Non deterministic classical logic: the $\lambda\mu^{++}$-calculus

Karim Nour

Non deterministic classical logic: the $\lambda\mu^{++}$-calculus

Karim NOUR
LAMA - Equipe de Logique,
Université de Savoie
73376 Le Bourget du Lac
e-mail nour@univ-savoie.fr

Abstract

In this paper, we present an extension of $\lambda\mu$-calculus called $\lambda\mu^{++}$-calculus which has the following properties: subject reduction, strong normalization, unicity of the representation of data and thus confluence only on data types. This calculus allows also to program the parallel-or.

1 Introduction

There are now many type systems which are based on classical logic ; among the best known are the system LC of J.-Y. Girard [2], the $\lambda\mu$-calulus of M. Parigot [6], the λ_c-calculus of J.-L. Krivine [3] and the λ^{Sym}-calculus of F. Barbanera and S. Berardi [1]. We consider here the $\lambda\mu$-calculus because it has very good properties: confluence, subject reduction and strong normalization. On the other hand, we lose in this system the unicity of the representation of data. Indeed, there are normal closed terms, different from Church integers, typable by integer type (they are called classical integers). The solutions which were proposed to solve this problem consisted in giving algorithms to find the value of classical integers ([5],[7]). Moreover the presentation of typed $\lambda\mu$-calculus is not very natural. For example, we do not find a closed $\lambda\mu$-term of type $\neg\neg A \to A$. In this paper, we present an extension of $\lambda\mu$-calculus called $\lambda\mu^{++}$-calculus which codes exactly the second order classical natural deduction. The system we propose contains a non deterministic simplification rule which allows a program to be reduced to one of its subroutines. This rule can be seen as a complicated `garbage collector`. This calculus which we obtain has the following properties: subject reduction, strong normalization, unicity of the representation of data and thus confluence only on data types. This calculus allows also to program the parallel-or.

Acknowledgement. We wish to thank C. Raffalli for helpful discussions. We do not forget the numerous corrections and suggestions by N. Bernard.

2 $\lambda\mu$-calculus

2.1 Pure $\lambda\mu$-calculus

$\lambda\mu$-calculus has two distinct alphabets of variables: the set of λ-variables $x, y, z, ...$, and the set of μ-variables α, β, γ,.... Terms (also called $\lambda\mu$-terms) are defined by the following grammar:

$$t := x \mid \lambda x\, t \mid (t\ t) \mid \mu\alpha\, [\beta]t$$

The reduction relation of $\lambda\mu$-calculus is induced by fives different notions of reduction :

The computation rules

$$\begin{array}{rcll} (\lambda x\, u\ v) & \to & u[x := v] & (c_\lambda) \\ (\mu\alpha\, u\ v) & \to & \mu\alpha\, u[\alpha :=^* v] & (c_\mu) \end{array}$$

where $u[\alpha :=^* v]$ is obtained from u by replacing inductively each subterm of the form $[\alpha]w$ by $[\alpha](w\ v)$

The simplification rules

$$\begin{array}{rcll} [\alpha]\mu\beta\, u & \to & u[\beta := \alpha] & (s_1) \\ \mu\alpha\, [\alpha]u & \to & u \quad (*) & (s_2) \\ \mu\alpha\, u & \to & \lambda x\, \mu\alpha\, u[\alpha :=^* x] \quad (**) & (s_3) \end{array}$$

(*) if α has no free occurence in u
(**) if u contains a subterm of the form $[\alpha]\lambda y\, w$

For any $\lambda\mu$-terms t, t', we shall write:
– $t \to_\mu^n t'$ if t' is obtained from t by applying n times these rules.
– $t \to_\mu t'$ if there is $n \in \mathbb{N}$ such that $t \to_\mu^n t'$.

We have the following result ([6],[9]):

Theorem 2.1 *In $\lambda\mu$-calculus, the reduction $\to_\mu$ is confluent.*

2.2 Typed $\lambda\mu$-calculus

Proofs are written in a second order natural deduction system with several conclusions, presented with sequents. The connectives we use are $\perp$, $\to$ and $\forall$. We denote by $A_1, A_2, ..., A_n \to A$ the formula $A_1 \to (A_2 \to (...(A_n \to A)...))$. We do not suppose that the language has a special constant for equality. Instead, we define the formula $a = b$ (where a, b are terms) to be $\forall X\, (X(a) \to X(b))$

where X is a unary predicate variable. Let E be a set of equations. We denote by $a \approx_E b$ the equivalence binary relation such that : if $a = b$ is an equation of E, then $a[x_1 := t_1, ..., x_n := t_n] \approx_E b[x_1 := t_1, ..., x_n := t_n]$.

Let t be a $\lambda\mu$-term, A a type, $\Gamma = x_1 : A_1, ..., x_n : A_n$, $\Delta = \alpha_1 : B_1, ..., \alpha_m : B_m$ are two contexts and E a set of equations. The notion "t is of type A in Γ and Δ with respect to E" (denoted by $\Gamma \vdash t : A, \triangle$) is defined by the following rules:

(1) $\Gamma \vdash x_i : A_i, \triangle$ $(1 \leq i \leq n)$
(2) If $\Gamma, x : A \vdash t : B, \triangle$, then $\Gamma \vdash \lambda x\, t : A \to B, \triangle$
(3) If $\Gamma_1 \vdash u : A \to B, \triangle_1$, and $\Gamma_2 \vdash v : A, \triangle_2$, then $\Gamma_1, \Gamma_2 \vdash (u\ v) : B, \triangle_1, \triangle_2$
(4) If $\Gamma \vdash t : A, \triangle$, and x not free in Γ and $\triangle$, then $\Gamma \vdash t : \forall x\, A, \triangle$
(5) If $\Gamma \vdash t : \forall x\, A, \triangle$, then, for every term a, $\Gamma \vdash t : A[x := a], \triangle$
(6) If $\Gamma \vdash t : A, \triangle$, and X is not free in Γ and $\triangle$, then $\Gamma \vdash t : \forall X\, A, \triangle$
(7) If $\Gamma \vdash t : \forall X\, A, \triangle$, then, for every formula G, $\Gamma \vdash t : A[X := G], \triangle$
(8) If $\Gamma \vdash t : A[x := a], \triangle$, and $a \approx_E b$, then $\Gamma \vdash t : A[x := b], \triangle$
(9) If $\Gamma \vdash t : A, \beta : B, \triangle$, then :
– $\Gamma \vdash \mu\beta\,[\alpha]t : B, \alpha : A, \triangle$ if $\alpha \neq \beta$
– $\Gamma \vdash \mu\alpha\,[\alpha]t : B, \triangle$ if $\alpha = \beta$

The typed $\lambda\mu$-calculus has the following properties ([6],[8]):

Theorem 2.2
1) **Subject reduction:** *Type is preserved during reduction.*
2) **Strong normalization:** *Typable $\lambda\mu$-terms are strongly normalizable.*

2.3 Representation of data types

Each data type generated by free algebras can be defined by a second order formula. The type of boolean is the formula $\mathrm{Bool}[x] = \forall X\,\{X(\mathbf{1}), X(\mathbf{0}) \to X(x)\}$ where $\mathbf{0}$ and $\mathbf{1}$ are constants. The type of integers is the formula $\mathrm{Ent}[x] = \forall X\,\{X(0), \forall y\,(X(y) \to X(sy)) \to X(x)\}$ where 0 is a constant symbol for zero, and s is a unary function symbol for successor.

In the rest of this paper, we suppose that every set of equations E satisfies the following properties: $\mathbf{0} \not\approx_E \mathbf{1}$ and if $n \neq m$, then $s^n(0) \not\approx_E s^m(0)$

We denote by $\underline{id} = \lambda x\, x$, $\underline{\mathbf{1}} = \lambda x \lambda y\, x$, $\underline{\mathbf{0}} = \lambda x \lambda y\, y$ and, for every $n \in \mathbb{N}$, $\underline{n} = \lambda x \lambda y\, (y^n\ x)$ (where $(y^0\ x) = x$ and $(y^{k+1}\ x) = (y\ (y^k\ x))$). It is easy to see that:

Lemma 2.1
1) $\vdash \underline{\mathbf{1}} : Bool[\mathbf{1}]$ *and* $\vdash \underline{\mathbf{0}} : Bool[\mathbf{0}]$.
2) For every $n \in \mathbb{N}$, $\vdash \underline{n} : Ent[s^n(0)]$.

The converse of (1) lemma 2.1 is true.

Lemma 2.2 *If* $\mathbf{b} \in \{\mathbf{0}, \mathbf{1}\}$ *and* $\vdash t : Bool[\mathbf{b}]$, *then* $t \rightarrow_\mu \underline{\mathbf{b}}$.

But the converse of (2) lemma 2.1 is not true. Indeed, if we take the closed normal term $\theta = \lambda x \lambda f\, \mu\alpha\, [\alpha](f\ \mu\beta\, [\alpha](f\ x))$, we have $\vdash \theta : \mathrm{Ent}[s(0)]$.

3 $\lambda\mu^{++}$-calculus

3.1 Pure $\lambda\mu^{++}$-calculus

The set of $\lambda\mu^{++}$-terms is given by the following grammar:

$$t := x \mid \alpha \mid \lambda x\, t \mid \mu\alpha\, t \mid (t\ t)$$

where x ranges over a set V_λ of λ-variables and α ranges over a set V_μ of μ-variables disjoint from V_λ.
The reduction relation of $\lambda\mu^{++}$-calculus is induced by eight notions of reduction:

The computation rules

$$(\lambda x\, u\ v) \rightharpoonup u[x := v] \qquad (C_\lambda)$$
$$(\mu\alpha\, u\ v) \rightharpoonup \mu\beta u\, [\alpha := \lambda y\, (\beta\ (y\ v))] \qquad (C_\mu)$$

The local simplification rules

$$((\alpha\ u)\ v) \rightharpoonup (\alpha\ u) \qquad (S_1)$$
$$\mu\alpha\mu\beta\, u \rightharpoonup \mu\alpha\, u[\beta := \mathbf{id}] \qquad (S_2)$$
$$(\alpha\ (\beta\ u)) \rightharpoonup (\beta\ u) \qquad (S_3)$$
$$(\beta\ \mu\alpha\, u) \rightharpoonup u[\alpha := \lambda y\, (\beta\ y)] \qquad (S_4)$$

The global simplification rules

$$\mu\alpha\, u \rightharpoonup \lambda z\, \mu\beta\, u[\alpha := \lambda y\, (\beta\ (y\ z))] \quad (*) \qquad (S_5)$$
$$\mu\alpha\, u[y := (\alpha\ v)] \rightharpoonup v \quad (**) \qquad (S_6)$$

(*) if u contains a subterm of the form $(\alpha\ \lambda x\, v)$
(**) if y is free in u and α is not free in v

For any $\lambda\mu^{++}$-terms t, t', we shall write
– $t \rightharpoonup^n_{\mu^{++}} t'$ if t' is obtained from t by applying n times these rules.
– $t \rightharpoonup_{\mu^{++}} t'$ if there is $n \in \mathbb{N}$ such that $t \rightharpoonup^n_{\mu^{++}} t'$.

Let us claim first that $\lambda\mu^{++}$-calculus is not confluent. Indeed, if we take $u = \lambda x\, \mu\alpha\, ((x\ \ (\alpha\ \ \underline{\mathbf{0}}))\ \ (\alpha\ \ \underline{\mathbf{1}}))$, we have (using rule S_6) $u \rightharpoonup_{\mu^{++}} \lambda x\, \underline{\mathbf{0}}$ and

$u \rightharpoonup_{\mu^{++}} \lambda x\, \underline{\mathbf{1}}$. The non confluence of $\lambda\mu^{++}$-calculus does not come only from rule S_6. Indeed, if we take $v = \mu\alpha\,((\alpha\ \mu\beta\,\beta)\underline{\mathbf{0}})$, we have $v \rightharpoonup_{\mu^{++}} \mu\alpha\lambda y\,(\alpha\ y)$ and $v \rightharpoonup_{\mu^{++}} \underline{\mathbf{0}}$.

The rules which are really new compared to $\lambda\mu$-calculus are S_1 and S_6. The rule S_1 means that the μ-variables are applied to more than one term. We will see that typing will ensure this condition. The rule S_6 means that if $\mu\alpha\, t$ has a subterm $(\alpha\ v)$ where v does not contain free variables which are bounded in $\mu\alpha\, t$, then we can return v as result. This results in the possibility of making a parallel computation. It is clear that this rule is very difficult to implement. But for the examples and the properties we will present, the condition "not active binders between $\mu\alpha$ and α" will be enough. Let us explain how we can implement the weak version of this rule. We suppose that the syntax of the terms has two λ-abstractions: λ and λ' and two μ-abstractions: μ and μ'. We write $\lambda' x\, u$ and $\mu'\alpha\, u$ only if the variables x and α do not appear in u. We suppose also that for each μ-variable α we have a special symbol ξ_α. We can thus simulate the weak version of rule S_6 by the following non deterministic rules:

$$\begin{array}{rcll} \mu\alpha\, u & \rightharpoonup & (\xi_\alpha\ u) & \\ (\xi_\alpha\ \lambda' x\, u) & \rightharpoonup & (\xi_\alpha\ u) & \\ (\xi_\alpha\ \mu'\beta\, u) & \rightharpoonup & (\xi_\alpha\ u) & \\ ((\xi_\alpha\ (\alpha\ v)) & \rightharpoonup & v & \\ ((\xi_\alpha\ (u\ v)) & \rightharpoonup & (\xi_\alpha\ u) & (*) \\ ((\xi_\alpha\ (u\ v)) & \rightharpoonup & (\xi_\alpha\ v) & (*) \end{array}$$

(*) $u \neq \alpha$
A result of a computation is a term which does not contain symbols ξ_α.

We will see that with the exception of rule S_6 the $\lambda\mu^{++}$-calculus is not different from $\lambda\mu$-calculus. We will establish codings which make it possible to translate each one in to the other.

3.2 Relation between $\lambda\mu$- calculus and $\lambda\mu^{++}$- calculus

We add to $\lambda\mu$-calculus the equivalent version of rule S_6:

$$\mu\alpha\,[\beta]u[y := [\alpha]v] \rightarrow' v$$

if y is free in u and α is not free in v.
We denote by $\lambda\mu^{+}$-calculus this new calculus.

For any $\lambda\mu$-terms t, t', we shall write :
– $t \rightarrow^n_{\mu^+} t'$ if t' is obtained from t by applying n times these rules.

– $t \rightarrow_{\mu^+} t'$ if there is $n \in \mathbb{N}$ such that $t \rightarrow^n_{\mu^+} t'$.

For each $\lambda\mu$-term t we define a $\lambda\mu^{++}$-term t^* in the following way:

$$\begin{array}{rcl} x^* & = & x \\ \{\lambda x\, t\}^* & = & \lambda x\, t^* \\ \{(u\ v)\}^* & = & (u^*\ v^*) \\ \{\mu\alpha\,[\beta]t\}^* & = & \mu\alpha\,(\beta\ t^*) \end{array}$$

We have the following result:

Theorem 3.1 *Let u, v be $\lambda\mu$-terms. If $u \rightarrow^n_{\mu^+} v$, then there is $m \geq n$ such that $u^* \rightharpoonup^m_{\mu^{++}} v^*$.*

Proof Easy. □

The converse of this coding is much more difficult to establish because it is necessary to include the reductions of administrative redexes. We first modify slightly the syntax of the $\lambda\mu^{++}$-calculus. We suppose that we have a particular μ-constant δ (i.e. $\mu\delta\, u$ is not a term) and two other λ-abstractions: λ^1 and λ^2. The only terms build with these abstractions are: $\lambda^1 xu$ where u contains only one occurence of x and $\lambda^2 xx$. For the rule C_μ, λ, λ^1 and λ^2 behave in the same way. We write rules C_μ, S_2, S_4 and S_5 in the following way:

$$\begin{array}{rcll} (\mu\alpha\, u\ v) & \rightharpoonup & \mu\beta\, u[\alpha := \lambda^1 y\,(\beta\ (y\ v))] & (C_\mu) \\ \mu\alpha\mu\beta\, u & \rightharpoonup & \mu\alpha\, u[\beta := \lambda^2 x\, x] & (S_2) \\ (\beta\ \mu\alpha\, u) & \rightharpoonup & u[\alpha := \lambda^1 y\,(\beta\ y)] & (S_4) \\ \mu\alpha\, u & \rightharpoonup & \lambda z\mu\beta\, u[\alpha := \lambda^1 y\,(\beta\ (y\ z))] & (S_5) \end{array}$$

It is clear that the new $\lambda\mu^{++}$-calculus is stable by reductions.

For each $\lambda\mu^{++}$-term t we define a $\lambda\mu$-term t° in the following way :

$$\begin{array}{rcll} x^\circ & = & x & \\ \alpha^\circ & = & \lambda x\mu\gamma\,[\alpha]x & (*) \\ \{\lambda x\, t\}^\circ & = & \lambda x\, t^\circ & \\ \{\lambda^1 x\, t\}^\circ & = & \lambda x\, t^\circ & \\ \{\lambda^2 x\, x\}^\circ & = & \lambda x\mu\gamma\,[\delta]x & \\ \{\mu\alpha\, t\}^\circ & = & \mu\alpha\,[\delta]t^\circ & \\ \{(\lambda^1 x\, u\ v)\}^\circ & = & u^\circ[x := v^\circ] & \\ \{(\lambda^2 x\, x\ v)\}^\circ & = & \mu\gamma\,[\delta]v^\circ & (**) \\ \{(u\ v)\}^\circ & = & (u^\circ\ v^\circ) & (***) \end{array}$$

(*) $\gamma \neq \alpha$
(**) γ is not free in v°
(***) $u \neq \lambda^i x\, w$ $i \in \{1, 2\}$

We have the following result:

Theorem 3.2 *Let u, v be $\lambda\mu^{++}$-terms. If $u \rightharpoonup^n_{\mu^{++}} v$, then there is $m \geq n$ and a $\lambda\mu$-term w such that $u^\circ \to^m_{\mu^+} w$ and $v^\circ \to_{\mu^+} w$.*

Proof We use the confluence of $\lambda\mu$-calculus and the following lemma:

Lemma 3.1 *Let u, v be $\lambda\mu^{++}$-terms.*
1) $\{u[x := v]\}^\circ \to_{\mu^+} u^\circ[x := v^\circ]$.
2) $\{u[\alpha := \lambda^1 y\, (\beta\ (y\ v))]\}^\circ \to_{\mu^+} u^\circ[\alpha :=^* v^\circ]$. □

We deduce the following corollary:

Corollary 3.1 *Let u be a $\lambda\mu^{++}$-term. If u° is strongly normalizable then u is also strongly normalizable.*

3.3 Typed $\lambda\mu^{++}$-calculus

Types are formulas of second order predicate logic constructed from $\perp$, $\to$ and $\forall$. For every formula A, we denote by $\neg A$ the formula $A \to\perp$ and by $\exists x\, A$ the formula $\neg\forall x\, \neg A$. Proofs are written in the ordinary classical natural deduction system.

Let t be a $\lambda\mu^{++}$-term, A a type, $\Gamma = x_1 : A_1, ..., x_n : A_n, \alpha_1 : \neg B_1, ..., \alpha_m : \neg B_m$ a context, and E a set of equations. We define the notion "t is of type A in Γ with respect to E" (denoted by $\Gamma \vdash' t : A$) by means of the following rules

(1) $\Gamma \vdash' x_i : A_i$ $(1 \leq i \leq n)$ and $\Gamma \vdash' \alpha_j : \neg B_j$ $(1 \leq j \leq m)$.
(2) If $\Gamma, x : A \vdash' u : B$, then $\Gamma \vdash' \lambda x\, u : A \to B$.
(3) If $\Gamma_1 \vdash' u : A \to B$, and $\Gamma_2 \vdash' v : A$, then $\Gamma_1, \Gamma_2 \vdash' (u\ v) : B$.
(4) If $\Gamma \vdash' u : A$, and x is not free in Γ, then $\Gamma \vdash' u : \forall x\, A$.
(5) If $\Gamma \vdash' u : \forall x\, A$, then, for every term a, $\Gamma \vdash' u : A[x := a]$.
(6) If $\Gamma \vdash' u : A$, and X is not free in Γ, then $\Gamma \vdash' u : \forall X\, A$.
(7) If $\Gamma \vdash' u : \forall X\, A$, then, for every formulas G, $\Gamma \vdash' u : A[X := G]$.
(8) If $\Gamma \vdash' u : A[x := a]$, and $a \approx_E b$, then $\Gamma \vdash' u : A[x := b]$.
(9) If $\Gamma, \alpha : \neg B \vdash' u :\perp$, then $\Gamma \vdash' \mu\alpha\, u : B$.

Consequently, we can give more explanations for rule S_6. It means that "in a proof of a formula we cannot have a subproof of the same formula". The terms $\mu\alpha\, u[y := (\alpha\ v)]$ and v has the same type, then the rule S_6 authorizes a program to be reduced to one of its subroutines which has the same behaviour.

If $\triangle = \alpha_1 : B_1, ..., \alpha_m : B_m$, then we denode by $\neg\triangle = \alpha_1 : \neg B_1, ..., \alpha_m : \neg B_m$. If $\Gamma = x_1 : A_1, ..., x_n : A_n, \alpha_1 : \neg B_1, ..., \alpha_m : \neg B_m$, then we denote by $\Gamma_\lambda = x_1 : A_1, ..., x_n : A_n$ and $\Gamma_\mu = \alpha_1 : B_1, ..., \alpha_m : B_m$.

We have the following results:

Theorem 3.3
1) If $\Gamma \vdash t : A, \triangle$, then $\Gamma, \neg\triangle \vdash' t^* : A$.
2) If $\Gamma \vdash' t : A$, then $\Gamma_\lambda \vdash t^\circ : A, \Gamma_\mu, \delta : \perp$

Proof By induction on typing. □

4 Theoretical properties of $\lambda\mu^{++}$-calculus

Theorem 4.1 (Subject reduction)
If $\Gamma \vdash' u : A$ and $u \rightharpoonup v$, then $\Gamma \vdash' v : A$.

Proof It suffices to verify that the reduction rules are well typed. □

Theorem 4.2 (Strong normalization)
If $\Gamma \vdash' u : A$, then u is strongly normalizable.

Proof According to the theorem 3.3 and the corollary 3.1, it is enough to show that the $\lambda\mu^+$-calculus is strongly normalizable. It is a direct consequence of the theorem 2.2 and the following lemma:

Lemma 4.1 *Let u, v, w be $\lambda\mu$-terms. If $u \to' v \to_\mu^n w$ then there is $m \geq n$ and a $\lambda\mu$-term v' such that $u \to_\mu^m v' \to' w$.* □

Let t be a $\lambda\mu^{++}$-term and $\mathcal{V}_t$ a set of normal $\lambda\mu^{++}$-terms. We write $t \to_{\mu^{++}} \mathcal{V}_t$ iff:
– for all $u \in \mathcal{V}_t$, $t \rightharpoonup_{\mu^{++}} u$.
– If $t \rightharpoonup_{\mu^{++}} u$ and u is normal, then $u \in \mathcal{V}_t$.
Intuitively $\mathcal{V}_t$ is the set of values of t.

Theorem 4.3 (Unicity of representation of integers)
If $n \in \mathbb{N}$ and $\vdash' t : Ent[s^n(0)]$, then $t \to_{\mu^{++}} \{\underline{n}\}$.

Proof Let t be a closed normal term such that $\vdash' t : \mathrm{Ent}[s^n(0)]$. Since we cannot use rules S_4 and S_5, we prove that $t = \lambda x \lambda f\, u$ and $x : X(0), f : \forall y\, (X(y) \to X(s(y))) \vdash' u : X(s^n(0))$. The term u does not contain μ-variables. Indeed, if not, we consider a subterm $(\alpha\ v)$ of u such that v does not contain μ-variables. It is easy to see that v is of the form $(f^m\ x)$, thus u is not normal (we can apply rule S_6). Therefore $u = (f^n\ x)$ and $t = \underline{n}$. □

5 Some programs in $\lambda\mu^{++}$-calculus

5.1 Classical programs

Let $\mathcal{I} = \lambda x \mu\alpha\, x$, $\mathcal{C} = \lambda x \mu\alpha\, (x\ \alpha)$ and $\mathcal{P} = \lambda x \mu\alpha\, (\alpha\ (x\ \alpha))$. It is easy to check that:

Theorem 5.1
1) $\vdash' \mathcal{I} : \forall X\, \{\perp \to X\}$, *and, for every* $t, t_1, ..., t_n$, $(\mathcal{I}\ t\ t_1...t_n) \rightharpoonup_{\mu^{++}} \mu\alpha\, t$.
2) $\vdash' \mathcal{C} : \forall X\, \{\neg\neg X \to X\}$, *and, for every* $t, t_1, ..., t_n$, $(\mathcal{C}\ t\ t_1...t_n) \rightharpoonup_{\mu^{++}}$ $\mu\alpha\, (t\ \lambda y\, (\alpha\ (y\ t_1...t_n)))$.
3) $\vdash' \mathcal{P} : \forall X\, \{(\neg X \to X) \to X\}$, *and, for every* $t, t_1, ..., t_n$, $(\mathcal{P}\ t\ t_1...t_n) \rightharpoonup_{\mu^{++}}$ $\mu\alpha\, (\alpha\ (t\ \lambda y\, (\alpha\ (y\ t_1...t_n)))\ t_1...t_n)$.

Let us note that the $\lambda\mu^{++}$-term $\mathcal{I}$ simulates the `exit` instruction of `C` programming language and the $\lambda\mu^{++}$-term $\mathcal{P}$ simulates the `Call/cc` instruction of the `Scheme` functional language (see [4]).

5.2 Producers of integers

For every $n_1, ..., n_m \in \mathbb{N}$, we define the following finite sequence $(U_k)_{1 \le k \le m}$:
$U_k = (\alpha\ (x\ \lambda d \lambda y\, (y\ \underline{n_k})\ \mathbf{id}\ (\mathcal{I}\ U_{k-1})))\ (2 \le k \le m)$
and $U_1 = (\alpha\ (x\ \lambda d \lambda y\, (y\ \underline{n_1})\ \mathbf{id}\ \alpha))$.
Let $P_{n_1,...,n_m} = \lambda x \mu\alpha\, U_m$. We have:

Theorem 5.2 $\vdash' P_{n_1,...,n_m} : \forall x\, \{Ent[x] \to \exists y\, Ent[y]\}$, *and* $(P_{n_1,...,n_m}\ \underline{0}) \to_{\mu^{++}}$ $\{\lambda y\, (y\ \underline{n_i})\ ;\ 1 \le i \le m\}$.

Proof For the typing, it suffices to prove that $x : \text{Ent}[x], \alpha : \neg\exists y\, \text{Ent}[y] \vdash'$ $\lambda d \lambda y\, (y\ \underline{n_k}) : \neg\exists y\, \text{Ent}[y] \to \exists y\, \text{Ent}[y]\ (1 \le k \le m)$ and thus $x : \text{Ent}[x], \alpha :$ $\neg\exists y\, \text{Ent}[y] \vdash' U_k : \perp\ (1 \le k \le m)$.
We define the following finite sequence $(V_k)_{1 \le k \le m}$:
$V_k = (\alpha\ (\lambda d \lambda y\, (y\ \underline{n_k})\ (\mathcal{I}\ V_{k-1})))\ (2 \le k \le m)$ and $V_1 = (\alpha\ \lambda y\, (y\ \underline{n_1}))$.
We have $(P_{n_1,...,n_m}\ \underline{0}) \rightharpoonup_{\mu^{++}} \lambda x \mu\alpha\, V_m \rightharpoonup_{\mu^{++}} \lambda y\, (y\ \underline{n_i})\ (1 \le i \le m)$. □

Let $P_{\mathbb{N}} = (Y\ F)$ where
$F = \lambda x \lambda y \mu\alpha\, (\alpha\ (y\ \lambda d\, (x\ (\underline{s}\ y))\ \mathbf{id}\ (\mathcal{I}\ (\alpha\ (y\ \lambda d \lambda z\, (z\ y)\ \mathbf{id}\ \alpha)))))$, Y is the Turing fixed point and $\underline{s}$ a $\lambda\mu^{++}$-term for successor on Church integers. It is easy to check that:

Theorem 5.3 $(P_{\mathbb{N}}\ \underline{0}) \to_{\mu^{++}} \{\lambda y\, (y\ \underline{m})\ ;\ m \in \mathbb{N}\}$.

We can check that $\vdash' F : \forall x\, \{\text{Ent}[x] \to \exists y\, \text{Ent}[y]\} \to \forall x\, \{\text{Ent}[x] \to \exists y\, \text{Ent}[y]\}$. Therefore, if we add to the typed system the following rule:

$$\text{If } \Gamma \vdash' F : A \to A, \text{ then } \Gamma \vdash' (Y\ F) : A$$

we obtain $\vdash' P_{\mathbb{N}} : \forall x \, \{\text{Ent}[x] \rightarrow \exists y \, \text{Ent}[y]\}$.
It is clear that, with this rule, we lose the strong normalization property. But we possibly can put restrictions on this rule to have weak normalization.

We can deduce the following corollary:

Corollary 5.1 *Let $\mathcal{R} \subseteq \mathbb{N}$ be a recursively enumerable set. There is a closed normal $\lambda\mu^{++}$-term $P_{\mathcal{R}}$ such that $(P_{\mathcal{R}} \; \underline{0}) \rightarrow_{\mu^{++}} \{\underline{m} \; ; \, m \in \mathcal{R}\}$.*

5.3 Parallel-or

Let $\mathcal{TB} = \{b \; ; \, b \rightarrow_{\mu^{++}} \{\underline{\mathbf{0}}\}$ or $b \rightarrow_{\mu^{++}} \{\underline{\mathbf{1}}\}\}$ the set of true booleans.

A closed normal $\lambda\mu^{++}$-term b is said to be a false boolean iff :
$b \not\rightharpoonup_{\mu^{++}} \lambda x \, u$
or
$b \rightharpoonup_{\mu^{++}} \lambda x \, u$ where $u \not\rightharpoonup_{\mu^{++}} \lambda y \, v$ and $u \not\rightharpoonup_{\mu^{++}} (x \; v_1 ... v_n)$
or
$b \rightharpoonup_{\mu^{++}} \lambda x \lambda y \, u$ where $u \not\rightharpoonup_{\mu^{++}} \lambda y \, v$, $u \not\rightharpoonup_{\mu^{++}} (x \; w_1 ... w_n)$ and $u \not\rightharpoonup_{\mu^{++}} (y \; w_1 ... w_n)$.

We denote $\mathcal{FB}$ the set of false booleans. Intuitively a false boolean is thus a term which can give the first informations on a true boolean before looping.

Let $\mathcal{B} = \mathcal{TB} \cup \mathcal{FB}$ the set of booleans.

We said that a closed normal $\lambda\mu^{++}$-term T is a parallel-or iff for all $b_1, b_2 \in \mathcal{B}$:
$(T \; b_1 \; b_2) \rightarrow_{\mu^{++}} \{\underline{\mathbf{0}}, \underline{\mathbf{1}}\}$;
$(T \; b_1 \; b_2) \rightharpoonup_{\mu^{++}} \underline{\mathbf{1}}$ iff $b_1 \rightarrow_{\mu^{++}} \underline{\mathbf{1}}$ or $b_2 \rightarrow_{\mu^{++}} \underline{\mathbf{1}}$;
$(T \; b_1 \; b_2) \rightharpoonup_{\mu^{++}} \underline{\mathbf{0}}$ iff $b_1 \rightarrow_{\mu^{++}} \underline{\mathbf{0}}$ and $b_2 \rightarrow_{\mu^{++}} \underline{\mathbf{0}}$.

Let *or* be a binary function defined by the following set of equations :

$$or(\mathbf{1}, x) = \mathbf{1} \qquad or(\mathbf{0}, x) = x \qquad or(x, \mathbf{1}) = \mathbf{1} \qquad or(x, \mathbf{0}) = x$$

Let $\bigvee = \lambda x \lambda y \mu \alpha \, (\alpha \; (x \; \widehat{\mathbf{1}} \; (y \; \widehat{\mathbf{1}} \; \widehat{\mathbf{0}}) \; (\mathcal{I} \; (\alpha \; (y \; \widehat{\mathbf{1}} \; (x \; \widehat{\mathbf{1}} \; \widehat{\mathbf{0}}) \; \alpha)))))$ where $\widehat{\mathbf{1}} = \lambda p \, \underline{\mathbf{1}}$ and $\widehat{\mathbf{0}} = \lambda p \, \underline{\mathbf{0}}$.

Theorem 5.4 $\vdash' \bigvee : \forall x \forall y \, \{Bool[x], Bool[y] \rightarrow Bool[or(x, y)]\}$ *and* $\bigvee$ *is a parallel-or.*

Proof Let $B[x] = \neg\text{Bool}[x] \rightarrow \text{Bool}[x]$.
$x : \text{Bool}[x] \vdash' x : B[\mathbf{1}], B[\mathbf{0}] \rightarrow B[x]$, then $x : \text{Bool}[x] \vdash' (x \; \widehat{\mathbf{1}} \; \widehat{\mathbf{0}}) : B[x]$.
In the same way we prove that $y : \text{Bool}[y] \vdash' (y \; \widehat{\mathbf{1}} \; \widehat{\mathbf{0}}) : B[y]$.
$y : \text{Bool}[y] \vdash' y : B[\mathbf{1}], B[x] \rightarrow B[or(x, y)]$, then
$x : \text{Bool}[x], y : \text{Bool}[y] \vdash' (y \; \widehat{\mathbf{1}} \; (x \; \widehat{\mathbf{1}} \; \widehat{\mathbf{0}})) : \text{Bool}[or(x, y)]$, therefore

$\alpha : \neg\mathrm{Bool}[or(x,y)], x : \mathrm{Bool}[x], y : \mathrm{Bool}[y] \vdash' (\alpha\ (y\ \widehat{\mathbf{1}}\ (x\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})\ \alpha))) : \perp$ and
$\alpha : \neg\mathrm{Bool}[or(x,y)], x : \mathrm{Bool}[x], y : \mathrm{Bool}[y] \vdash' (\mathcal{I}\ (\alpha\ (y\ \widehat{\mathbf{1}}\ (x\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})\ \alpha)))) : \neg\mathrm{Bool}[or(x,y)]$.
$x : \mathrm{Bool}[x] \vdash' x : B[\mathbf{1}], B[y] \to B[or(x,y)]$, then
$x : \mathrm{Bool}[x], y : \mathrm{Bool}[y] \vdash' (x\ \widehat{\mathbf{1}}\ (y\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})) : B[or(x,y)]$, therefore
$\alpha : \neg\mathrm{Bool}[or(x,y)], x : \mathrm{Bool}[x], y : \mathrm{Bool}[y] \vdash' (x\ \widehat{\mathbf{1}}\ (y\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})\ (\mathcal{I}\ (\alpha\ (y\ \widehat{\mathbf{1}}\ (x\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})\ \alpha))) : \mathrm{Bool}[or(x,y)]$.
And finally : $\vdash' \bigvee : \forall x \forall y\, \{\mathrm{Bool}[x], \mathrm{Bool}[y] \to \mathrm{Bool}[or(x,y)]\}$.

We will make three examples of reductions. Let $b_1, b_2, b_3 \in \mathcal{B}$ such that $b_1 \to_{\mu^{++}} \{\underline{\mathbf{0}}\}$, $b_2 \to_{\mu^{++}} \{\underline{\mathbf{1}}\}$ and $b_3 \rightharpoonup_{\mu^{++}} \lambda x \lambda y\, u$ where $u \not\rightharpoonup_{\mu^{++}} \lambda y\, v$, $u \not\rightharpoonup_{\mu^{++}} (x\ w_1 ... w_n)$ and $u \not\rightharpoonup_{\mu^{++}} (y\ w_1 ... w_n)$. We will reduce $(\bigvee\ b_1\ b_3)$, $(\bigvee\ b_2\ b_3)$, and $(\bigvee\ b_3\ b_2)$.

The reductions of $R_1 = (b_3\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})$ and $R_2 = (b_3\ \widehat{\mathbf{1}}\ (b_i\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})\ \alpha)))$ do not terminate, and α is free in each R such that $R_2 \rightharpoonup_{\mu^{++}} R$. Therefore, the only way to be compute $(\bigvee\ b_1\ b_3)$ and $(\bigvee\ b_2\ b_3)$ are the following:

$(\bigvee\ b_1\ b_3)$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ (\mathbf{0}\ \widehat{\mathbf{1}}\ R'_1\ (\mathcal{I}\ (\alpha\ R'_2))))$
$\rightharpoonup_{\mu^{++}} ...$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ (R''_1\ (\mathcal{I}\ (\alpha\ R''_2))))$
$\rightharpoonup_{\mu^{++}} ...$
Then the computation does not terminate.

$(\bigvee\ b_2\ b_3)$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ (\mathbf{1}\ \widehat{\mathbf{1}}\ R'_1\ (\mathcal{I}\ (\alpha\ R'_2))))$
$\rightharpoonup_{\mu^{++}} ...$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ \widehat{\mathbf{1}}\ (\mathcal{I}\ (\alpha\ R''_2)))$
$\rightharpoonup_{\mu^{++}} ...$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ \underline{\mathbf{1}}) \rightharpoonup_{\mu^{++}} \underline{\mathbf{1}}$.

The reductions of $R_3 = (b_3\ \widehat{\mathbf{1}}\ (b_2\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}}))$ and $R_4 = (b_3\ \widehat{\mathbf{1}}\ \widehat{\mathbf{0}})$ do not terminate. Therefore, the only way to compute $(\bigvee\ b_3\ b_2)$ is the following:

$(\bigvee\ b_3\ b_2)$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ (R'_1\ (\mathcal{I}\ (\alpha\ ((\mathbf{1}\ \widehat{\mathbf{1}}\ R'_4)\ \alpha)))))$
$\rightharpoonup_{\mu^{++}} ...$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ (R''_1\ (\mathcal{I}\ (\alpha\ (\widehat{\mathbf{1}}\ \alpha)))))$
$\rightharpoonup_{\mu^{++}} ...$
$\rightharpoonup_{\mu^{++}} \mu\alpha\,(\alpha\ (R'''_1\ (\mathcal{I}\ (\alpha\ \underline{\mathbf{1}}))))$
$\rightharpoonup_{\mu^{++}} ...$
$\rightharpoonup_{\mu^{++}} \underline{\mathbf{1}}$. □

References

[1] F. Barbanera and S. Berardi *A symmetric lambda-calculus for classical program extraction.* In M. Hagiya and J.C. Mitchell, editors, Proceedings of theoretical aspects of computer software, volume 789 of LNCS, pp. 495-515. Springer Verlag, 1994.

[2] J.-Y. Girard *A new constructive logic: classical logic.* Mathematical Structures in Computer Science, num 1, pp. 255-296, 1991.

[3] J.-L. Krivine *Classical logic, storage operators and 2nd order lambda-calculus.* Annals of Pure and Applied Logic, num 68, pp. 53-78, 1994.

[4] J.-L. Krivine *About classical logic and imperative programming.* Ann. of Math. and Artif. Intell., num 16, pp. 405-414, 1996.

[5] K. Nour *La valeur d'un entier classique en $\lambda\mu$-calcul.* Archive for Mathematical Logic 36, pp. 461-473, 1997.

[6] M. Parigot *$\lambda\mu$-calculus : an algorithm interpretation of classical natural deduction.* Lecture Notes in Artificial Intelligence, Springer Verlag, num 624, pp. 190-201, 1992.

[7] M. Parigot *Classical proofs as programs.* Lectures Notes in Computer Science, Springer Verlag, num 713, 263-276, 1992.

[8] M. Parigot *Strong normalization for second order classical natural deduction.* Proceedings of the eighth annual IEEE symposium on logic in computer science, pp. 39-46, 1993.

[9] W. Py *Confluence en $\lambda\mu$-calcul.* Thèse de doctorat, Université de Savoie, 1998.