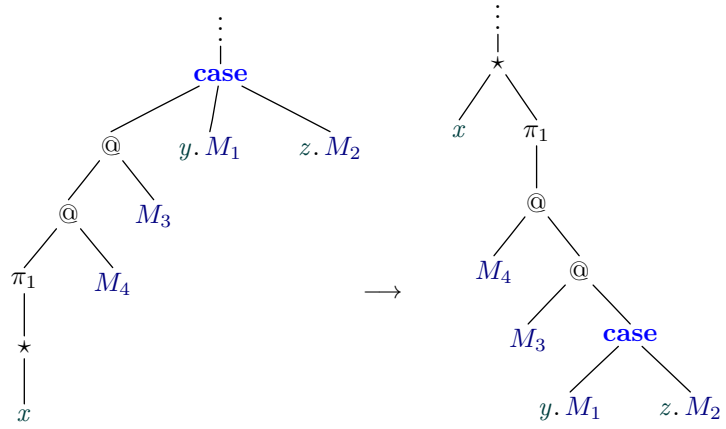


Fig. 5: Reversing a full chain into a head

intermediate data structure `v` vanishes, and both passes are merged into one and we get back the previous type checker.

The transformation is now over, and our goal is achieved: the resulting program `check` is the composition of reversal `rev` and checking `check`. Checking is more space-efficient than the original one, because we made function `infer` tail-recursive, with an accumulator storing the synthesized type. The top-level function is observationally equivalent to the original program: we traded stack space for an intermediate data structure of "reversed" terms.

Theorem 1. `Initial.check env c m` *is defined iff* `Final.check env c m` *is defined.*

Proof. By composition of the soundness of the transformations.

However, decoupling these two phases offers the opportunity to study the intermediate data structure unveiled.

2.5 LJT and the $\bar{\lambda}$-calculus

Let us transliterate back data structures `v`, `h` and `s` and functions `check`, `head` and `spine` into BNF syntax and inference rules, a metalanguage more suitable for logical interpretation. Fig. 7 presents this system[4]. It is precisely the $\bar{\lambda}$-calculus of Herbelin [10,11], a proof term assignment for LJT. LJT is a restriction of the sequent calculus LJ, with features of *focusing* [13].

Reversed terms V contain all introductions, and two "structural" constructs: variables and type annotations, both attached to a *spine* S of eliminations. This spine is terminated by a $\cdot$ ("nil"), or a **case** construct. When restricted

[4] With two small differences: we inlined type `h` and function `head` for compactness, and lambda-lifted all inner functions.

```
type v = VLam of var × v | VPair of v × v | VInl of v | VInr of v | VHead of h
and h = HVar of var × s | HAnnot of v × a × s
and s = SApp of v × s | SPil of s | SPir of s | SAtom | SCase of var × v × var × v

let check env c : m → unit =
  let rec rev_spine : r → s → h = fun r s → match r with
    | Var x → HVar (x, s)
    | Annot (m, a) → HAnnot (rev m, a, s)
    | App (r, m) → rev_spine r (SApp (rev m, s))
    | Pil r → rev_spine r (SPil s)
    | Pir r → rev_spine r (SPir s)
  and rev : m → v = function
    | Lam (x, m) → VLam (x, rev m)
    | Pair (m, n) → VPair (rev m, rev n)
    | Inl m → VInl (rev m)
    | Inr n → VInr (rev n)
    | Case (r, x, m, y, n) → VHead (rev_spine r (SCase (x, rev m, y, rev n)))
    | Atom r → VHead (rev_spine r SAtom) in
  let rec check env c : v → unit =
    let rec spine : s × a → unit = function
      | SApp (m, s), Imp (a, b) → check env a m; spine (s, b)
      | SPil s, And (a, _) → spine (s, a)
      | SPir s, And (_, b) → spine (s, b)
      | SCase (x, m, y, n), Or (a, b) → check ((x, a) :: env) c m; check ((y, b) :: env) c n
      | SAtom, c' when c=c' → () in
    let head : h → unit = function
      | HVar (x, s) → spine (s, List.assoc x env)
      | HAnnot (m, a, s) → check env a m; spine (s, a) in
    fun v → match v, c with
      | VLam (x, v), Imp (a, b) → check ((x, a) :: env) b v
      | VPair (v, w), And (a, b) → check env a v; check env b w
      | VInl v, Or (a, _) → check env a v
      | VInr w, Or (_, b) → check env b w
      | VHead h, c → head h
  in fun m → check env c (rev m)
```

Fig. 6: Final program, i.e., Fig. 7 in OCaml (module Final)

to its implicative fragment, this calculus is sometimes called a *spine calculus* [3]: applications can be viewed as n-ary[5], i.e., applied to a list of arguments $f\,(M_1, M_2, \ldots, M_n, \cdot)$, in contrast with the usual $(((f\ M_1)\ M_2)\ \ldots)\ M_n$ of NJ. Espiríto Santo [7] speaks of a difference of *associativity* of application. Generalizing to our extended fragment, the situation is more subtle: an elimination chain is piled up in reverse order, and its head construct, a variable or an annotated

[5] Note that partial application is still possible, since the length of this list can vary and the return type C in SAtom can be an arrow.

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

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

$$\textsc{VLam}\ \frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash \lambda x.\, M \Leftarrow A \supset B} \qquad \textsc{VPair}\ \frac{\Gamma \vdash M \Leftarrow A \qquad \Gamma \vdash N \Leftarrow B}{\Gamma \vdash \langle M, N\rangle \Leftarrow A \wedge B}$$

$$\textsc{VInl}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash \mathbf{inl}(M) \Leftarrow A \vee B} \qquad \textsc{VInr}\ \frac{\Gamma \vdash M \Leftarrow B}{\Gamma \vdash \mathbf{inr}(M) \Leftarrow A \vee B}$$

$$\textsc{HVar}\ \frac{x : A \in \Gamma \qquad \Gamma \mid A \vdash S \Leftarrow C}{\Gamma \vdash x\,(S) \Leftarrow C} \qquad \textsc{HAnnot}\ \frac{\Gamma \vdash M \Leftarrow A \qquad \Gamma \mid A \vdash S \Leftarrow C}{\Gamma \vdash (M : A)\,(S) \Leftarrow C}$$

$\boxed{\Gamma \mid A \vdash S \Leftarrow C}$ Focused left rules

$$\textsc{SApp}\ \frac{\Gamma \vdash V \Leftarrow A \qquad \Gamma \mid B \vdash S \Leftarrow C}{\Gamma \mid A \supset B \vdash V, S \Leftarrow C} \qquad \textsc{SPil}\ \frac{\Gamma \mid A \vdash S \Leftarrow C}{\Gamma \mid A \wedge B \vdash \pi_1, S \Leftarrow C}$$

$$\textsc{SPir}\ \frac{\Gamma \mid B \vdash S \Leftarrow C}{\Gamma \mid A \wedge B \vdash \pi_1, S \Leftarrow C} \qquad \textsc{SCase}\ \frac{\Gamma, x : A \vdash V \Leftarrow C \qquad \Gamma, y : B \vdash W \Leftarrow C}{\Gamma \mid A \vee B \vdash \mathbf{case}\langle x.\, V \mid y.\, W\rangle \Leftarrow C}$$

$$\textsc{SAtom}\ \frac{}{\Gamma \mid C \vdash \cdot \Leftarrow C}$$

Fig. 7: The LJT/$\bar{\lambda}$-calculus [10]

term that was buried under eliminations, is brought back at the top level. For example, the full chain $\mathbf{case}\ \pi_1(f\ x)\ \mathbf{of}\ \langle x_1.\, M_1 \mid x_2.\, M_2\rangle$ is now written with the head variable f first: $f\,(x\,(\cdot), \pi_1, \mathbf{case}\langle x_1.\, M_1 \mid x_2.\, M_2\rangle, \cdot)$.

As promised, the typing rules are in Curry-Howard correspondence with a sequent calculus-like system. Like the rules of Fig. 3, they come in two judgments; unlike them, no judgment infers a type: both are in checking mode. This fact is a notable difference with the usual definition of spine-form calculi [3]: their restriction to negative connectives makes possible to infer the types of spines, which is impossible when extended with e.g., disjunction. The right rules are unchanged with respect to Fig. 3, except for two new rules: HVar, sometimes called Focus, which focuses on a particular premise (variable) and checks the attached spine, and HAnnot, which corresponds to the usual Cut rule. In "focused mode", all rules act on a distinguished premise A (the *stoup*) hence their

names: left rules. Once focused on a premise, these rules oblige us to continue working on it until we can either close the branch by SAtom (usually called Init) or by a "polarity switch", i.e., here when the stoup contains a disjunction. Tracing the stoup back through the transformations, it corresponds to the *accumulator* threaded in `spine` on Fig. 6, which was the returned type of function `infer` on Fig. 4. In other words, the focused hypothesis of a left rule in LJT corresponds to the principal premise of an elimination in NJ.

3 Extensions

Although we chose to start with a reduced set of logical connectives ($\wedge$, $\vee$, $\supset$), the same scheme extends to all connectives of intuitionistic predicate logic: $\top$, $\bot$, $\forall$, $\exists$, as well as variants of these and related systems.

3.1 Multiplicative Connectives

For instance, taking the *multiplicative* definition of the conjunction via the unique elimination:

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

leads to the following normal term assignment (showing only the $\wedge$, $\supset$ fragment):

$$\begin{array}{rcl} M, N & ::= & \lambda x.\, M \mid \langle M, N\rangle \mid \mathbf{let}\ \langle x, y\rangle = R\ \mathbf{in}\ M \mid R \\ R & ::= & x \mid (M : A) \mid R\ M \end{array}$$

Note that the **let** construct is a general term, for the same reason the **case** construct was in Section 2.1. Applying the same transform, we get the following term assignment:

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

where the top-level **let** gets buried under the chain of eliminations, and the corresponding checking rule:

$$\frac{\begin{array}{c}\textsc{ConjL'} \\ \Gamma, x : A, y : B \vdash M \Leftarrow C\end{array}}{\Gamma \mid A \wedge B \vdash \langle x, y\rangle.\, M \Leftarrow C}$$

which is the usual left rule of the multiplicative conjunction. Note that the premise loses the focus on the hypothesis, just like for disjunction. The same system was proposed by Herbelin in his PhD thesis [11].

3.2 Modal Logic of Necessity

Pfenning and Davies [15] propose a reconstruction of modal logic in terms of the Gentzen apparatus. They present the necessity modality $\Box\, A$ (denoting the necessity for A to be true *under no hypotheses*) as a connective defined by the following introduction and elimination rules:

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

The environment is split in two sets: Γ and Δ, *resp.* the true and the necessarily true assumptions. To use a necessary hypothesis, we add rule:

$$\textsc{Meta}\ \frac{A \in \Delta}{\Delta; \Gamma \vdash A}$$

The authors also propose a term assignment for these rules, that we easily make bidirectional by stratification (again, the $\supset$, $\Box$ fragment):

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

Note the new set of *metavariables* $\mathtt{X}$ referring to necessary hypotheses. Again, applying our transformation, we get the following reversed syntax:

$$\begin{array}{rcl} V & ::= & \lambda x.\, V \mid \mathbf{box}(V) \mid x\,(S) \mid \mathtt{X}\,(S) \mid (M : A)\,(S) \mid R \\ S & ::= & \cdot \mid M, S \mid \mathtt{X}.\, M \end{array}$$

The associated rules for the new syntactic constructs are the left and right rules for necessity, and a focus rule for necessary hypotheses:

$$\textsc{BoxR}\ \frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \mathbf{box}(M) : \Box\, A} \qquad \textsc{BoxL}\ \frac{\Delta, \mathtt{X} : A; \Gamma \vdash M : C}{\Delta; \Gamma \mid \Box\, A \vdash \mathtt{X}.\, M : C}$$

$$\textsc{FocusM}\ \frac{\mathtt{X} : A \in \Delta \qquad \Delta; \Gamma \mid A \vdash S : C}{\Delta; \Gamma \vdash \mathtt{X}\,(S) : C}$$

Seeing the stoup as a non-necessary hypothesis, and erasing all term information, this system is the sequent calculus proposed by the authors [15].

3.3 Full Sequent Calculus

As we noted previously, LJT is a focused system: it is equivalent to LJ in terms of provability but not all LJ proofs are represented. Espírito Santo [7] proposes two

term assignments λ^{Gtz} and λ_{Nat} for *resp.* full LJ (without the focusing restriction) and its corresponding natural deduction. This pair constitutes an interesting test bed for the transformation. Let us start from λ_{Nat} (restricted to the $\supset$ fragment, but easily extensible):

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

It generalizes the previous bidirectional calculus by replacing the coercion from M to R by a substitution $M[x/R]$.[6] This construct corresponds to the *cut* rule:

$$\textsc{Cut}\ \frac{\Gamma \vdash R \Rightarrow A \qquad \Gamma, x : A \vdash M \Leftarrow B}{\Gamma \vdash M[x/R] \Leftarrow B} \qquad\qquad \textsc{Annot}\ \frac{\Gamma \vdash M \Leftarrow A}{\Gamma \vdash (M : A) \Rightarrow A}$$

Transforming the corresponding type checker amounts to turn eliminations R upside down, putting annotation nodes at the top level and substitution nodes at the bottom:

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

Like in $\bar{\lambda}$, the annotation $(M : A)$ becomes a "focusing *cut*" $(V : A)\,(S)$; its "nil" construct $\cdot$ however is replaced by a new binder $x.\, M$ that allows losing the focus on the stoup:

$$\textsc{HAnnot}\ \frac{\Gamma \vdash V \Leftarrow A \qquad \Gamma \mid A \vdash S \Leftarrow B}{\Gamma \vdash (V : A)\,(S) \Leftarrow B} \qquad\qquad \textsc{Unfocus}\ \frac{\Gamma, x : A \vdash M \Leftarrow B}{\Gamma \mid A \vdash x.\, M \Leftarrow B}$$

It is precisely the calculus λ^{Gtz} of Espírito Santo.

4 Conclusion

We presented a modular, semantics-preserving program transformation turning a logical system presented in natural-deduction style into one in sequent-calculus style. It achieves the systematic and simultaneous derivation of the "reversed" term structure, the type checker (and thus the sequent rules) and the translation function from one to the other. In particular, starting from a *bidirectional* presentation of the λ-calculus, we ended up with the composition of a reversal function, taking λ-terms to $\bar{\lambda}$-terms, and a type checker for $\bar{\lambda}$-terms, in *accumulator-passing style*. The accumulator corresponds to the *stoup*, and is used to check

[6] Also, a variable is a general, checked term, and not an atom as before; this shallow difference only forces to put more type annotations to make it a bidirectional checking algorithm.

spines. Spines are *contexts* of atomic terms, which are *checked* contrarily to previous presentations, and were evidenced by CPS and defunctionalization. These two steps can be seen as a form of (partial) *compilation*, since the computation on spines is more direct than on atomic terms. Reforestation showed how spines "plug into" reversed terms, and evidenced the final structure of $\bar{\lambda}$-terms.

Composing CPS and defunctionalization has many well-documented applications: it turns evaluation functions into abstract machines [1], and exhibits the *zipper* [12], or *one-hole context* of a traversal [5]. Combined with deforestation, it turns small-step into big-step semantics [4]. This scheme was recently used to check types "by reduction" [18], but without the purpose of proof-theoretic interpretation. To the best of our knowledge, the present work is the first application of these techniques to proof theory.

One could rightfully argue that our starting point, the bidirectional λ-calculus, is only a notation for an *extension* of NJ, and is already an important step toward LJ. Indeed, it would be desirable to explain this extension similarly in terms of a systematic program transformation. Besides, we showcased the behavior of our transformation on a few known pairs of calculi; a natural continuation of this work will be to apply it to other logics, in particular to get a better understanding of focusing [13]. For instance, LJQ [11] is dual to LJT in that its focus is biased toward the conclusion, and not the hypotheses, and features a call-by-value semantics where LJT reduces in call-by-name. Still, we do not know what natural-deduction style calculus corresponds to LJQ [7]; applying our transformation backwards could help finding out. Finally, another interesting application concerns classical logic. In natural deduction it usually takes the form of a control operators (**call/cc**), whereas it appears as a facility to switch between multiple conclusions in sequent calculus. Will our transformation turn one presentation into the other? Answering these questions will require an analysis of bidirectional and canonical forms in these logics, that we leave for further investigation.

Acknowledgements We are grateful to Pierre-Louis Curien, Olivier Danvy, Hugo Herbelin, Yann Régis-Gianas, and the anonymous reviewers for their valuable comments on various drafts of this document.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *PPDP*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.
2. Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Design and correctness of program transformations based on control-flow analysis. In Naoki Kobayashi and Benjamin C. Pierce, editors, *TACS*, volume 2215 of *Lecture Notes in Computer Science*, pages 420–447. Springer, 2001.

[7] Espírito Santo [8] recently raised the question and conjectured a correspondence with his $\underline{\lambda}\mu\mathrm{let}_Q$-calculus.

3. Iliano Cervesato and Frank Pfenning. A linear spine calculus. *J. Log. Comput.*, 13(5):639–688, 2003.
4. Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion. *Inf. Process. Lett.*, 106(3):100–109, 2008.
5. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *PPDP*, pages 162–174. ACM, 2001.
6. José Espírito Santo. An isomorphism between a fragment of sequent calculus and an extension of natural deduction. In Matthias Baaz and Andrei Voronkov, editors, *LPAR*, volume 2514 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2002.
7. 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.
8. José Espírito Santo. Towards a canonical classical natural deduction system. In Anuj Dawar and Helmut Veith, editors, *CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 290–304. Springer, 2010.
9. Gerhard Gentzen. Untersuchungen über das logische Schließen. I. *Math. Z.*, 39(1):176–210, 1935.
10. 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.
11. 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.
12. Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.
13. Chuck Liang and Dale Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009.
14. Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2):177–207, 2010.
15. Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.
16. Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.
17. Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
18. Ilya Sergey and Dave Clarke. A correspondence between type checking via reduction and type checking via evaluation. *Inf. Process. Lett.*, 112(1-2):13–20, January 2012.
19. Wilfried Sieg and Saverio Cittadini. Normal natural deduction proofs (in non-classical logics). In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 169–191. Springer, 2005.
20. Philip Wadler. Deforestation: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *ESOP*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer, 1988.