The computational content of classical logic

Hugo Herbelin

INRIA

Proofs-as-programs summer school - Eugene, Oregon, USA

July 2002

Plan

- The computational content of classical logic

- The computational content of (classical) sequent calculus

Part I

The computational content of classical logic

$$
\begin{aligned}
\text{Classical logic} &= \text{Intuitionistic logic} + A \vee \neg A \text{ (excluded-middle)} \\
&= \text{Intuitionistic logic} + ((A \to B) \to A) \to A \text{ (Peirce's Law)} \\
&= \text{Intuitionistic logic} + \neg\neg A \to A
\end{aligned}
$$

Has classical logic a computational content?

Intuitionistic logic has a computational content: λ-calculus. The λ-terms can be evaluated (by hand or on a computer) and reduced to normal forms. Especially, evaluating a proof of $\exists x : int.P(x)$ produces an integer t satisfying $P(t)$.

In natural deduction, evaluation is normalisation of the proofs. But classical logic also enjoys normalisation properties.

The first proof of normalisation for classical logic is Gentzen cut-elimination proof of (classical) sequent calculus (1935). What is the "meaning" of this result. Why not to implement sequent calculus and cut-elimination and then compute in classical logic?

A first reason is that sequent calculus is missing the intuitiveness of natural deduction (how to understand the left introduction rules?).

A second reason is that sequent calculus is a symmetric calculus and that cut-elimination forces to break the symmetry. There are several ways to break the symmetry and no clear argument why to make a choice rather than the other. Moreover, is there really no hope to preserve the symmetry?

In Prawitz, a proof of normalisation of classical proofs in natural deduction + $\neg\neg A \rightarrow A$ is presented (1965). The Curry-Howard correspondance becoming manifest in the late 70's. Then it would have been "natural" to extend λ-calculus with an operator of type $\neg\neg A \rightarrow A$ and to compute with, following the rules of Prawitz.

It does not happen this way. Instead, it is from the study of compilers that the execution of classical logic arrived. Scheme people introduced an operator named *call-with-current-continuation* (or *call-cc*) to let the user access the evaluation context of an ongoing computation. In 1990, Griffin discovered that the $\mathcal{C}$ operator (a control operator close to *call-cc* was typable of type $\neg\neg A \rightarrow A$. That an axiom of type $\neg\neg A \rightarrow A$ was *really* implementable has been the starting point of the study of the computational content of classical logic.

Plan of part I

- From classical logic to intuitionistic logic: $\neg\neg$-translation

- Especially, classical logic is conservative over intuitionistic logic for $\forall x \exists y.P(x, y)$ statements ($P(x, y)$ decidable)

- The main discovery: `call-cc` has type $((\neg A) \to A) \to A$

- $\neg\neg$-translation and continuation-passing-style translation

The non-determinism of classical logic: call-by-value and call-by-name classical logic

A step to symmetry:

- classical logic from multiple conclusions: classical natural deduction and the $\lambda\mu$-calculus

Preliminary remarks

Minimal logic: logic based on $\rightarrow$ (no rule about $\perp$ nor $\neg$)

The intuitionistic axiom:

- $\perp \rightarrow A$ ("ex falso quodlibet")

Intuitionistic logic: minimal logic + the axiom above

Remark second-order minimal logic is intuitionistic since $\perp$ is definable

Preliminary remarks (continued)

The classical axioms

- Peirce's Law $((A \to B) \to A) \to A$
- Excluded-middle $A \vee \neg A$ ("tertio non datur")
- Double negation elimination: $\neg\neg A \to A$

Classical logic: intuitionistic logic + one of these axioms

Lemma

- $A \vee \neg A$ is equivalent to $((A \to \bot) \to A) \to A$
- if $\bot \to A$ then EM is equivalent to Peirce's Law
- $\neg\neg A \to A$ is equivalent to $(((A \to \bot) \to A) \to A) \wedge (\bot \to A)$

Then, it's equivalent to add $\neg\neg A \to A$ to minimal logic (it implies $\bot \to A$) or to add Peirce's Law (or excluded-middle, or $\neg\neg A \to A$) to intuitionistic logic.

Remark there is a minimal classical logic: no negation, no $\bot$, but Peirce's Law.

From classical to intuitionistic logic: ¬¬-translation

(Kolmogorov 1925, Glivenko, Gödel, Gentzen)

$$
\begin{array}{lcl}
X^* & = & \neg\neg X \\
\perp^* & = & \perp \\
(A \wedge B)^* & = & A^* \wedge B^* \\
(A \vee B)^* & = & \neg\neg(A^* \vee B^*) \\
(A \to B)^* & = & A^* \to B^* \\
(\exists x.P)^* & = & \neg\neg\exists x.(P^*) \\
(\forall x.P)^* & = & \forall x.P^*
\end{array}
$$

Lemma $\vdash_I \neg\neg(A \vee \neg A)$

Lemma $\Gamma \vdash_C A$ implies $\Gamma^* \vdash_I A^*$.

Corollary if $\Gamma \vdash_C A$ then there is A' such that $\Gamma \vdash_C A \leftrightarrow A'$ and $\Gamma \vdash_I A'$.

Lemma if A has no $\vee$, $\exists$, then $\vdash_{PA} A$ implies $\vdash_{HA} A$.

Remark $\vdash_I \perp \to A^*$

Uninteresting for extraction... A^* is a Harrop formula.

From classical to minimal logic: $\neg_A\neg_A$-translation

Actually, the $\neg\neg$-translation is a translation to minimal logic, hence $\perp$ is neutral and can be replaced by any arbitrary formula.

Let A be a formula and define $\neg_A = B \to A$.

Define $\neg_A\neg_A$-translation ^{*A} from $\neg\neg$-translation by replacing $\perp$ by A (hence $\neg$ by $\neg_A$).

Lemma $\vdash_M \neg_A\neg_A(B \vee \neg_A B)$

Lemma $\vdash_M (\perp \to B)^{*A}$

Lemma $\Gamma \vdash_C B$ implies $\Gamma^{*A} \vdash_M B^{*A}$

From intuitionistic to minimal logic: A-translation

(Friedman 1978)

Let A a given formula. The A-translation of B, written B_A is obtained by replacing $\perp$ by A and atomic formulas X by $X \vee A$.

Lemma $\Gamma \vdash_M A \to B_A$

Lemma $\Gamma \vdash_I B$ implies $\Gamma_A \vdash_M B_A$

Lemma $\Gamma \vdash_{HA} B$ implies $\Gamma_A \vdash_{HA-\perp} B_A$

Digression: Markov's rule

Consider a Σ^0_1-formula, i.e. an arithmetical formula of the form $\exists x F(x,y) = 0$ with $F(x,y)$ (provably) recursive.

Markov's rule states that $\vdash \neg\neg\exists x F(x,y) = 0$ implies $\vdash \exists x F(x,y) = 0$.

Theorem (Kreisel) Markov's rule is not derivable in Heyting Arithmetic.

Theorem Markov's rule is admissible in Heyting Arithmetic.

Proof Just set $A \equiv \exists x F(x,y) = 0$ then $\vdash_{HA} \neg\neg\exists x F(x,y) = 0$ implies $\vdash_{HA} ((\exists x(F(x,y) = 0 \vee A)) \to A) \to A$, which implies $\vdash_{HA} ((\exists x F(x,y) = 0 \to A) \to A$, which implies $\vdash_{HA} \exists x F(x,y) = 0$

Conservativity of classical logic over intuitionistic logic for Π^0_2-formulas

Lemma $\vdash_{PA} \forall x \exists y P(x,y)$ implies $\vdash_{HA} \forall x \exists y P(x,y)$

There is no specification $\forall x \exists y P(x,y)$ *not provable in intuitionistic logic.*

Especially, `call-cc` which "implements" classical logic, does not provide more (extensional) functions from `integer` to `integer`.

`call-with-current-continuation`

`call-with-current-continuation` is a control operator introduced in Scheme (available in SML too with name `call-cc`, and written $\mathcal{K}$ in literature). Its operational semantics requires an abort operator (not available to the user in Scheme, called $\mathcal{A}$ in literature and close the `throw` operator of SML).

The reduction rules of `call-cc` (call-by-name) are

$$\begin{array}{lcl} E(\mathcal{K}M) & \longrightarrow & E(M\lambda x.\mathcal{A}(E(x))) \\ E(\mathcal{A}M) & \longrightarrow & M \end{array}$$

where the evaluation contexts are defined by

$$E \;::=\; (\;) \mid EM$$

Example of use of $\mathcal{K}$

$$
\begin{array}{l}
\text{let } G(l) = \mathcal{K}(\lambda k. \\
\qquad \text{let rec } F(l) = \ \text{case } l \text{ of} \\
\qquad \begin{array}{lcl}
|[\,] & \rightarrow & 1 \\
|0 :: \_ & \rightarrow & k(0) \\
|a :: l & \rightarrow & a * F(l)
\end{array} \\
\qquad \text{in } F(l)) \\
\text{in } G([3;0;6;9]) + 4
\end{array}
$$

computes to 4 shortcutting the multiplication by 6 and 9.

How it computes?

Since we have a let operator, let's first generalize the language of contexts (in call by name evaluation)

$$E \ ::= \ [] \mid EM \mid \text{let } x = E \text{ in } M \mid \text{let } x = V \text{ in } E \mid E + M \mid n + E \mid \ \ldots$$

Then we can proceed

$$\begin{array}{l}\lambda l.\mathcal{K}(\lambda k. \\ \qquad \begin{array}{l}\text{let rec } F(l) = \ \text{case } l \text{ of} \\ \quad \begin{array}{lll} |[\,] & \rightarrow & 1 \\ |0 :: \_ & \rightarrow & k(0) \\ |a :: l & \rightarrow & a * F(l) \end{array} \\ \text{in } F(l)) \\ ([3;0;6;9]) + 4 \end{array}\end{array} \longrightarrow \begin{array}{l}\mathcal{K}(\lambda k. \\ \qquad \begin{array}{l}\text{let rec } F(l) = \ \text{case } l \text{ of} \\ \quad \begin{array}{lll} |[\,] & \rightarrow & 1 \\ |0 :: \_ & \rightarrow & k(0) \\ |a :: l & \rightarrow & a * F(l) \end{array} \\ \text{in } F([3;0;6;9])) + 4 \end{array}\end{array} \longrightarrow$$

$$\begin{array}{l}\text{let rec } F(l) = \ \text{case } l \text{ of} \\ \quad \begin{array}{lll} |[\,] & \rightarrow & 1 \\ |0 :: \_ & \rightarrow & \mathcal{A}(0+4) \\ |a :: l & \rightarrow & a * F(l) \end{array} \\ \text{in } F([3;0;6;9]) + 4 \end{array} \xrightarrow{*} \begin{array}{l}\text{let rec } F(l) = \ \text{case } l \text{ of} \\ \quad \begin{array}{lll} |[\,] & \rightarrow & 1 \\ |0 :: \_ & \rightarrow & \mathcal{A}(0+4) \\ |a :: l & \rightarrow & a * F(l) \end{array} \\ \text{in } F([0;6;9]) + 4 \end{array} \xrightarrow{*} \mathcal{A}(0+4) + 4 \longrightarrow 0 + 4$$

`call-with-current-continuation` has type Peirce's Law

SML introduces the type $A \to$ `cont` to type `call-cc`.

$$\begin{aligned} \texttt{call-cc} &: (A \text{ cont} \to A) \to A \\ \texttt{throw} &: A \text{ cont} \to A \to B \end{aligned}$$

Of course, $A \to$ `cont` can be interpreted by $\neg A$

Where SML writes

```
call-cc (fun k -> .... throw k a .... throw k b ...),
```

Scheme writes

```
call-with-current-continuation (fun k -> .... k a .... k b ...).
```

From which it is clear that `call-with-current-continuation` is typable by $(\neg A \to A) \to A$ and $k : \neg A$ implicitly contains a call to the abort operator of type $\perp \to B$.

Using `call-cc` to encode forward `goto`

	instr_1	/* modifying x, y */
	instr_2 ... goto L ...	/* modifying x */
	instr_3	/* modifying y, z */
	instr_4 ... goto L...	/* modifying x, z */
	instr_5	/* modifying x */
L:	instr_6	

can be rewritten into functional code:

```
let (x,y,z) = call-cc (fun k ->
  let (x,y) = ... in
  let x = ... k (x,y,z) ... in
  let (y,z) = ... in
  let (x,z) = ... k (x,y,z) ... in
  let x = ... in
  (x,y,z)
in
  (* L *)
  ...
```

The "binary example"

Let f be a sequence from N to {0,1}. Consider a (classical) proof of the fact there is an infinity of 0 or an infinity of 1. Infer that f has the same value on at least two points!

`infinity` realizes $\forall f \exists b \forall x \exists y . y \geq x \wedge f(y) = b$ (source code is in Scheme)

```
;;; infinity : (nat -> bool) -> bool * (nat -> nat)
(define (infinity f)
  (call-with-current-continuation
   (lambda (k)
     (k (cons #t (lambda (x0)
                  (call-with-current-continuation
                   (lambda (k0)
                     (k (cons #f (lambda (x1)
                                  (call-with-current-continuation
                                   (lambda (k1)
                                     (let ((m (max x0 x1)))
                                       (if (f m) (k0 m) (k1 m))))))))))))))))
```

The "binary example" (continued)

get_two realizes $\forall f \exists (b, x1, x2). x2 \neq x1 \wedge f(x1) = b \wedge f(x2) = b$

```
;;; get-two : (nat -> bool) -> bool * nat * nat
(define (get-two f)
  (let* ((r (infinity f))
         (b (car r))
         (g (cdr r))
         (x1 (g 0))
         (x2 (g (+ 1 x1))))
    (cons b (cons x1 x2))))
```

The classical proof of the binary example is asymmetric

```
;;; f = #t #t ...
(get-two (lambda (x) #t))
;;; => '(#t 0 . 1)

;;; f = #f #f ...
(get-two (lambda (x) #f))
;;; => '(#f 0 . 1)

;;; f = #f #t #t ...
(get-two (lambda (x) (if (= x 0) #f #t)))
;;; => '(#t 1 . 2)

;;; f = #t #f #f ...
(get-two (lambda (x) (if (= x 0) #t #f)))
;;; => '(#f 1 . 2)

;;; f = #t #f #t ...
(get-two (lambda (x) (even? x)))
;;; => '(#t 0 . 2)

;;; f = #f #t #f #f
(get-two (lambda (x) (if (= x 1) #t #f)))
;;; => '(#f 2 . 3)

;;; f = #f #t #f #t ...
(get-two (lambda (x) (odd? x)))
;;; => '(#t 1 . 3)
```

The $\lambda_{\mathcal{K}}$-calculus

$\lambda_{\mathcal{K}}$-calculus extends Curry-Howard correspondence to natural deduction $+ ((A \to \bot) \to A) \to A$.

$$
\begin{array}{llll}
(\lambda x.M)N & \longrightarrow & M[x := N] & (\beta) \\
E(\mathcal{K}M) & \longrightarrow & \mathcal{K}(\lambda k.E(M\lambda x.k(E(x)))) & (\mathcal{K}) \\
 & & & \\
E(\mathcal{K}M) & \longrightarrow & E(M\lambda x.\mathcal{A}(E(x))) & \text{only at toplevel} \\
E(\mathcal{A}M) & \longrightarrow & M & \text{only at toplevel}
\end{array}
$$

The *toplevel* rules are well-typed only because $\bot$ (which is formal) is identified with the type of the toplevel result.

Felleisen's $\mathcal{C}$ operator

Felleisen's $\mathcal{C}$ operator has the following reduction rule

$$E(\mathcal{C}M) \longrightarrow \mathcal{C}(\lambda k.M(\lambda x.k(E(x))))$$

and, if at toplevel,

$$E(\mathcal{C}M) \longrightarrow M(\lambda x.\mathcal{A}(E(x)))$$

$\mathcal{C}$ is typable of type $\neg\neg A \to A$

The $\lambda_{\mathcal{C}}$-calculus extend Curry-Howard correspondence to natural deduction $+ \neg\neg A \to A$.

Question: define $\mathcal{A}$ from $\mathcal{C}$.

Question: define $\mathcal{C}$ from $\mathcal{K}$ and $\mathcal{A}$.

Continuation-Passing-Style Translation

A program using `call-cc` can be turned into a program without `call-cc`.

$$
\begin{array}{lcl}
x^n & = & \lambda k.xk \\
(\lambda x.M)^n & = & \lambda k.(k\lambda x.M^n) \\
(MN)^n & = & \lambda k.M^n(\lambda m.mN^n k)) \\
\texttt{call-cc}(M)^n & = & \lambda k.M^n(\lambda m.m(\lambda z.z(\lambda d.zk))k) \\
(M,N)^n & = & \lambda k.k(M^n, N^n) \\
i_j(M)^n & = & \lambda k.M^n(\lambda m.k(i_j(m))) \\
n^n & = & \lambda k.kn \\
\mathrm{Y}(M)^n & = & \lambda k.M^n(\lambda f.Y(f)k)
\end{array}
$$

Proposition $M \longrightarrow_{CBN} n$ iff $M^n(\lambda x.x) \longrightarrow n$ (any strategy)

Proposition $M =_{\beta\mathcal{K}} N$ iff $M^n =_\beta N^n$

$\neg_A\neg_A$-translation types CPS-Translation

Proposition $\Gamma \vdash M : B$ implies $\Gamma^{*^n_A} \vdash M^n : B^{*^n_A}$

where

$$
\begin{array}{lcl}
X^{*^n_A} & = & \neg_A\neg_A X \\
(B \to C)^{*^n_A} & = & \neg_A\neg_A(B^{*^n_A} \to C^{*^n_A}) \\
(B \wedge C)^{*^n_A} & = & \neg_A\neg_A(B^{*^n_A} \wedge C^{*^n_A}) \\
(B \vee C)^{*^n_A} & = & \neg_A\neg_A(B^{*^n_A} \vee C^{*^n_A})
\end{array}
$$

Corollary $\vdash M : \mathit{integer}$ implies $\vdash M^n : (\mathit{integer} \to \mathit{integer}) \to \mathit{integer}$, hence $\vdash M(\lambda x.x) : \mathit{integer}$

Call-by-value CPS-translation

$$
\begin{array}{lcl}
x^v & = & \lambda k.kx \\
(\lambda x.M)^v & = & \lambda k.(k\lambda x.M^v) \\
(MN)^v & = & \lambda k.M^v(\lambda m.N^v(\lambda n.mnk)) \\
\texttt{call-cc}(M)^v & = & \lambda k.M^v(\lambda m.m(\lambda z.z(\lambda d.zk))k) \\
(M,N)^v & = & \lambda k.M^v(\lambda m.N^v(\lambda n.kmn)) \\
i_j(M)^v & = & \lambda k.k(i_j(M^v)) \\
n^v & = & \lambda k.kn \\
\mathrm{Y}(M)^v & = & \lambda k.M^v(\lambda f.Y(\lambda x.\lambda h.x(\lambda y.fyh)k)
\end{array}
$$

Proposition $t \longrightarrow_{CBV} n$ iff $t^v(\lambda x.x) \longrightarrow n$ (any strategy)

Proposition $M =_{\beta_V \mathcal{K}_V} N$ implies $M^n =_\beta N^n$

where the rule K_V is similar to $\mathcal{K}$ but for CBV contexts

Kuroda's $\neg_A\neg_A$-translation types CBV CPS-Translation

Proposition $\Gamma \vdash M : B$ implies $\Gamma^{*^v_A} \vdash M^v : B^{*^v_A}$

where

$$
\begin{array}{lcl}
B^{*^v_A} & = & \neg_A\neg_A B \\
\\
X^{+^v_A} & = & X \\
(B \to C)^{+^v_A} & = & B^{+^v_A} \to C^{*^v_A} \\
(B \land C)^{+^v_A} & = & B^{+^v_A} \land C^{+^v_A} \\
(B \lor C)^{+^v_A} & = & \neg_A\neg_A(B^{+^v_A} \lor C^{+^v_A})
\end{array}
$$

Plotkin's call-by-value Calculus

Values

$$V \;::=\; x \mid \lambda x.M$$

Restriction of β-reduction to values

$$\begin{array}{lcll} (\lambda x.M)V & = & M[x := V] & (\beta_V) \\ \lambda x.Vx & = & V \quad x \notin V & (\eta_V) \end{array}$$

Fails to capture all equalities true for its CPS-translation.

Proposition (Plotkin) $\vdash M =_{\beta_V \eta_V} N$ 'implies $\vdash M =_{\beta\eta} N$ but not conversely

Felleisen-Sabry call-by-value calculus

Add commutation rules

$$
\begin{array}{lcll}
E((\lambda x.N)M) & \longrightarrow & (\lambda x.E(N))M & (\beta_{lift}) \\
E(MNL) & \longrightarrow & (\lambda x.E(N))M & (\beta_{flat})
\end{array}
$$

This calculus is similar to Moggi computational calculus.

Classical natural deduction

(Parigot, 1992)

Classical natural deduction is an extension of natural deduction with multiple conclusions, as in Gentzen's sequent calculus.

$$\frac{\Gamma \vdash \Delta, A \to B \qquad \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, B} \to_L \qquad \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \to B} \to_R$$

$$\frac{}{\Gamma, A \vdash \Delta, A} \; Ax \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A} \; Cont$$

Example: derivability of Peirce's Law.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(A \to B) \to A \vdash (A \to B) \to A} \qquad \dfrac{A \vdash B, A}{\vdash A \to B, A}}{(A \to B) \to A \vdash A, A}}{(A \to B) \to A \vdash A}}{\vdash ((A \to B) \to A) \to A}$$

$\lambda\mu$-calculus

(Parigot 1992)

Syntax of untyped $\lambda\mu$-calculus

$$\begin{array}{lcll} t & ::= & x \mid tt \mid \lambda x.t \mid \mu\alpha c & \text{(terms)} \\ c & ::= & [\alpha]t & \text{(executables)} \end{array}$$

The variables α, β, ... are continuation variables.

The $\lambda\mu$-calculus extend Curry-Howard correspondence to classical natural deduction.

Typing uses two kinds of sequents: $\Gamma \vdash \Delta; A$ and $\Gamma \vdash \Delta$

$$\frac{\Gamma \vdash \Delta; t : A \to B \qquad \Gamma \vdash \Delta; u : A}{\Gamma \vdash \Delta; tu : B} \qquad \frac{\Gamma, x : A \vdash \Delta; t : B}{\Gamma \vdash \Delta; \lambda x.t : A \to B}$$

$$\frac{}{\Gamma, x : A \vdash \Delta; x : A} \qquad \frac{\Gamma \vdash \Delta, \alpha : A; t : A}{[\alpha]t : \Gamma \vdash \Delta, \alpha : A} \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \Delta; \mu\alpha c : A}$$

Some proofs

Peirce's law: $\lambda H.\mu\alpha[\alpha](H\lambda a.\mu\delta[\alpha]a) \ : \ ((A \to B) \to A) \to A$

Excluded-middle: $\mu\alpha[\alpha]i_2(\lambda a.\mu\delta[\alpha]i_1(a)) \ : \ A \vee \neg A$

Can we prove $\bot \to A$ in (minimal) simply typed $\lambda\mu$-calculus? No, except assuming a free continuation variable of type $\bot$.

Question: How many proofs of $\neg A \vee \neg\neg A$?

Question: Is $\neg\neg A \to A$ derivable in simply typed $\lambda\mu$-calculus?

Extension to $\exists$ and $\forall$

$$\frac{\Gamma \vdash \Delta; t : P(e)}{\Gamma \vdash \Delta; (e,t) : \exists a.P(a)} \qquad \frac{\Gamma \vdash \Delta; t : P(a) \qquad (a \notin \Gamma, \Delta)}{\Gamma \vdash \Delta; \lambda a.t : \forall a.P(a)}$$

$$\frac{\Gamma \vdash \Delta; t : \exists a.P(a) \qquad \Gamma, x : P(a) \vdash \Delta; u : C \qquad (a \notin \Gamma, \Delta, C)}{\Gamma \vdash \Delta; \mathrm{let}\ (a,x) = t\ \mathrm{in}\ u : C} \qquad \frac{\Gamma \vdash \Delta; t : \forall a.P(a)}{\Gamma \vdash \Delta; te : P(e)}$$

How to prove Drinker's Paradox ?

Proof of the Drinker Paradox

This is a normal proof of the drinker proof (you can find it e.g. by using excluded-middle and then normalizing)

$$\mu\alpha[\alpha](a_0, \lambda x_a.\lambda b.\mu\beta[\alpha](b, \lambda x_b.\lambda b'.\mu\delta[\beta]x_b)) : \exists a.(P(a) \to \forall b.P(b))$$

where a_0 is an arbitrary inhabitant of the domain.

Call-by-name and call-by-value $\lambda\mu$-calculus

Call-by-name

$$\begin{array}{lll} (\lambda x.t)u & \longrightarrow & t[x := u] & (\beta) \\ (\mu\alpha[\beta]t)u & \longrightarrow & \mu\alpha(([\beta]t)[[\alpha]\bullet := [\alpha] \bullet u]) & (\mu_N) \\ \mu\alpha[\alpha]t & \longrightarrow & t \qquad (\alpha \notin t) & (\mu_{simpl}) \end{array}$$

Lemma Call-by-name $\lambda\mu$-calculus is confluent

Lemma Various typed version of call-by-name $\lambda\mu$-calculus are strongly normalizing (e.g. second-order logic)

Call-by-value ($V \ ::= \ x \mid \lambda x.t$)

$$\begin{array}{llll} (\lambda x.t)V & \longrightarrow & t[x := V] & (\beta_V) \\ u(\mu\alpha[\beta]t) & \longrightarrow & \mu\alpha(([\beta]t)[[\alpha]\bullet := [\alpha]u\bullet]) & (\mu_V) \\ \mu\alpha[\alpha]t & \longrightarrow & t \qquad (\alpha \notin t) & (\mu_{simpl}) \end{array}$$

Lemma Call-by-value $\lambda\mu$-calculus is confluent

Remark μ_N introduces a critical pair with β_V and μ_V: no confluence can be expected for the whole set of rules.

Relations between $\mathcal{K}$, $\mathcal{C}$ and μ

$$\mathcal{K}(\lambda k.....ka....kb...)$$

is equivalent to

$$\mu\alpha[\alpha](....\mu\delta[\alpha]a....\mu\delta[\alpha]b...)$$

assuming that the argument of $\mathcal{K}$ is η-expanded.

$$\mathcal{C}(\lambda k.k'...\mathcal{C}(\lambda k''.ka...)...)$$

is equivalent to

$$\mu\alpha[\beta](...\mu\gamma[\alpha]a...)$$

Classical Pure Type Systems

(Barthe, Hatcliff, Sorensen)

$$
\begin{array}{lcl}
t, u, T, U & ::= & x \mid tt \mid \lambda x : t.t \mid \pi x : t.t \mid \mu\alpha : t.c \\
c & ::= & [\alpha]t \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, \alpha : T^{\perp}
\end{array}
$$

Conversion: $\beta + \mu_N$

$$
\frac{}{\emptyset \text{ wf}} \qquad \frac{\Gamma \vdash T : s}{\Gamma, x : T \text{ wf}} \qquad \frac{\Gamma \vdash T : s}{\Gamma, \alpha : T^{\perp} \text{ wf}} \qquad \frac{x : T \in \Gamma \text{ wf}}{\Gamma \vdash x : T} \qquad \frac{\Gamma \text{ wf} \qquad (s_1, s_2) \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}
$$

$$
\frac{\Gamma \vdash t : T \qquad (\alpha : T^{\perp}) \in \Gamma}{[\alpha]t : (\Gamma \vdash)} \qquad \frac{c : (\Gamma, \alpha : T^{\perp} \vdash)}{\Gamma \vdash \mu\alpha c : T)}
$$

$$
\frac{\Gamma, x : T \vdash u : U \qquad \Gamma \vdash (\Pi x : T.U) : s}{\Gamma \vdash \lambda x : T.u : (\Pi x : T.U)} \qquad \frac{\Gamma \vdash T : s_1 \qquad \Gamma, x : T \vdash U : s_2 \qquad (s_1, s_2, s_3) \in \mathcal{R}}{\Gamma \vdash (\Pi x : T.U) : s_3}
$$

$$
\frac{\Gamma \vdash u : (\Pi x : T.U) \qquad \Gamma \vdash t : T}{\Gamma \vdash ut : U[t/x]} \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash U : s \qquad \Gamma \vdash T =_{\beta\mu_N} U}{\Gamma \vdash t : U}
$$

There is a CPS-translation from classical Barendregt's cube to the intuitionistic one. The classical cube is consistent and strongly normalizing.

Part II

The computational content of (classical) sequent calculus

Gentzen's sequent calculus (Gentzen 1935)

Initially called "Logistischer Kalkül" (LJ in the intuitionistic case and LK in the classical case) is now known (since Prawitz) as sequent calculus.

- No elimination rules but left introduction rules

- No Modus Ponens, computation is condensed into one single rule: the cut rule.

$$\frac{}{\Gamma, A \vdash \Delta, A}\ Ax$$

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta}\ \to_L \qquad \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta; A \to B}\ \to_R$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge_L^1 \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\ \wedge_L^2 \qquad \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}\ \wedge_R$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\ \vee_L \qquad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B}\ \vee_R^1 \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B}\ \vee_R^2$$

$$\frac{\Gamma \vdash \Delta, A \qquad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}\ Cut$$

Interpreting right introduction rules

Right introduction rules of sequent calculus can be interpreted as in natural deduction.

As in Parigot's $\lambda\mu$-calculus, this requires however to distinguish the formula we are currently working on, which leads to two kinds of sequents.

$$\frac{\Gamma, x : A \vdash \Delta; t : B}{\Gamma \vdash \Delta; \lambda x.t : A \to B} \to_R$$

What about the axiom rule?

As in λ- or $\lambda\mu$-calculus it declares a variable.

$$\frac{}{\Gamma, x : A \vdash \Delta; x : A} Ax_R$$

Changing the conclusion we are working on

This is typical from classical logic, we may need to change the conclusion we are working on (this is typically the case in the proof of $A \vee \neg A$).

As in $\lambda\mu$-calculus, we introducing two new rules (which clearly have no counterpart in the usual presentation of sequent calculus).

$$\frac{\Gamma \vdash \Delta, \alpha : A; t : A}{t \star \alpha : (\Gamma \vdash \Delta, \alpha : A)}\ {\star_R} \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \Delta; \mu\alpha.c : A}\ \mu$$

Construction $\mu\alpha.c$ is the same as in $\lambda\mu$-calculus (which is the same $\mathcal{C}(\lambda\alpha.c)$ in $\lambda_{\mathcal{C}}$-calculus) and $t \star \alpha$ is just another way to write $[\alpha]t$ (or αc in $\lambda_{\mathcal{C}}$-calculus).

Interpreting implication left introduction rule

... as an argument list builder!

We need for this a third kind of sequents distinguishing a formula not on the right but on the left (consistently with the symmetry of sequent calculus.

$\Gamma \vdash \Delta; t : A$
$c : (\Gamma \vdash \Delta)$
$\Gamma; e : B \vdash \Delta$

$\Gamma; e : B \vdash A$ interpreted by "continuations"

Example: in $xu_1 ... u_n$, the argument $u_1 ... u_n$ can be seen as the continuation of the evaluation of x.

Left implication introduction rule in *intuitionistic* sequent calculus

Three kinds of sequents:

$\Gamma \vdash ; t : A$
$c : (\Gamma \vdash A)$
$\Gamma ; e : B \vdash A$

We can now make explicit our interpretation of the left implication introduction rule:

$$\frac{\Gamma \vdash ; t : A \qquad \Gamma ; e : B \vdash C}{\Gamma ; t \cdot e : A \to B \vdash C} \to_L$$

where $t \cdot e$ is just a notation for the list $t t_1 ... t_n$ assuming e is $t_1 ... t_n$.

We need a rule for the empty list of arguments (let's write it ϵ). Let's introduce a rule dual to the previous axiom rule:

$$\frac{}{\Gamma ; \epsilon : A \vdash \epsilon : A} Ax_L$$

The cut rule

We need to apply arguments to terms and will use the cut rule.

$$\frac{\Gamma \vdash ; t : A \qquad \Gamma ; e : A \vdash B}{t \star e : (\Gamma \vdash B)} \; Cut$$

where $t \star e$ is just another notation for te.

Some redundancies appear: rule $\star_R$ to undistinguish a formula is derivable from the cut rule :

$$\frac{\Gamma \vdash ; t : A \qquad \dfrac{}{\Gamma ; \epsilon : A \vdash \epsilon : A} \; Ax_R}{t \star \epsilon : (\Gamma \vdash \epsilon : A)} \; Cut$$

We are now faced to a choice, do we have to restrict the cut rule or do we accept to express cut-free proofs (in the usual sense) by proofs using a cut rule, even if the use of the cut rule is here degenerated in some sense since applied to an axiom rule.

Summarizing the interpretation of CBN intuitionistic sequent calculus

Until now, what we get is *call-by-name* intuitionistic sequent calculus.

$$
\begin{array}{rcl}
t & ::= & \lambda x.t \mid \mu\epsilon.c \mid x \\
e & ::= & t \cdot e \mid \epsilon \\
c & ::= & t \star e
\end{array}
$$

where the same notation $t \star e$ actually covers different rules

$$
\frac{}{\Gamma, x : A \vdash ; x : A}\; Ax_L
\qquad\qquad
\frac{}{\Gamma ; \epsilon : A \vdash \epsilon : A}\; Ax_R
$$

$$
\frac{\Gamma \vdash \Delta ; t : A \qquad \Gamma ; e : B \vdash C}{\Gamma ; t \cdot e : A \to B \vdash C}\; \to_L
\qquad\qquad
\frac{\Gamma, x : A \vdash ; t : B}{\Gamma \vdash ; \lambda x.t : A \to B}\; \to_R
$$

$$
\frac{c : (\Gamma \vdash \epsilon : A)}{\Gamma \vdash ; \mu\epsilon.c : A}\; \mu
$$

$$
\frac{\Gamma \vdash ; t : A \qquad \Gamma ; e : A \vdash B}{t \star e : (\Gamma \vdash B)}\; Cut
$$

Moving to CBN classical sequent calculus

We move to classical logic by considering that ϵ (for empty list) is more than the bottom of the list but actually the name of the continuation of the application.

The new syntax is

$$\begin{array}{rcl} t & ::= & \lambda x.t \mid \mu\alpha.c|x \\ e & ::= & t \cdot e \mid \alpha \\ c & ::= & t \star e \end{array}$$

and the rules are

$$\frac{}{\Gamma; \alpha : A \vdash \Delta, \alpha : A}\ Ax_L \qquad \frac{}{\Gamma, x : A \vdash \Delta; x : A}\ Ax_R$$

$$\frac{\Gamma \vdash \Delta; t : A \qquad \Gamma; e : B \vdash \Delta}{\Gamma; t \cdot e : A \to B \vdash \Delta}\ \to_L \qquad \frac{\Gamma, x : A \vdash \Delta; t : B}{\Gamma \vdash \Delta; \lambda x.t : A \to B}\ \to_R$$

$$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \Delta; \mu\alpha.c : A}\ \mu$$

$$\frac{\Gamma \vdash \Delta; t : A \qquad \Gamma; e : A \vdash \Delta}{t \star e : (\Gamma \vdash \Delta)}\ Cut$$

The missing operator: dual of μ is a `let-in` operaror

To get the symmetry, we must understand the rule

$$\frac{c : (\Gamma, x : B \vdash \Delta)}{\Gamma; ? : B \vdash \Delta}$$

From an executable depending on $x : B$, we should arrive to a continuation. It is the continuation which expects a term of type B, bind it to x and compute according to c. This is a let $x = [\,]$ in c waiting for a term t to fill the hole $[\,]$.

For symmetry of notation, let's write $\tilde{\mu}$ for this operator.

$$\frac{c : (\Gamma, x : B \vdash \Delta)}{\Gamma; \tilde{\mu}x.c : B \vdash \Delta}$$

Then $t \star \tilde{\mu}x.c$ is just a notation for let $x = t$ in c.

Remark: the `let-in` operator is crucial to describe the reduction and the normal forms of call-by-value λ-calculus.

The dual of "throw"

It remains to interpret the rule unfocusing a distinguished hypotheses.

$$\frac{\Gamma ; e : B \vdash \Delta}{? : (\Gamma, x : B \vdash \Delta)}$$

It is just building an applicative subterm xe.

Actually, the latter rule can be derived using the cut rule

$$\frac{\overline{\Gamma, x : A \vdash ; x : A} \qquad \Gamma ; e : A \vdash \Delta}{x \star e : (\Gamma \vdash \Delta)}$$

which suggests to use the notation $x \star e$.

The full interpretation of sequent calculus

With `let-in`, we now get a sequent calculus able to interpret both call-by-name and call-by-value classical λ-calculus.

Its syntax (called $\overline{\lambda}\mu\tilde{\mu}$-calculus) is

$$\begin{array}{rcl} t & ::= & \lambda x.t \mid \mu\alpha.c \mid x \\ e & ::= & t \cdot e \mid \tilde{\mu}x.c \mid \alpha \\ c & ::= & t \star e \end{array}$$

and the rules are

$$\frac{}{\Gamma; \alpha : A \vdash \Delta, \alpha : A} \qquad \frac{}{\Gamma, x : A \vdash \Delta; x : A}$$

$$\frac{\Gamma \vdash \Delta; t : A \qquad \Gamma; e : B \vdash \Delta}{\Gamma; t \cdot e : A \to B \vdash \Delta} \qquad \frac{\Gamma, x : A \vdash \Delta; t : B}{\Gamma \vdash \Delta; \lambda x.t : A \to B}$$

$$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma; \tilde{\mu}x.c : A \vdash \Delta} \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \Delta; \mu\alpha.c : A}$$

$$\frac{\Gamma \vdash ; t : A \qquad \Gamma; e : A \vdash \Delta}{t \star e : (\Gamma \vdash \Delta)}$$

The full interpretation of sequent calculus inside λ-calculus + $\mathcal{C}$ + `let-in`

The corresponding presentation of λ-calculus is

$$\begin{array}{lcl} t & ::= & \lambda x.t \mid \mathcal{C}(\lambda\alpha.c) \mid x \\ e[\,] & ::= & e[[\,]t] \mid \text{let } x = [\,] \text{ in } c \mid \alpha[\,] \\ c & ::= & e[t] \end{array}$$

with typing rules

$$\frac{}{\Gamma; \alpha[\,] : A \vdash \Delta, \alpha : A} \qquad \frac{}{\Gamma, x : A \vdash \Delta; x : A}$$

$$\frac{\Gamma \vdash \Delta; t : A \qquad \Gamma; e : B \vdash \Delta}{\Gamma; e[[\,]t] : A \to B \vdash \Delta} \qquad \frac{\Gamma, x : A \vdash \Delta; t : B}{\Gamma \vdash \Delta; \lambda x.t : A \to B}$$

$$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma; \text{let } x = [\,] \text{ in } c : A \vdash \Delta} \qquad \frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \Delta; \mathcal{C}(\lambda\alpha.c) : A}$$

$$\frac{\Gamma \vdash ; t : A \qquad \Gamma; e : A \vdash \Delta}{e[t] : (\Gamma \vdash \Delta)}$$

Reduction rules

$$
\begin{array}{lll}
\lambda x.t \star u \cdot e & \longrightarrow & u \star \tilde{\mu} x.(t \star e) \quad (\beta) \\
\mu\alpha.c \star e & \longrightarrow & c[\alpha := e] \quad (\mu) \\
t \star \tilde{\mu} x.c & \longrightarrow & c[x := t] \quad (\tilde{\mu})
\end{array}
$$

or in λ-calculus + $\mathcal{C}$ + `let` syntax

$$
\begin{array}{lll}
E[(\lambda x.t)u] & \longrightarrow & \text{let } x = u \text{ in } E[t] \\
E[\mathcal{C}(\lambda\alpha.c)] & \longrightarrow & c[\alpha\bullet := E[\bullet]] \\
\text{let } x = t \text{ in } c & \longrightarrow & c[x := t]
\end{array}
$$

There is a critical pair!

Let ϕ be the translation from $\overline{\lambda}\mu\tilde{\mu}$-calculus to λ-calculus + $\mathcal{C}$ + `let` following the above interpretation.

Lemma Giving priority to $\tilde{\mu}$, then $\lambda \vdash \phi(t) \longrightarrow_{CBN} \phi(t')$ implies $\overline{\lambda} \vdash t \xrightarrow{*} t'$

Lemma Giving priority to μ, then $\lambda \vdash \phi(t) \longrightarrow_{CBV} \phi(t')$ implies $\overline{\lambda} \vdash t \xrightarrow{*} t'$

Conclusions

- In sequent calculus, terms and continuations are syntactically dual objects

- The `let-in` is dual to μ (i.e. $\mathcal{C}(\lambda k...)$)

- Sequent calculus structure is a superposition of call-by-value and call-by-name λ-calculi

- In sequent calculus, call-by-name and call-by-value are syntactically dual

Bibliography

Part I

- Griffin, *A formulae-as-types notion of control*, POPL'90.

- Sabry-Felleisen, *Reasoning about programs in cps-style*, Lisp and symbolic computation 6, 1993.

- Coquand, *Computational content of classical logic*, `www.cs.chalmers.se/ coquand`.

- Parigot, *$\lambda\mu$-calculus: An Algorithmic interpretation of classical natural deduction*, LPAR'92.

Part II

- Barbanera-Berardi, *A Symmetric λ-Calculus for "Classical" Program*, Information and Computation 125, 1996.

- Curien-Herbelin, *The Duality of Computation*, ICFP'00.