A sequent calculus presentation of the Calculus of Inductive Constructions

(work in progress)

Hugo Herbelin

(jointly with Jeffrey Sarnat and Vincent Siles)

10 July 2010

Dependently Typed Programming Workshop

Edinburgh
Motivation

- sequent calculus can be seen as a \( \lambda \)-calculus
  \( \mapsto \) two main variants: \( LJ_T/LK_T \) for call-by-name, \( LJ_Q/LK_Q \) for call-by-value

- sequent calculus is a typing system for abstract machine, hence a priori for efficient reduction
  \( \mapsto \) left introduction rules build “stack”, right introduction rules build code, cut rule builds states and closures

- sequent calculus is the natural framework for proof search
  \( \mapsto \) see e.g. Lengrand’s presentation of Pure Type Systems in sequent calculus form

- sequent calculus is good at making some symmetries explicit
  \( \mapsto \) a symmetry syntactic presentation of fixpoints and cofixpoints and of the respective guard conditions
**$LJ_T$ aka Spine Calculus**

$LK_T$ and $LK_Q$ (Danos, Joinet and Schellinx, 1995) are two dual complete restrictions of $LK$ respectively connected to call-by-name and call-by-value $\lambda$-calculus with control.

$LJ_T$ is the intuitionistic restriction of $LK_T$.

$LJ_T$ normal proofs (unless $LJ$ proofs) are in bijective correspondence with call-by-name normal $\lambda$-terms.

$LJ_T$ has been independently designed by Cervesato and Pfenning under the name of Spine Calculus.
$LJ_T$ aka Spine Calculus (the propositional case)

Two kinds of sequents: $\Gamma \vdash A$ and $\Gamma; B \vdash A$ (the place for $B$ is called “stoup”).

\[
A ::= X \mid A \to A
\]

\[
\begin{array}{c}
\Gamma; A \vdash A \\
\hline
\text{Ax}
\end{array} \quad
\begin{array}{c}
\Gamma, A; A \vdash B \\
\hline
\text{CONT}
\end{array}
\]

\[
\begin{array}{c}
\Gamma \vdash A \\
\Gamma; B \vdash C
\end{array} \quad
\Gamma; A \to B \vdash C \\
\hline
\to_L
\]

\[
\begin{array}{c}
\Gamma, A \vdash B \\
\Gamma \vdash A \to B
\end{array} \quad
\Gamma \vdash B \\
\hline
\to_R
\]

\[
\begin{array}{c}
\Gamma \vdash A \\
\Gamma; A \vdash B
\end{array} \quad
\Gamma \vdash B \\
\hline
\text{CUT}
\]
$LJ_T$ aka Spine Calculus (the propositional case, annotated)

$$M, N ::= xK \mid \lambda x : A.M \mid MK \quad \text{(terms)}$$

$$K, L ::= \epsilon \mid M :: K \quad \text{(spines, or stacks)}$$

Two kinds of sequents:
- $\Gamma \vdash M : A$ for terms
- $\Gamma ; A \vdash K : B$ for spines (expecting a term of type $A$ for building a term of type $B$)

\[\frac{(x : A) \in \Gamma}{\Gamma ; A \vdash \epsilon : A} \quad \text{AX} \quad \frac{\Gamma ; A \vdash K : B}{\Gamma \vdash xK : B} \quad \text{CONT} \]

\[\frac{\Gamma \vdash M : A \quad \Gamma ; B \vdash K : C}{\Gamma ; A \to B \vdash M :: K : C} \quad \to_L \quad \frac{\Gamma \vdash \lambda x^A.M : A \to B}{\Gamma \vdash \lambda x^A.M : A \to B} \quad \to_R \]

\[\frac{\Gamma \vdash M : A \quad \Gamma ; A \vdash K : B}{\Gamma \vdash MK : B} \quad \text{CUT} \]
$LJ_T$ aka Spine Calculus (the propositional case, annotated)

In $LJ_T$, the reduction rules are cut-elimination rules for an abstract machine.

<table>
<thead>
<tr>
<th>code</th>
<th>stack</th>
<th>next state or result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\lambda x^A.M)$ $N :: K$</td>
<td>$\rightarrow$</td>
<td>$M{N/x} \ K$</td>
</tr>
<tr>
<td>$(MK)$ $L$</td>
<td>$\rightarrow$</td>
<td>$M \ (K@L)$</td>
</tr>
<tr>
<td>$(\lambda x^A.M) \ \epsilon$</td>
<td>$\rightarrow$</td>
<td>$\lambda x^A.M$</td>
</tr>
<tr>
<td>$(xK)$ $L$</td>
<td>$\rightarrow$</td>
<td>$x \ (K@L)$</td>
</tr>
</tbody>
</table>

(we use here effective substitutions but it could be done with explicit ones)
\(LJ_T\) (the Pure Type Systems case, Lengrand, 2006)

\[M, N, T, U ::= xK \mid \lambda x^T.M \mid MK \mid s \mid \Pi x^T.U\]

\[K ::= \epsilon \mid M :: K\]

\[
\frac{\Gamma \vdash T : s}{\Gamma; T \vdash \epsilon : T} \quad \text{Ax} \quad \frac{(x : T) \in \Gamma}{\Gamma; T \vdash K : U} \quad \text{CONT} \quad \frac{\Gamma \vdash M : T}{\Gamma; T \vdash K : U} \quad \text{CUT}
\]

\[
\frac{\Gamma \vdash M : T \quad \Gamma; U\{M/x\} \vdash K : C \quad \Gamma \vdash \Pi x^T.U : s}{\Gamma; \Pi x^T.U \vdash M :: K : C} \quad \rightarrow_L \quad \frac{\Gamma, x : T \vdash M : U \quad \Gamma \vdash \Pi x^T.U : s}{\Gamma \vdash \lambda x^T.M : \Pi x^T.U} \quad \rightarrow_R
\]

\[
\frac{\Gamma \text{ wf} \quad (s, s') \in \text{Ax}}{\Gamma \vdash s : s'} \quad \text{SORT} \quad \frac{\Gamma \vdash T : s_1 \quad \Gamma, x : T \vdash U : s_2 \quad (s_1, s_2, s_3) \in \text{Rel}}{\Gamma \vdash \Pi x^T.U : s} \quad \text{PI}
\]

\[
\frac{\Gamma\{;C\} \vdash M : T \quad \Gamma \vdash U : s}{\Gamma\{;C\} \vdash M : U} \quad \text{CONV}_1 \quad \frac{\Gamma; T \vdash K : C \quad \Gamma \vdash U : s}{\Gamma; U \vdash K : C} \quad \text{CONV}_R
\]

and same reduction rules
Adding inductive types
First introducing contexts and substitutions

To deal with the arity of inductive types and constructors, it is convenient to consider a “calculus of context” (see Pientka et al) with declarations asserting judgements:

\[ \Gamma \quad ++ = \quad \Gamma, x : [\Gamma \vdash T] \]

together with rules for defining substitutions:

\[ \Gamma \vdash M_0 : U_0 \quad \Gamma \vdash \overrightarrow{M} : [\Gamma' \vdash T]\{M_0/x_0\} \rightarrow [\vdash T'] \]

\[ \Gamma \vdash \epsilon : [\vdash T] \mapsto [\vdash T] \quad \Gamma \vdash M_0\overrightarrow{M} : [x_0 : U_0, \Gamma' \vdash T] \mapsto [\vdash T'] \]

and rules for applying these substitutions:

\[ \Gamma \vdash N : [\Gamma' \vdash T] \quad \Gamma \vdash \overrightarrow{M} : [\Gamma' \vdash T] \mapsto [\vdash T'] \]

\[ \Gamma \vdash N\overrightarrow{M} : T' \]
Adding dependent pattern-matching

We can then consider inductive types as declarations of the following form:

\[
I : [z : V \vdash s_I], C_i : [x_i : U_i \vdash I N_i]
\]

and we interpret a case-analysis match \( N \) with \( \ldots \mid C_i x_i \to M_i \mid \ldots \) end of natural deduction as a cut between \( N \) and a continuation \( [\ldots \mid C_i x_i \to M_i \mid \ldots] \) that matches \( N \). The extended syntax is:

\[
M, N, T, U \quad \text{++=} \quad C M, I M
\]

\[
K \quad \text{++=} \quad [\ldots \mid C x_i \to M \mid \ldots]
\]

Regarding typing, the placeholder is now dependent in types and we need to give it a name!

\[
\Gamma, z : V, y : I z \vdash T : s \quad \ldots \Gamma, x_i : U_i \vdash M_i : T\{N_i/z\}\{C_i x_i/y\}\ldots \quad \ldots C_i : [x_i : U_i \vdash I N_i].
\]

\[
\Gamma ; y : I P \vdash [\ldots \mid C_i x_i \to M_i \mid \ldots] : T\{P/x\}\{y/y\}
\]

The reduction rule is

\[
C_{i_0}(\overrightarrow{P}) \quad [\ldots \mid C_i x_i \to M_i \mid \ldots] \quad \to \quad M_{i_0}\{\overrightarrow{P}/x_{i_0}\}
\]
Because the typing rule for \([\ldots | C_i \overrightarrow{x_i} \rightarrow M_i | \ldots]\) is dependent in the type, the cut rule now needs to be dependent too:

\[
\frac{\Gamma \vdash M : T \quad \Gamma ; y : T \vdash K : U}{\Gamma \vdash MK : U\{M/y\}} \quad \text{CUT}
\]
Adding fixpoints and cofixpoints
Adding fixpoints and cofixpoints

We want to exhibit a duality between fixpoints and cofixpoints. Let us first consider a tail-recursive fixpoint without dependencies at all:

\[ f := \text{fix}_f \lambda n. \text{match } n \text{ with } 0 \rightarrow 0 \mid S \ n \rightarrow f(S(Sn)) \ \text{end} \]

Obviously, this function is cutting \( n \) with a continuation that does a case analysis on it, then depending on the result, recursively does the same case analysis. We want to interpret this recursive part as a fixpoint definition over evaluation contexts.

This suggests to consider variables \( \alpha, \beta, \ldots \) for evaluation contexts as in

\[ M, N, T, U \quad \text{++} \quad \text{cofix}_x.M \]
\[ K \quad \text{++} \quad \alpha \mid \text{fix}_\alpha.K \]

and to represent \( f \) above as the expression

\[ \lambda n. n \text{fix}_\alpha.[0 \rightarrow 0 \mid S \ n \rightarrow (S(Sn))\alpha] \]

The reduction rules come naturally:

\[ M \quad \text{fix}_\alpha.K \quad \rightarrow \quad M \quad K\{\text{fix}_\alpha.K/\alpha\} \]
\[ \text{cofix}_x.M \quad K \quad \rightarrow \quad M\{\text{cofix}_x.M/x\} \quad K \]
Adding fixpoints and cofixpoints:typing

To type evaluation context variables, we need to consider sequents with several (non-dependent) conclusions, i.e. either of the form \( \Gamma \vdash \Delta; M : T \) or \( \Gamma; x : U \vdash \Delta; K : T \) and since evaluation context variables denote terms with a hole, this suggests to have:

\[
\Delta ::= \epsilon | \Delta, \alpha : [U \vdash T]
\]

Then, we need an axiom rule for conclusions:

\[
\frac{(\alpha : [U \vdash T]) \in \Delta}{\Gamma; U \vdash \Delta; \alpha : T}
\]

We are then ready for giving the following dual rules:

\[
\begin{align*}
\Gamma, x : I \vdash M : I & \quad \Gamma; I \vdash \alpha : [I \vdash U]; K : U \\
\Gamma \vdash \text{cofix}_x.M : I & \quad \Gamma; I \vdash \text{fix}_\alpha.K : U
\end{align*}
\]

(note that the symmetry would be perfect if in \( L J^T_{\mu \mu} \) instead of \( LJ_T \))
Adding fixpoints and cofixpoints with parameters

Dependencies introduce a reading of the sequent from left to right. Let us consider the extended syntax:

\[
\begin{align*}
M, N, T, U & \quad ++ = \text{cofix}_x(\vec{y}).M \\
K & \quad ++ = \alpha | \text{fix}_\alpha(\vec{y}).K
\end{align*}
\]

For cofixpoints, the rule scales easily using declarations of contexts:

\[
\Gamma, x : [\vec{y} : \overrightarrow{T} \vdash I \overrightarrow{N}], y : \overrightarrow{T} \vdash M : I \overrightarrow{N} \\
\Gamma \vdash \text{cofix}_x(\vec{y}).M : [\vec{y} : \overrightarrow{T} \vdash I \overrightarrow{N}]
\]

For fixpoints (and we are still restricting ourselves to the tail-recursive case and no dependency in the conclusion), we need to type evaluation context variables with contexts too:

\[
\Delta ::= \epsilon \mid \Delta, \alpha : [\Gamma; U \vdash T]
\]

Then, the new rule is:

\[
\Gamma, y : \overrightarrow{T}; I \overrightarrow{N} \vdash \alpha : [y : \overrightarrow{T}; I \overrightarrow{N} \vdash U]; K : U \\
\Gamma; [y : \overrightarrow{T}; I \overrightarrow{N} \vdash U] \vdash \text{fix}_\alpha(\vec{y}).K : U
\]
Adding fixpoints and cofixpoints with parameters:
reduction rules

The reduction rules extend easily:

\[
\begin{align*}
M \rightarrow (\text{fix}_\alpha(\vec{y}).K) & \rightarrow M \\
(cofix_x(\vec{y}).M) & \rightarrow M \\
\end{align*}
\]
Adding fixpoints and cofixpoints: the general case

In the non-tail recursive case, as e.g. in \( f := \text{fix}_f \lambda n.\text{match } n \text{ with } 0 \rightarrow 0 | \ S n \rightarrow S(fn) \text{ end} \), we need to pass a continuation to the recursive evaluation-context variable. But in \( LJ_T \) a continuation is itself represented by an evaluation-context variable. Hence, we have a dependency of the recursive evaluation-context variable into another evaluation-context variable. This leads to the following generalised syntax:

\[
M, N, T, U \quad \text{++} = \quad \text{cofix}_x(\vec{y})M \\
K \quad \text{++} = \quad \alpha \mid \text{fix}_\alpha(\vec{y}\alpha)K \\
\Delta \quad ::= \quad \epsilon \mid \Delta, \alpha : [\Gamma; U \vdash \Delta; T]
\]

The axiom rule for conclusions does not change much:

\[
(\alpha : [\Gamma'; U \vdash \Delta'; T]) \in \Delta \\
\frac{}{\Gamma; [\Gamma'; U \vdash \Delta'] \vdash \Delta; \alpha : T}
\]
Adding fixpoints and cofixpoints: the general case

Now, we need to build substitutions referring to evaluation contexts:

\[ \Gamma \vdash \epsilon : [; V \vdash] \quad \Gamma \vdash M_0 : U_0 \quad \Gamma \vdash \overrightarrow{MK} : [\Gamma'; V \vdash] \{M_0/x_0\} \mapsto [; V' \vdash] \]

\[ \Gamma \vdash \overrightarrow{MK} : [x_0 : U_0, \Gamma'; V \vdash] \mapsto [; V' \vdash] \]

\[ \Gamma; V_0 \vdash K : T_0 \quad \Gamma \vdash \overrightarrow{K} : [\Gamma'; V \vdash \Delta;] \mapsto [; V' \vdash] \]

\[ \Gamma \vdash K_0 \overrightarrow{K} : [\Gamma'; V \vdash \alpha : [V_0 \vdash T_0], \Delta;] \mapsto [; V' \vdash] \]

And we need to apply these substitutions:

\[ \Gamma; [\Gamma'; V \vdash \Delta; T] \vdash K' : T' \quad \Gamma \vdash \overrightarrow{MK} : [\Gamma'; V \vdash \Delta] \mapsto [; V' \vdash] \]

\[ \Gamma; V' \vdash K' \overrightarrow{MK} : T' \]

We are now ready to give the general rule for fixpoints:

\[ \Gamma, y : \overrightarrow{T}; x : I \overrightarrow{N} \vdash \beta : [; P(y, x) \vdash U], \alpha : [y : \overrightarrow{T}; x : I \overrightarrow{N} \vdash \beta : [; P(y, x) \vdash U]; U]; K : U \]

\[ \Gamma; [y : \overrightarrow{T}; x : I \overrightarrow{N} \vdash \beta : [; P(y, x) \vdash U]] \vdash \text{fix}_\alpha(y\beta).K : U \]

(this complexity is the price to pay for tail-recursive simulation of non tail-recursive fixpoints)
Adding fixpoints and cofixpoints: the general case

The reduction rules does not change much:

\[ M \ (\text{fix}_\alpha(\vec{y}\beta).K) \overset{N}{\to} K' \rightarrow M \ K\{\overset{N}{\vec{y}/y}\}\{K'/\beta\}\{\text{fix}_\alpha(\vec{y}).K/\alpha\} \]

The non tail-recursive example is expressed like this:

\[ \lambda n. \ n \text{fix}_\alpha(\beta).[0 \rightarrow 0 \mid S \ n \rightarrow n(\alpha(\tilde{\mu}x.\ (S \ x))) ] \]

Note that for building non linear evaluation contexts, we a priori need the following extra rule adapted from \( LJ^{T}_{\mu\tilde{\mu}} \) to \( LJ_{T} \):

\[
\frac{\Gamma, x : A \vdash M : B}{\Gamma; A \vdash \tilde{\mu}x. M : B}
\]
Symmetry of the guard conditions

We have the following symmetry:

Guard condition for fixpoint = recursion traverses at least one left introduction rule

Guard condition for cofixpoint = recursion traverses at least one right introduction rule

For inductive types and fixpoints, termination comes from the interaction between a finite term and an infinite guarded evaluation context.

For coinductive types and cofixpoints, termination comes from the interaction between a guarded infinite term and a finite evaluation context.

Note that in this duality, the difference between inductive and coinductive types is not a built-from-constructor vs built-from-destructors duality but a finite-infinite vs infinite-finite duality.