On the Computational Interpretation of Negation

Michel Parigot

Equipe de Logique Mathématique
case 7012, Université Paris 7
2 place Jussieu, 75251 Paris cedex 05, France

Abstract. We investigate the possibility of giving a computational interpretation of an involutive negation in classical natural deduction. We first show why this cannot be simply achieved by adding $\neg\neg A = A$ to typed λ-calculus: the main obstacle is that an involutive negation cannot be a particular case of implication at the computational level. It means that one has to go out typed λ-calculus in order to have a safe computational interpretation of an involutive negation.
We then show how to equip $\lambda\mu$-calculus in a natural way with an involutive negation: the abstraction and application associated to negation are simply the operators μ and $[\,]$ from $\lambda\mu$-calculus. The resulting system is called symmetric $\lambda\mu$-calculus.
Finally we give a translation of symmetric λ-calculus in symmetric $\lambda\mu$-calculus, which doesn't make use of the rule of μ-reduction of $\lambda\mu$-calculus (which is precisely the rule which makes the difference between classical and intuitionistic proofs in the context of $\lambda\mu$-calculus). This seems to indicate that an involutive negation generates an original way of computing. Because symmetric $\lambda\mu$-calculus contains both ways, it should be a good framework for further investigations.

1 Introduction

A lot of efforts have been done in the past 10 years to give computational interpretations of classical logic, starting from the work of Felleisen [5,6], Griffin [9] and Murthy [15]. It has been shown that classical natural deduction allows to modelize imperative features added to functional languages like Scheme, Common Lisp or ML. Two particular systems, λ_C-calculus ([5], [6]) and $\lambda\mu$-calculus ([17]), have been intensively studied and the relation between features of languages, rules of natural deduction, machines and semantics seems to be well understood.

In the context of sequent calculus, several other computational interpretations of classical logic have been constructed following the spirit of Girard's linear logic [7]. It is often claimed in this context that computational interpretations of negation in classical logic should be involutive, that is $\neg\neg A = A$ should be

P. Clote and H. Schwichtenberg (Eds.): CSL 2000, LNCS 1862, pp. 472–484, 2000.

realised at the computational level. It is even sometimes claimed that this is the distinguishing feature of classical logic. But the real computational effect of the involutive character is not clear.

Systems coming from a natural deduction setting, like λ_C-calculus or $\lambda\mu$-calculus, don't have an involutive negation. There is only one exception: the symmetric λ-calculus of Barbanera and Berardi [2,3], which is explicitely based on an involutive negation, but whose concrete programming counterpart is not so well understood.

This paper is devoted to the study of the possibility of having an involutive negation in a computational interpretation of the usual natural deduction system.

In section 2 we discuss in details the possiblity of adding $\neg\neg A = A$ to typed λ-calculus (as a way of adding the classical absurdity rule to intuitionistic natural deduction). We show that there are two obstacles: negation cannot be a particular case of implication and $\bot$ cannot be an atomic type, contrary to the use coming from intuitionistic logic. The fact that negation and implication need to have different computational interpretations means that one has to go out typed λ-calculus in order to have a safe computational interpretation of an involutive negation.

In section 3 we show how to equip $\lambda\mu$-calculus in a natural way with an involutive negation: the abstraction and application associated to negation are simply the operators μ and $[\,]$ from $\lambda\mu$-calculus. The resulting system is called symmetric $\lambda\mu$-calculus.

In section 4 we give a translation of symmetric λ-calculus in symmetric $\lambda\mu$-calculus, which doesn't make use of the rule of μ-reduction of $\lambda\mu$-calculus (which is precisely the rule which makes the difference between classical and intuitionistic proofs in the context of $\lambda\mu$-calculus). This seems to indicate that an involutive negation generates an original way of computing. Because symmetric $\lambda\mu$-calculus contains both ways, it should be a good framework for further investigations.

In the sequel types are designated by letters A, B, C etc., while atomic types are designated by P, Q, R, etc. Terms of λ-calculus are constructed upon variables x, y, z using two rules:

(abstraction) if x is a variable and u a term, then $\lambda x.u$ is a term.

(application) if u and v are terms, then $(u)v$ is a term.

Reduction of λ-calculus is denoted by $\triangleright$.

2 About Typed λ-Calculus and $\neg\neg A = A$

Let us consider usual typed λ-calculus whose types are constructed from atomic types using $\rightarrow$, $\neg$ and $\bot$ ($\bot$ is considered as an atomic type). We denote this system by $S_{\rightarrow,\neg,\bot}$. Judgements are expressions of the form $\Gamma \vdash u : A$, where A is a type, u is a term of λ-calculus and Γ is a context of the form $x_1 : A_1, ..., x_n : A_n$.

The rules of derivation of $S_{\rightarrow,\neg,\bot}$ are the following:

$$x : A \vdash x : A$$

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x.u : A \to B} \qquad \frac{\Gamma_1 \vdash u : A \to B \quad \Gamma_2 \vdash v : A}{\Gamma_1, \Gamma_2 \vdash (u)v : B}$$

$$\frac{\Gamma, x : A \vdash u : \bot}{\Gamma \vdash \lambda x.u : \neg A} \qquad \frac{\Gamma_1 \vdash u : \neg A \quad \Gamma_2 \vdash v : A}{\Gamma_1, \Gamma_2 \vdash (u)v : \bot}$$

We adopt in these rules an implicit management of contraction and weakening. Contraction is obtained through the fact that contexts are considered as sets: in a conclusion of a rule a context Γ_1, Γ_2 denotes the union of the contexts Γ_1 and Γ_2. Weakening is obtained by the convention that in a premise of an introduction rules, $\Gamma, x : A$ denotes a context where $x : A$ doesn't necessary appear.
Note that in $S_{\to,\neg,\bot}$, $\neg A$ is identified with $A \to \bot$. Indeed $\neg$ is often considered as a derived connective whose definition is precisely $\neg A = A \to \bot$.
Suppose now that we add the rule $\neg\neg A \subseteq A$, i.e.

$$\frac{\Gamma \vdash u : \neg\neg A}{\Gamma \vdash u : A}$$

which is equivalent (up to η-equivalence) to the trivial interpretation of the absurdity rule

$$\frac{\Gamma, x : \neg A \vdash u : \bot}{\Gamma \vdash \lambda x.u : A}$$

We call the resulting system $S^*_{\to,\neg,\bot}$. In this system one can prove $x : A \vdash \lambda k.(k)x : A$ as follows

$$\frac{\dfrac{k : \neg A \vdash k : \neg A \qquad x : A \vdash x : A}{k : \neg A, x : A \vdash (k)x : \bot}}{x : A \vdash \lambda k.(k)x : A}$$

The term $\lambda k.(k)x$ will play a fundamental role in the examples of sections 2.1 and 2.2.
We show in the next sections that the system $S^*_{\to,\neg,\bot}$ doesn't satisfy normalisation and correctness properties. This means that the addition of the rule $\neg\neg A \subseteq A$ (and a fortiori the addition of $\neg\neg A = A$) to $S_{\to,\neg,\bot}$ destroys normalisation and correctness properties.

2.1 Normalisation

Proposition 1. *Let $\theta = \lambda f.\lambda x.(\lambda k.(k)f)(f)x$. The term $((\theta)\theta)\theta$ is typable in $S^*_{\to,\neg,\bot}$ and not normalisable.*

Proof. Let C be a type. One defines C^n by induction on n by: $C^1 = C$ and $C^{n+1} = C^n \to C^n$.
One proves that $\vdash \theta : C^{n+2}$, for each $n \geq 1$. We have

$$f : C^{n+1}, x : C^n \vdash (f)x : C^n$$

Because $f : C^{n+1} \vdash \lambda k.(k)f : C^{n+1}$, we have also

$$f : C^{n+1}, x : C^n \vdash (\lambda k.(k)f)(f)x : C^n$$

and thus $\vdash \lambda f.\lambda x.(\lambda k.(k)f)(f)x : C^{n+2}$, i.e $\vdash \theta : C^{n+2}$.
It follows that $\vdash ((\theta)\theta)\theta : C^{n+2}$, for each $n \geq 1$: it suffices to type the first occurence of θ with C^{n+4}, the second with C^{n+3} and the third with C^{n+2}.
Now it is easy to check that $((\theta)\theta)\theta$ is not normalisable because
$\theta = \lambda f.\lambda x.(\lambda k.(k)f)(f)x$ and θ reduces in one step to $\theta^1 = \lambda f.\lambda x.((f)x)f$ and $((\theta^1)\theta^1)\theta^1$ has only one reduction sequence and reduces to itself in two steps as follows:

$$\begin{aligned}((\theta^1)\theta^1)\theta^1 = &\ ((\lambda f.\lambda x.((f)x)f)\theta^1)\theta^1 \\ \triangleright &\ (\lambda x.((\theta^1)x)\theta^1)\theta^1 \\ \triangleright &\ ((\theta^1)\theta^1)\theta^1\end{aligned}$$

2.2 Correctness

In $S^*_{\rightarrow,\neg,\perp}$ types are not preserved by reduction in an essential way, which forbids the derivation of correct programs from proofs. This loss of correctness can be easily shown if one extends typed λ-calculus to a second order typed λ-calculus. Let us take for example the simplest such system, due to Leivant [14] and widely developed in [12,13], which allows to derive correct programs from equational specifications of functions. In such a system one can easely prove that $\lambda x.\lambda y.(x)y$ is a program which computes the exponential y^x. More precisely one has a term e, $\beta\eta$-equivalent to $\lambda x.\lambda y.(x)y$ such that:

$\vdash e : \forall u \forall v(Nu \rightarrow (Nv \rightarrow Nv^u))$

where Nx is the second order type $\forall X(\forall y(Xy \rightarrow Xsy) \rightarrow (X0 \rightarrow Xx))$ saying that x is a natural number.
If one adds $\neg\neg A \subseteq A$, one can prove that $\lambda x.\lambda y.(y)x$ is a also a program which computes the exponential y^x. In other words, the calculus mixed up x^y and y^x! This forbids obviously any hope to derive correct programs in this calculus.

Proof. Suppose $\vdash e : \forall u \forall v(Nu \rightarrow (Nv \rightarrow Nv^u))$. Then
$x : Nu \vdash (e)x : Nv \rightarrow Nv^u$ and $y : \neg(Nv \rightarrow Nv^u), x : Nu \vdash (y)(e)x : \perp$. It follows $x : Nu \vdash \lambda y.(y)(e)x : \neg\neg(Nv \rightarrow Nv^u)$ and because $\neg\neg A \subseteq A$,
$x : Nu \vdash \lambda y.(y)(e)x : Nv \rightarrow Nv^u$.
Therefore $\vdash \lambda x.\lambda y.(y)(e)x : Nu \rightarrow Nv \rightarrow Nv^u$ and
$\vdash \lambda x.\lambda y.(y)(e)x : \forall u \forall v(Nu \rightarrow (Nv \rightarrow Nv^u))$.
This means that $\lambda x.\lambda y.(y)(e)x$ is also a program for y^x. But $\lambda x.\lambda y.(y)(e)x$ is $\beta\eta$-equivalent to $\lambda x.\lambda y.(y)x$:

$$\begin{aligned}\lambda x.\lambda y.(y)(e)x \equiv_{\beta\eta} &\ \lambda x.\lambda y.(y)(\lambda x.\lambda y.(x)y)x \\ \equiv_{\beta\eta} &\ \lambda x.\lambda y.(y)\lambda y.(x)y \\ \equiv_{\beta\eta} &\ \lambda x.\lambda y.(y)x\end{aligned}$$

2.3 Discussion

The problem behind the examples of sections 2.1 and 2.2 appears clearly in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{k : \neg(A \to B) \ \vdash \ k : \neg(A \to B) \quad f : A \to B \ \vdash \ f : A \to B}{k : \neg(A \to B), f : A \to B \ \vdash \ (k)f : \bot}}{f : A \to B \ \vdash \ \lambda k.(k)f : \neg\neg(A \to B)}}{f : A \to B \ \vdash \ \lambda k.(k)f : A \to B} \qquad x : A \ \vdash \ x : A}{f : A \to B, x : A \ \vdash \ (\lambda k.(k)f)x : B}$$

This derivation shows that in $S^*_{\to,\neg,\bot}$, the term $(\lambda k.(k)f)x$ is typable of type B in the context $f : A \to B, x : A$. But $(\lambda k.(k)f)x$ reduces to the term $(x)f$, which is not typable in the context $f : A \to B, x : A$. Therefore typing in $S^*_{\to,\neg,\bot}$ is not preserved under reduction.
This derivation also shows that the addition of the trivial absurdity rule to typed λ-calculus produces the effect of adding the following rule:

$$\dfrac{\Gamma_1 \ \vdash \ u : A \to B \quad \Gamma_2 \ \vdash \ v : A}{\Gamma_1, \Gamma_2 \ \vdash \ (v)u : B}$$

to the usual rule of elimination of implication:

$$\dfrac{\Gamma_1 \ \vdash \ u : A \to B \quad \Gamma_2 \ \vdash \ v : A}{\Gamma_1, \Gamma_2 \ \vdash \ (u)v : B}$$

The effect of choosing an involutive negation is indeed to induce a symetry at the level of application. As the application associated to $\to$ cannot be symmetric, the only possibility to get a safe calculus with an involutive negation is to keep separated the computational interpretations of $\neg$ and $\to$. The obvious way of doing is to choose two different abstractions and two different applications.
As shown below there is one more obstacle to an involutive negation in the context of typed λ-calculus.

2.4 The Role of $\bot$

Suppose now that we restrict our system by forgetting $\to$. The resulting system $S^*_{\neg,\bot}$ has the following rules:

$$x : A \ \vdash \ x : A$$

$$\dfrac{\Gamma\, x : A \ \vdash \ u : \bot}{\Gamma \ \vdash \ \lambda x.u : \neg A} \qquad \dfrac{\Gamma_1 \ \vdash \ u : \neg A \quad \Gamma_2 \ \vdash \ v : A}{\Gamma_1, \Gamma_2 \ \vdash \ (u)v : \bot}$$

and in addition the trivial absurdity rule:

$$\dfrac{\Gamma, x \ : \ \neg A \vdash u \ : \bot}{\Gamma \vdash \lambda x.u \ : \ A}$$

We show that normalisation fails for $S^*_{\neg,\bot}$.

Proposition 2. *Let* $\xi = \lambda x.\lambda f.(\lambda k.(k)f)(f)x$ *and* y *a variable. The term* $(\lambda k.(k)\xi)(\xi)y$ *is typable in* $S^*_{\neg,\bot}$ *and not normalisable.*

Proof. We first show that $(\lambda k.(k)\xi)(\xi)y$ is typable in $S^*_{\neg,\bot}$.

We have $x : \bot, f : \neg\bot \vdash (f)x : \bot$ and $f : \neg\bot \vdash \lambda k.(k)f : \neg\bot$. Therefore $x : \bot\,, f : \neg\bot \vdash (\lambda k.(k)f)(f)x : \bot$ and $x : \bot \vdash \lambda f.(\lambda k.(k)f)(f)x : \neg\neg\bot$. Because $\neg\neg\bot \subseteq \bot$, we also $\vdash \lambda x.\lambda f.(\lambda k.(k)f)(f)x : \neg\bot$ i.e. $\vdash \xi : \neg\bot$. It follows $\vdash \lambda k.(k)\xi : \neg\bot$ and $y : \bot \vdash (\lambda k.(k)\xi)(\xi)y : \bot$.

Let $\xi' = \lambda x.\lambda f.((f)x)f$. The term $(\lambda k.(k)\xi)(\xi)y$ reduces to the term $((\xi')y)\xi'$ which has only one reduction sequence and reduces in two steps to itself as follows:

$$\begin{aligned} ((\xi')y)\xi' = {} & ((\lambda x.\lambda f.((f)x)f)y)\xi' \\ \triangleright {} & (\lambda f.((f)y)f)\xi' \\ \triangleright {} & ((\xi')y)\xi' \end{aligned}$$

In order to type a non normalisable term in the system $S^*_{\neg,\bot}$ we have made an essential use of the fact that $\bot$ is an atomic type of the system, which can be used to built other type (we used the type $\neg\bot$). The problem lies in the confusion between two uses of $\bot$: as indicating a contradiction in a proof and as an atomic type. Therefore in order to get normalising calculus with an involutive negation we have to forbid $\bot$ as an atomic type (it can be a "special" type, which is outside the system).

Note that the two obstacles to an involutive computational interpretation of negation are completely different. In particular, the examples of sections 2.1 and 2.2 do not use the fact that $\bot$ is an atomic type of the system: they hold for the system $S^*_{\to,\neg}$, where $\bot$ is used only for indicating a contradiction in a proof.

3 Typed $\lambda\mu$-Calculus with $\neg\neg A = A$

In this section, we extend typed $\lambda\mu$-calculus in a natural way with an involutive negation: the abstraction and application associated to negation are simply the operator μ and $[\,]$ from $\lambda\mu$-calculus.

3.1 $\lambda\mu$-Calculus

Typed $\lambda\mu$-calculus is a simple computational interpretation of classical logic introduced in [17]. It has both a clear interpretation in terms of environment machines and a clear semantics in terms of continuations [10,11,20].

The $\lambda\mu$-calculus has two kinds of variables: the λ-variables $x, y, z, ...$, and the μ-variables $\alpha, \beta, \gamma, ...$ Terms are defined inductively as follows:

- x is a term, for x a λ-variable;
- $\lambda x.u$ is a term, for x a λ-variable and u a term;
- $(t)u$ is a term, for t and u terms;
- $\mu\alpha.[\beta]t$ is a term, for t a term and α, β μ-variables.

Expressions of the form $[\beta]t$, where β is μ-variables and t a term, are called named terms. They correspond to type $\bot$ in typed $\lambda\mu$-calculus.

Typed $\lambda\mu$-calculus is a calculus for classical logic, enjoying confluence and strong normalisation [17,18], which doesn't make use of negation. Types are build from atomic types using $\rightarrow$ only. Type $\bot$ is not needed, but is added for convenience as a special type denoting a contradiction in a proof (in the context of typed $\lambda\mu$-calculus one could also consider it as an atomic type). Judgments have two contexts: one to the left for λ-variables and one to the right for μ-variables. In order to make the symmetric extension easier to understand, we adopt here a presentation where the right context is replaced by a negated left context. Of course, this doesn't change the calculus; in particular negation is not needed inside types.
Judgments are expressions of the form $\Gamma;\ \Delta\ \vdash\ u : A$, where A is a type, u is a term of $\lambda\mu$-calculus, Γ is a context of the form $x_1 : A_1, ..., x_n : A_n$ and Δ a context of the form $\alpha_1 : \neg A_1, ..., \alpha_n : \neg A_n$.
The typing rules of $\lambda\mu$-calculus are the following:

$$x : A\ \vdash\ x : A$$

$$\frac{\Gamma,\ x : A;\ \Delta\ \vdash\ u : B}{\Gamma;\ \Delta\ \vdash\ \lambda x.u : A \rightarrow B} \qquad \frac{\Gamma_1;\ \Delta_1\ \vdash\ u : A \rightarrow B \quad \Gamma_2;\ \Delta_2\ \vdash\ v : A}{\Gamma_1, \Gamma_2;\ \Delta_1, \Delta_2\ \vdash\ (u)v : B}$$

$$\frac{\Gamma;\ \Delta,\ \alpha : \neg A\ \vdash\ u : \bot}{\Gamma;\ \Delta\ \vdash\ \mu\alpha.u : A} \qquad \frac{\Gamma;\ \Delta\ \vdash\ u : A}{\Gamma;\ \Delta,\ \alpha : \neg A\ \vdash\ [\alpha]u : \bot}$$

As for typed λ-calculus we adopt in these rules an implicit management of contraction and weakening, with the same conventions as in section 2.
The $\lambda\mu$-calculus has two fundamental reduction rules:

(R_1) $\quad (\lambda x.u)v\ \triangleright\ u[v/x]$

(R_2) $\quad (\mu\alpha.u)v\ \triangleright\ \mu\alpha'.u[[\alpha'](w)v/[\alpha]w]$

and in addition simplification rules (like η-rule of λ-calculus):

(S_1) $\quad \lambda x.(u)x\ \triangleright\ u$

(S_2) $\quad \mu\alpha.[\alpha]u\ \triangleright\ u$

Simplification rules are subject to the following restrictions: in (S_1), x has no free occurences in u; in (S_2), α has no free occurences in u.
In (R_2), the term $u[[\alpha'](w)v/[\alpha]w]$ is defined as the result of substituting to each subterm of u of the form $[\alpha]w$, the new subterm $[\alpha'](w)v$. Note that if α has type $\neg(A \rightarrow B)$, then α' has type $\neg B$.

3.2 Symmetric $\lambda\mu$-Calculus

We introduce an extension of $\lambda\mu$-calculus with an involutive negation, called symmetric $\lambda\mu$-calculus. We simply take the abstraction μ and the application $[\,]$ of $\lambda\mu$-calculus as beeing the new abstraction and application corresponding to the computational content of this involutive negation.

Types of symmetric $\lambda\mu$-calculus are defined as follows:

$$A := P \mid \neg P \mid A \to B \mid \neg(A \to B)$$

where P denotes atomic types.
The negation $\neg$ is extended to an involutive negation on types in the obvious way.
For convenience, one adds a special type $\bot$. For the reason explained in section 2.4 , $\bot$ doesn't belong to the set of atomic types.
Note that an involutive negation, invites to confuse the rule of introduction of negation

$$\frac{\Gamma,\ x : A \ \vdash\ u : \bot}{\Gamma \ \vdash\ \mu x.u : \neg A}$$

and the absurdity rule

$$\frac{\Gamma,\ x : \neg A \ \vdash\ u : \bot}{\Gamma \ \vdash\ \mu x.u : A}$$

and also to have only one kind of variable.
This is this drastic solution that we adopt with symmetric $\lambda\mu$-calculus, because it should better capture the essence of an involutive negation, but more permissive ones might also be interesting at the computational level.

Terms of Symmetric $\lambda\mu$-Calculus.

Symmetric $\lambda\mu$-calculus has only one kind of variables. Terms are defined inductively as follows:

- x is a term, for x a λ-variable;
- $\lambda x.u$ is a term, for x a variable and u a term;
- $(t)u$ is a term, for t and u terms;
- $\mu x.[u]v$ is a term, for x a variable and u, v terms.

Expressions of the form $[u]v$, where u, v are terms, are called named terms. They correspond to type $\bot$ in typed symmetric $\lambda\mu$-calculus.

Typing Rules of Symmetric $\lambda\mu$-Calculus.

$$x : A \ \vdash\ x : A$$

$$\frac{\Gamma,\ x : A \ \vdash\ u : B}{\Gamma \ \vdash\ \lambda x.u : A \to B} \qquad \frac{\Gamma_1 \ \vdash\ u : A \to B \quad \Gamma_2 \ \vdash\ v : A}{\Gamma_1, \Gamma_2 \ \vdash\ (u)v : B}$$

$$\frac{\Gamma, x : A \ \vdash\ u : \bot}{\Gamma \ \vdash\ \mu x.u : \neg A} \qquad \frac{\Gamma_1 \ \vdash\ u : \neg A \quad \Gamma_2 \ \vdash\ v : A}{\Gamma_1, \Gamma_2 \ \vdash\ [u]v : \bot}$$

As for typed λ-calculus one adopts in these rules an implicit management of contraction and weakening, with the same conventions as in section 2.

Reduction Rules of Symmetric $\lambda\mu$-calculus.

The symmetric $\lambda\mu$-calculus has the following reduction rules:

(R_1) $(\lambda x.u)v \triangleright u[v/x]$

(R_2) $(\mu x.u)v \triangleright \mu x'.u[\mu z.[x'](z)v/x]$

(R_3) $[\mu x.u]v \triangleright u[v/x]$

(R_4) $[u]\mu x.v \triangleright v[u/x]$

and in addition simplification rules (like η-rule of λ-calculus):

(S_1) $\lambda x.(u)x \triangleright u$

(S_2) $\mu x.[x]u \triangleright u$

(S_3) $\mu x.[u]x \triangleright u$

Simplification rules are subject to the following restriction: in (S_1), (S_2), (S_3), x has no free occurences in u.

Symmetric $\lambda\mu$-calculus is clearly an extension of $\lambda\mu$-calculus. Rule (R1) is the usual β-reduction of λ-calculus. Because we make a more liberal use of variables in symmetric $\lambda\mu$-calculus, the rule (R2) of μ-reduction is stated in a more general setting than the corresponding rule of $\lambda\mu$-calculus, but his effect is exactly the same when restricted to terms of $\lambda\mu$-calculus.

In $\lambda\mu$-calculus the substituted variable x (which is a μ-variable) always occurs in a subterm $[x]w$ and the result of the substitution is in this case $[\mu z.[x'](z)v]w$ which reduces to $[x'](w)v$. This corresponds exactly to the substitution of the rule (R2) of $\lambda\mu$-calculus.

The reduction rule (R2) of symmetric $\lambda\mu$-calculus is simpler to understand with typed terms. Suppose that one reduces the term $(\mu x^{\neg(A\to B)}.u^{\perp})v^A$. One replaces in u the occurences of $x^{\neg(A\to B)}$ by a canonical term of type $\neg(A \to B)$ which is $\mu z^{A\to B}.[x'^{\neg B}](z^{A\to B})v^A$. This term can be thought as a pair $\langle v^A, x'^{\neg B}\rangle$.

The two new rules are the rules (R3) and (R4), which correspond respectively to a kind β-reduction for μ and its symmetric, are exactly those of the symmetric λ-calculus of Barbanera and Berardi [2].

Note that the rule (R2) introduces a "communication" between $\to$ and $\neg$, which has no real equivalent in the symmetric λ-calculus.

Because of its symmetric nature (appearing in rules (R3) and (R4)), symmetric $\lambda\mu$-calculus is essentially not confluent. As in symmetric λ-calculus, this non confluence could be used in positive way to derive symmetric programs.

Indeed symmetric $\lambda\mu$-calculus contains two different ways of computing with classical proofs: the one of $\lambda\mu$-calculus, based on the specific rule (R2) of μ-reduction, which is well understood in terms of machines and continuations; the one of symmetric λ-calculus, based on rules (R3) and (R4), which is of a different computational nature. The embedding of $\lambda\mu$-calculus in symmetric $\lambda\mu$-calculus is obvious and doesn't make use of (R3) and (R4). In the next section we develop

an embedding of symmetric λ-calculus in symmetric $\lambda\mu$-calculus, which doesn't make use of (R2).

4 Interpretation of Symmetric λ-Calculus in Symmetric $\lambda\mu$-Calculus

In this section, we give a translation of symmetric λ-calculus in symmetric $\lambda\mu$-calculus, which doesn't make use of rule (R2): reduction involves only the rules (R1), (R3) and (R4).

4.1 The Symmetric λ-Calculus of Barbanera and Berardi

The types of the system are defined by:

$$A := P|\neg P|A \wedge B|A \vee B$$

where P denotes atomic types.
An involutive negation on types is defined as follows:

$\neg(A) = \neg A$
$\neg(\neg A) = A$
$\neg(A \wedge B) = \neg A \vee \neg B$
$\neg(A \vee B) = \neg A \wedge \neg B$

There is also a special type $\perp$, which doesn't belong to the set of atomic types.

Derivation Rules of Symmetric λ-Calculus

$$x : A \vdash x : A$$

$$\frac{\Gamma \vdash u : A \qquad \Delta \vdash v : B}{\Gamma, \Delta \vdash \langle u, v\rangle : A \wedge B} \ \wedge\text{-}intro \qquad \frac{\Gamma \vdash u_i : A_i}{\Gamma \vdash \sigma_i(u_i) : A_1 \vee A_2} \ \vee\text{-}intro \ (i{=}1{,}2)$$

$$\frac{\Gamma, x : A \vdash u : \perp}{\Gamma \vdash \lambda x.u : \neg A} \ \neg\text{-}intro \qquad \frac{\Gamma \vdash u : \neg A \qquad \Delta \vdash v : A}{\Gamma, \Delta \vdash u * v : \perp} \ \neg\text{-}elim$$

As for typed λ-calculus one adopts in these rules an implicit management of contraction and weakening, with the same conventions as in section 2.

Reduction Rules of Symmetric λ-Calculus

(β) $\quad \lambda x.u * v \ \triangleright \ u[v/x]$
$(\beta^{\perp})$ $\quad u * \lambda x.v \ \triangleright \ v[u/x]$
(π) $\quad \langle u_1, u_2\rangle * \sigma_i(v_i) \ \triangleright \ u_i * v_i$
$(\pi^{\perp})$ $\quad \sigma_i(v_i) * \langle u_1, u_2\rangle \ \triangleright \ v_i * u_i$

Symmetric λ-calculus is obviously not confluent but enjoys strong normalisation [2,3]. Moreover its non-confluence can be used in a positive way to derive symmetric programs [4].

4.2 Interpretation of the Symmetric λ-Calculus in the Symmetric $\lambda\mu$-Calculus

Connectives $\wedge$ and $\vee$ are translated as follows:
$A \vee B = \neg A \to B$
$A \wedge B = \neg(A \to \neg B)$
The rules $\wedge$-intro, $\vee$-intro, $\neg$-intro and $\neg$-elim are translated as follows:
$\wedge$-intro

$$\dfrac{\dfrac{z : A \to \neg B \vdash z : A \to \neg B \qquad \Gamma_1 \vdash u : A}{\Gamma_1, z : A \to \neg B \vdash (z)u : \neg B} \qquad \Gamma_2 \vdash v : B}{\dfrac{\Gamma_1, \Gamma_2, z : A \to \neg B \vdash [(z)u]v : \bot}{\Gamma_1, \Gamma_2 \vdash \mu z.[(z)u]v : \neg(A \to \neg B)}}$$

$\vee$-intro$_1$

$$\dfrac{\dfrac{\dfrac{z : \neg A \vdash z : \neg A \qquad \Gamma \vdash u : A}{\Gamma, z : \neg A \vdash [z]u : \bot}}{\Gamma, z : \neg A \vdash \mu d.[z]u : B}}{\Gamma \vdash \lambda z.\mu d.[z]u : \neg A \to B}$$

$\vee$-intro$_2$

$$\dfrac{\dfrac{\dfrac{z : \neg B \vdash z : \neg B \qquad \Gamma \vdash u : B}{\Gamma, z : \neg B \vdash [z]u : \bot}}{\Gamma \vdash \mu z.[z]u : B}}{\Gamma \vdash \lambda d.\mu z.[z]u : \neg A \to B}$$

$\neg$-intro

$$\dfrac{\Gamma, x : A \vdash u : \bot}{\Gamma \vdash \mu x.u : \neg A}$$

$\neg$-elim

$$\dfrac{\Gamma_1 \vdash u : \neg A \qquad \Gamma_2 \vdash v : A}{\Gamma_1, \Gamma_2 \vdash [u]v : \bot}$$

Let T be the translation defined inductively as follows:
$T(x) = x$
$T(\langle u, v\rangle) = \mu z[(z)T(u)]T(v)$
$T(\sigma_1(u) = \lambda z.\mu d.[z]T(u)$
$T(\sigma_2(u) = \lambda d.\mu z.[z]T(u)$
$T(\lambda x.u) = \mu x.T(u)$
$T(u * v) = [T(u)]T(v)$

As shown before, T preserves types and it is easy to check that T is compatible with substitution, i.e. $T(u[v/x]) = T(u)[T(v)/x]$. We prove now that T preserves reduction, in a way which doesn't make use of rule (R2).

$$\begin{aligned}
T(\{\lambda x.u\} * v) &= [\mu x.T(u)]T(v) \\
&\triangleright\ T(u)[T(v)/x] \\
&= T(u[v/x]) \\
T(\langle u_1, u_2\rangle *\ \sigma_1(v)) &= [\mu z.[(z)T(u_1)]T(u_2)]\lambda z.\mu d.[z]T(v) \\
&\triangleright\ [(\lambda z.\mu d.[z]T(v))T(u_1)]T(u_2) \\
&\triangleright\ [\mu d.[T(u_1)]T(v)]T(u_2) \\
&\triangleright\ [T(u_1)]T(v) \\
&= T(u_1 * v) \\
T(\langle u_1, u_2\rangle *\ \sigma_2(v)) &= [\mu z.[(z)T(u_1)]T(u_2)]\lambda d.\mu z.[z]T(v) \\
&\triangleright\ [(\lambda d.\mu z.[z]T(v))T(u_1)]T(u_2) \\
&\triangleright\ [\mu z.[z]T(v)]T(u_2) \\
&\triangleright\ [T(u_2)]T(v) \\
&= T(u_2 * v)
\end{aligned}$$

The symmetric rules are preserved in the same way.

References

1. H. Barendregt : The Lambda-Calculus. North-Holland, 1981.
2. F. Barbanera, S. Berardi : A symmetric lambda-calculus for classical program extraction. Proceedings TACS'94, Springer LNCS **789** (1994).
3. F. Barbanera, S. Berardi : A symmetric lambda-calculus for classical program extraction. Information and Computation **125** (1996) 103-117.
4. F. Barbanera, S. Berardi, M. Schivalocchi : "Classical" programming-with-proofs in lambda-sym: an analysis of a non-confluence. Proc. TACS'97.
5. M. Felleisen, D.P. Friedman, E. Kohlbecker, B. Duba : A syntactic theory of sequential control. Theoretical Computer Science **52** (1987) pp 205-237.
6. M. Felleisen, R. Hieb : The revised report on the syntactic theory of sequential control and state. Theoretical Computer Science **102** (1994) 235-271.
7. J.Y. Girard : Linear logic. Theoretical Computer Science. **50** (1987) 1-102.
8. J.Y. Girard, Y. Lafont, and P. Taylor : Proofs and Types. Cambridge University Press, 1989.
9. T. Griffin : A formulae-as-types notion of control. Proc. POPL'90 (1990) 47-58.
10. M. Hofmann, T. Streicher : Continuation models are universal for $\lambda\mu$-calculus. Proc. LICS'97 (1997) 387-397.
11. M. Hofmann, T. Streicher : Completeness of continuation models for $\lambda\mu$-calculus. Information and Computation (to appear).
12. J.L. Krivine, M. Parigot: Programming with proofs. J. of Information Processing and Cybernetics **26** (1990) 149-168.
13. J.L. Krivine : Lambda-calcul, types et modèles. Masson, 1990.
14. D. Leivant : Reasoning about functional programs and complexity classes associated with type disciplines. Proc. FOCS'83 (1983) 460-469.
15. C. Murthy : Extracting Constructive Content from Classical Proofs. PhD Thesis, Cornell, 1990.

16. M. Parigot : Free Deduction: an Analysis of "Computations" in Classical Logic. Proc. Russian Conference on Logic Programming, 1991, Springer LNCS **592** 361-380.
17. M. Parigot : $\lambda\mu$-calculus: an Algorithmic Interpretation of Classical Natural Deduction. Proc. LPAR'92, Springer LNCS **624** (1992) 190-201.
18. M. Parigot : Strong normalisation for second order classical natural deduction, Proc. LICS'93 (1993) 39-46.
19. C.H.L. Ong, C.A. Stewart : A Curry-Howard foundation for functional computation with control. Proc. POPL'97 (1997)
20. P. Selinger : Control categories and duality: on the categorical semantics of lambda-mu calculus, Mathematical Structures in Computer Science (to appear).