The Calculus of Explicit Substitutions $\lambda\upsilon$

Pierre LESCANNE and Jocelyne ROUYER-DEGLI
Centre de Recherche en Informatique de Nancy (CNRS)
and INRIA-Lorraine
Campus Scientifique, BP 239,
F54506 Vandœuvre-lès-Nancy, France

email: Pierre.Lescanne@loria.fr, Jocelyne.Rouyer@loria.fr

April 26, 1995

Abstract

$\lambda\upsilon$-calculus is a new simple calculus of explicit substitutions. In this paper we explore its properties, namely we prove that it correctly implements β reduction, it is confluent, its simply typed version is strongly normalizing. We associate with $\lambda\upsilon$ an abstract machine called the U-machine and we prove that the U-machine correctly implements $\lambda\upsilon$..

1 The $\lambda\upsilon$-calculus

The main mechanism of λ-calculus is β-conversion which is usually defined as $(\lambda x.a)b \to a\{b/x\}$, where $\{b/x\}$ is the substitution of the term b to the variable x. In classical λ-calculus [2] the mechanism of substitution is usually described at a meta-level by a specific and external formalism unlike λ-calculi of explicit substitutions which contain in a same framework both the β-rule and a description of the evaluation of the substitution. λ-calculi of explicit substitutions are first order term rewrite systems and were introduced by de Bruijn in 1978 [9]. Such calculi allow nice and uniform descriptions of implementations of λ-calculus. Since several of them have been proposed [1, 10, 14, 7, 18, 12, 16, 17, 20]. $\lambda\upsilon$ is a new calculus of explicit substitutions very simple if compared with others. The main property of $\lambda\upsilon$ (read *lambda-upsilon*) is that its set of operators is minimal in the sense that it contains only operators that are necessary to describe the calculus. There are four operators on terms namely *abstraction*, *application*, *closure* and *variables*. The three operators on substitutions *slash*, *lift* and *shift* are introduced by need. The operator *closure* $\_[\_]$ introduces *substitutions* into the calculus. $\lambda\upsilon$ uses de Bruijn's indices [8] and we write variables $\underline{1}$, $\underline{2}$,..., $\underline{n}$, $\underline{n+1}$, ... Notice that underlining a natural number is making a variable from a natural number. "Underline" is actually the operator which was written $V(n)$ in [16]. It has an interpretation in Figure 2. A term that does not contain closures is called a pure term. In $\lambda\upsilon$ the β-rule is replaced by a more elementary rule namely

$$(B) \quad (\lambda a)b \to a[b/]$$

where $b/$ is the substitution with the intuitive meaning:

$$\begin{array}{llll} b/: & \underline{1} & \mapsto & b \\ & \underline{2} & \mapsto & \underline{1} \\ & & \vdots & \\ & \underline{n+1} & \mapsto & \underline{n} \\ & & \vdots & \end{array}$$

Other rules are given to get rid of substitutions, these rules will form the calculus v. λv is the calculus $(B) \cup v$. The first rule *App* distributes substitution into an application (ab).

$$(App) \quad (ab)[s] \rightarrow a[s]b[s].$$

When a substitution goes under a λ it has to be modified. A new operator is introduced. It is called *Lift* and written $\Uparrow$. It performs this modification on the substitution and a rule of λv called $(Lambda)$ introduces this operator.

$$(Lambda) \quad (\lambda a)[s] \rightarrow \lambda(a[\Uparrow(s)]).$$

Lift has the following intuitive meaning:

$$\begin{array}{rlcl} \Uparrow(s): & \underline{1} & \mapsto & \underline{1} \\ & \underline{2} & \mapsto & s(\underline{1})[\uparrow] \\ & & \vdots & \\ & \underline{n+1} & \mapsto & s(\underline{n})[\uparrow] \\ & & \vdots & \end{array}$$

$\uparrow$ is a specific substitution that just *shifts* the variables in a term.

$$\begin{array}{rlcl} \uparrow: & \underline{1} & \mapsto & \underline{2} \\ & \underline{2} & \mapsto & \underline{3} \\ & & \vdots & \\ & \underline{n} & \mapsto & \underline{n+1} \\ & & \vdots & \end{array}$$

The meaning of *Lambda* can be explained as follows. In the expression $(\lambda a)[s]$, s does not affects the $\underline{1}$'s which occur in a. Similarly, in the expression $\lambda(a[\Uparrow(s)])$, $\Uparrow(s)$ should not affect the $\underline{1}$'s which occur in a. On the other hand when $[\Uparrow(s)]$ is applied to other variables, it has to take into account that variables under λ have been renamed and to reset the name of the variables in $s(\underline{n})$ accordingly. This is done by $\uparrow$. Notice that in λv there is no need for closure rules. Indeed, in a term of the form $a[s][t]$ it is not necessary to tell how t acts on $a[s]$ since by induction one gets rid of s. Now to specify completely the behavior of substitutions one has just to describe by rewrite rules their action on variables. Putting together all these ideas, we get the rewrite rules of Figure 1. Notice that the system is essentially *lazy*, in the sense that the evaluation of the substitution $a[b/]$ created by $(\lambda a)b$ can be delayed. The rewrite system v terminates. The proof is easy and can be done with elementary interpretations (functions made of polynomials and exponentials) [16, 15]. It is given in Figure 2. v is also an *orthogonal* rewrite system, which means that it is left-linear and without superposition. This property is very important both for implementation and proofs, for instance Luc Maranget (private communication) used it to prove termination by structural induction. λv has three sorts of objects, namely

$$\begin{array}{llcl} \textbf{Terms} & a & ::= & \underline{n} \mid ab \mid \lambda a \mid a[s] \\ \textbf{Substitutions} & s & ::= & a/ \mid \Uparrow(s) \mid \uparrow \\ \textbf{Naturals} & n & ::= & n+1 \mid 1. \end{array}$$

λv does not introduce composition of substitutions. This makes the system simpler. Indeed for presenting a calculus of explicit substitutions, such a composition is useless, at least at the logical level and its introduction in other calculi seems dictated by "efficiency". If new rules dealing with composition need to be introduced, they should be first proved correct as induction theorems and then added to the system. See [16] for a discussion on the way to mechanize the

(B)	$(\lambda a)b$	$\to$	$a[b/]$
(App)	$(ab)[s]$	$\to$	$a[s]b[s]$
(Lambda)	$(\lambda a)[s]$	$\to$	$\lambda(a[\Uparrow(s)])$
(FVar)	$\underline{1}[a/]$	$\to$	a
(RVar)	$\underline{n+1}[a/]$	$\to$	$\underline{n}$
(FVarLift)	$\underline{1}[\Uparrow(s)]$	$\to$	$\underline{1}$
(RVarLift)	$\underline{n+1}[\Uparrow(s)]$	$\to$	$\underline{n}[s][\uparrow]$
(VarShift)	$\underline{n}[\uparrow]$	$\to$	$\underline{n+1}$

Figure 1: The rewrite system $\lambda\upsilon$

$$
\begin{array}{rclcrcl}
[\![\underline{n}]\!]_1 & = & 2^{[\![n]\!]_1} & & [\![\underline{n}]\!]_2 & = & 2^{[\![n]\!]_2} \\
[\![n+1]\!]_1 & = & [\![n]\!]_1 + 1 & & [\![n+1]\!]_2 & = & [\![n]\!]_2 + 1 \\
[\![1]\!]_1 & = & 2 & & & & \\
[\![ab]\!]_1 & = & [\![a]\!]_1 + [\![b]\!]_1 + 1 & & & & \\
[\![\lambda a]\!]_1 & = & [\![a]\!]_1 + 1 & & & & \\
[\![a[s]\,]\!]_1 & = & [\![a]\!]_1 [\![s]\!]_1 & & [\![a[s]\,]\!]_2 & = & [\![a]\!]_2 [\![s]\!]_2 \\
[\![\Uparrow(s)]\!]_1 & = & [\![s]\!]_1 & & [\![\Uparrow(s)]\!]_2 & = & 2[\![s]\!]_2 \\
[\![\uparrow]\!]_1 & = & 2 & & [\![\uparrow]\!]_2 & = & 3 \\
[\![a/]\!]_1 & = & any & & & &
\end{array}
$$

Figure 2: Interpretations for proving the termination of υ

introduction of composition and a comparison with other approachs. Among other systems of explicit substitutions, [16] introduces $\lambda\upsilon$ but does not prove any of its properties.

This paper is structured as follows. In Section 2, we prove that $\lambda\upsilon$ correctly implements β-reduction. In Section 3, we prove the confluence of $\lambda\upsilon$. In Section 4, we show how $\lambda\upsilon$ can be simply typed. In Section 5, we prove that simply typed $\lambda\upsilon$ terms are strongly normalizing. In Section 6, we present the U-machine, an environment machine that intends to implement $\lambda\upsilon$ and finally in Section 7, we prove that the U-machine is actually a correct implementation of $\lambda\upsilon$.

2 Correction of the β-reduction in $\lambda\upsilon$

We write $\upsilon(a)$ the normal form of the term a w.r.t. υ. β is the classical β-reduction of λ-calculus. It is the relation $a \underset{\beta}{\longrightarrow} b$ between pure terms where $a \underset{B}{\longrightarrow} b'$ and $b = \upsilon(b')$, we also write $b' \overset{!}{\underset{\upsilon}{\longrightarrow}} b$. This definition is correct. Indeed, let us introduce an external definition of substitution σ_0. The classical definition of β-reduction in terms of this operation (with definitions from [12]), is

$$(\lambda a)b \underset{\beta}{\longrightarrow} \sigma_0(a, b)$$

where σ_0 is the instance in 0 of a function σ_n defined as follows.

$$\begin{array}{c} \sigma_n(ac, b) = \sigma_n(a, b)\sigma_n(c, b) \\ \sigma_n(\lambda a, b) = \lambda(\sigma_{n+1}(a, b)) \end{array} \quad \sigma_n(\underline{m}, b) = \left\{ \begin{array}{rl} \underline{m-1} & \text{if } m > n+1 \\ \tau_0^n(b) & \text{if } m = n+1 \\ \underline{m} & \text{if } m \leq n \end{array} \right.$$

where:

$$\begin{array}{c} \tau_i^n(ab) = \tau_i^n(a)\tau_i^n(b) \\ \tau_i^n(\lambda a) = \lambda(\tau_{i+1}^n(a)) \end{array} \quad \tau_i^n(\underline{m}) = \left\{ \begin{array}{rl} \underline{m+n} & \text{if } m > i \\ \underline{m} & \text{if } m \leq i \end{array} \right.$$

Notice that $\tau_i^n \circ \tau_i^m = \tau_i^{n+m}$ and $\tau_i^0(a) = a$. We define a translation μ that links impure terms with σ_n and τ_n^i.

$$\begin{array}{rcl} \mu(a[\Uparrow^n(b/)]) & = & \sigma_n(\mu(a), \mu(b)) \\ \mu(a[\Uparrow^n(\uparrow)]) & = & \tau_n^1(\mu(a)) \\ \mu(\underline{n}) & = & \underline{n} \\ \mu(ab) & = & \mu(a)\mu(b) \\ \mu(\lambda a) & = & \lambda(\mu(a)) \end{array}$$

Notice that if a is a pure term, then $\mu(a) = a$, in particular, $\mu(\upsilon(a)) = \upsilon(a)$. The following proposition shows that both definitions coincide.

Proposition 1

1. *$a \underset{\upsilon}{\longrightarrow} b \Rightarrow \mu(a) = \mu(b)$, hence $\upsilon(a) = \mu(a)$.*
2. *$\upsilon(a[b/]) = \sigma_0(\mu(a), \mu(b))$.*

Proof: In order to prove the first assertion we consider each rule of υ.

- $(ab)[s] \underset{\upsilon}{\longrightarrow} a[s]b[s]$
  - case $s = \Uparrow^n(c/)$

$$\begin{array}{rcl} \mu((ab)[s]) & = & \sigma_n(\mu(ab), \mu(c)) = \sigma_n(\mu(a)\mu(b), \mu(c)) \\ & = & \sigma_n(\mu(a), \mu(c))\ \sigma_n(\mu(b), \mu(c)). \\ \mu(a[s]b[s]) & = & \mu(a[s])\mu(b[s]) \\ & = & \sigma_n(\mu(a), \mu(c))\ \sigma_n(\mu(b), \mu(c)). \end{array}$$

- case $s = \Uparrow^n(\uparrow)$
  $\mu((ab)[s]) = \tau_n^1(\mu(ab)) = \tau_n^1(\mu(a)\ \mu(b)) = \tau_n^1(\mu(a))\ \tau_n^1(\mu(b))$.
  $\mu(a[s]b[s]) = \mu(a[s])\ \mu(b[s]) = \tau_n^1(\mu(a))\ \tau_n^1(\mu(b))$.

- $(\lambda a)[s] \underset{v}{\longrightarrow} \lambda(a[\Uparrow(s)])$
  - case $s = \Uparrow^n(b/)$
    $\mu((\lambda a)[s]) = \sigma_n(\mu(\lambda a), \mu(b)) = \sigma_n(\lambda\mu(a), \mu(b)) = \lambda\sigma_{n+1}(\mu(a), \mu(b))$.
    $\mu(\lambda(a[\Uparrow(s)])) = \lambda\mu(a[\Uparrow(s)]) = \lambda\sigma_{n+1}(\mu(a), \mu(b))$.
  - case $s = \Uparrow^n(\uparrow)$
    $\mu((\lambda a)[s]) = \tau_n^1(\mu(\lambda a)) = \tau_n^1(\lambda\mu(a)) = \lambda\tau_{n+1}^1(a)$.
    $\mu(\lambda(a[\Uparrow(s)])) = \lambda\mu(a[\Uparrow(s)]) = \lambda\tau_{n+1}^1(a)$.
- $\underline{1}[a/] \underset{v}{\longrightarrow} a$, $\mu(\underline{1}[a/]) = \sigma_0(\mu(\underline{1}), \mu(a)) = \sigma_0(\underline{1}, \mu(a)) = \tau_0^0(\mu(a)) = \mu(a)$.
- $\underline{n+1}[a/] \underset{v}{\longrightarrow} \underline{n}$
  $\mu(\underline{n+1}[a/]) = \sigma_0(\mu(\underline{n+1}), \mu(a)) = \sigma_0(\underline{n+1}, \mu(a)) = \underline{n} = \mu(\underline{n})$.
- $\underline{1}[\Uparrow(s)] \underset{v}{\longrightarrow} \underline{1}$
  - case $s = \Uparrow^n(b/)$
    $\mu(\underline{1}[\Uparrow(s)]) = \sigma_{n+1}(\mu(\underline{1}), \mu(b)) = \sigma_{n+1}(\underline{1}, \mu(b)) = \underline{1} = \mu(\underline{1})$
  - case $s = \Uparrow^n(\uparrow)$
    $\mu(\underline{1}[\Uparrow(s)]) = \tau_{n+1}^1(\mu(\underline{1})) = \tau_{n+1}^1(\underline{1}) = \underline{1} = \mu(\underline{1})$.
- $\underline{n+1}[\Uparrow(s)] \underset{v}{\longrightarrow} \underline{n}[s][\uparrow]$
  - case $s = \Uparrow^k(b/)$
    $\mu(\underline{n+1}[\Uparrow(s)]) = \sigma_{k+1}(\mu(\underline{n+1}), \mu(b)) = \sigma_{k+1}(\underline{n+1}, \mu(b))$. By case,
    * $\sigma_{k+1}(\underline{n+1}, \mu(b)) = \underline{n+1}$ if $n \leq k$,
    * $\sigma_{k+1}(\underline{n+1}, \mu(b)) = \underline{n}$ if $n > k+1$
    * and $\sigma_{k+1}(\underline{n+1}, \mu(b)) = \tau_0^{k+1}(\mu(b))$ if $n = k+1$.

    $\mu(\underline{n}[s][\uparrow]) = \tau_0^1(\mu(\underline{n}[\Uparrow^k\ (b/)])) = \tau_0^1(\sigma_k(\mu(\underline{n}), \mu(b))) = \tau_0^1(\sigma_k(\underline{n}, \mu(b)))$. By case,
    * $\tau_0^1(\sigma_k(\mu(\underline{n}), \mu(b))) = \tau_0^1(\sigma_k(\underline{n}, \mu(b))) = \tau_0^1(\underline{n}) = \underline{n+1}$ if $n \leq k$,
    * $\tau_0^1(\sigma_k(\mu(\underline{n}), \mu(b))) = \tau_0^1(\sigma_k(\underline{n}, \mu(b))) = \tau_0^1(\underline{n-1}) = \underline{n}$ if $n > k+1$
    * $\tau_0^1(\sigma_k(\mu(\underline{n}), \mu(b))) = \tau_0^1(\sigma_k(\underline{n}, \mu(b))) = \tau_0^1(\tau_0^k(\mu(b))) = \tau_0^{k+1}(\mu(b))$ if $n = k+1$, from $\tau_i^n \circ \tau_i^m = \tau_i^{n+m}$.
  - case $s = \Uparrow^k(\uparrow)$
    $\mu(\underline{n+1}[\Uparrow(s)]) = \tau_{k+1}^1(\mu(\underline{n+1})) = \tau_{k+1}^1(\underline{n+1})$.
    Thus
    * $\underline{n+2}$ if $n > k$
    * $\underline{n+1}$ if $n \leq k$.

    $\mu(\underline{n}[s][\uparrow]) = \tau_0^1(\mu(\underline{n}[\Uparrow^k\ (\uparrow)])) = \tau_0^1\tau_k^1(\mu(\underline{n})) = \tau_0^1\tau_k^1(\underline{n})$.
    Therefore
    * $\tau_0^1(\underline{n+1}) = \underline{n+2}$ if $n > k$
    * and $\tau_0^1(\underline{n}) = \underline{n+1}$ if $n \leq k$.
- $\underline{n}[\uparrow] \underset{v}{\longrightarrow} \underline{n+1}$
  $\mu(\underline{n}[\uparrow]) = \tau_0^1(\mu(\underline{n})) = \tau_0^1(\underline{n}) = \underline{n+1} = \mu(\underline{n+1})$.

The proof of 2 comes from $\sigma_0(\mu(a), \mu(b)) = \mu(a[b/])$, by definition of μ. $\mu([b/]) = v([b/])$ comes from Part 1. □

3 Confluence of $\lambda\upsilon$

$\lambda\upsilon$ is confluent and this requires a few lemmas. Notice that in $\lambda\upsilon$, every term can be written as either a or $a[\Uparrow^{i_1}(s_1)]\ldots[\Uparrow^{i_k}(s_k)]\ldots[\Uparrow^{i_n}(s_n)]$ where a is not a closure and s_k is either $\uparrow$ or $b/$. We want to prove that for every term a, $a[b/][s] \overset{*}{\underset{\upsilon}{\longleftrightarrow}} a[\Uparrow(s)][b[s]/]$, that we call υ-convertibility of $a[b/][s]$ and $a[\Uparrow(s)][b[s]/]$. We prove the υ-convertibility when a is a pure term, but the result remains true for impure terms since if a is impure

$$a[b/][s] \overset{*}{\underset{\upsilon}{\longleftrightarrow}} \upsilon(a)[b/][s] \overset{*}{\underset{\upsilon}{\longleftrightarrow}} \upsilon(a)[\Uparrow(s)][b[s]/] \overset{*}{\underset{\upsilon}{\longleftrightarrow}} a[\Uparrow(s)][b[s]/].$$

The same lifting from pure terms to impure terms is true for every lemma which follows. That result is the $\lambda\upsilon$ version of the *substitution lemma* which plays a fundamental role in λ-calculus. For that we need to prove several lemmas about $\lambda\upsilon$.

Lemma 1 *For $n \geq 1$ and $i \geq 0$, $\underline{n}[\Uparrow^{n+i}(s)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n}$.*

Proof: By induction on n. If $n = 1$, this is just rule *FVarLift*. In general,

$$\underline{n+1}[\Uparrow^{n+i+1}(s)] \underset{\upsilon}{\longrightarrow} \underline{n}[\Uparrow^{n+i}(s)][\uparrow] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n}[\uparrow] \underset{\upsilon}{\longrightarrow} \underline{n+1}.$$

□

We assume now that

$$a[\![\uparrow^i]\!] \equiv a[\uparrow]\ldots{\scriptstyle i \text{ times}}\ldots[\uparrow].$$

Obviously,

$$\underline{n}[\![\uparrow^i]\!] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n+i}.$$

Proof of next lemma resembles this of Lemma 1.

Lemma 2 *For $n \geq 1$ and $i \geq 0$, $\underline{n+i}[\Uparrow^i(s)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n}[s][\![\uparrow^i]\!]$.*

Corollary 1 *For $n > i \geq 1$, $\underline{n}[\Uparrow^i(\uparrow)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n+1}$.*

Proof: By Lemma 2,

$$\underline{n}[\Uparrow^i(\uparrow)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n-i}[\uparrow][\![\uparrow^i]\!]$$

and it is clear that

$$\underline{n-i}[\uparrow][\![\uparrow^i]\!] = \underline{n-i}[\![\uparrow^{i+1}]\!] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n+1}.$$

□

Lemma 3 *For $i \geq 1$, $a[\Uparrow^i(\uparrow)][\Uparrow^i(b/)] \overset{*}{\underset{\upsilon}{\longrightarrow}} a$.*

Proof: By structural induction,

1. a is a variable $\underline{n}$ with $n \leq i$, one uses Lemma 1,

$$\underline{n}[\Uparrow^i(\uparrow)][\Uparrow^i(b/)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n}[\Uparrow^i(b/)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n}.$$

2. a is a variable $\underline{n}$ with $n > i$, one uses Lemma 2,

$$\underline{n}[\Uparrow^i(\uparrow)][\Uparrow^i(b/)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n-i}[\uparrow][\![\uparrow^i]\!][\Uparrow^i(b/)] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n+1}[\Uparrow^i(b/)]$$

$$\overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n+1-i}[b/][\![\uparrow^i]\!] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n-i}[\![\uparrow^i]\!] \overset{*}{\underset{\upsilon}{\longrightarrow}} \underline{n}.$$

3. a is an application $a_1 a_2$,

$$(a_1a_2)[\Uparrow^i(\uparrow)][\Uparrow^i(b/)] \xrightarrow[v]{*} (a_1[\Uparrow^i(\uparrow)][\Uparrow^i(b/)])\ (a_2[\Uparrow^i(\uparrow)][\Uparrow^i(b/)]) \xrightarrow[v]{*} a_1a_2.$$

The last rewrite comes by induction.

4. a is an abstraction $\lambda(a')$,

$$\lambda(a')[\Uparrow^i(\uparrow)][\Uparrow^i(b/)] \xrightarrow[v]{*} \lambda(a'[\Uparrow^{i+1}(\uparrow)][\Uparrow^{i+1}(b/)]) \xrightarrow[v]{*} \lambda(a').$$

The last rewrite comes again by induction.

□

Lemma 4 *For all $j \geq i \geq 0$, $a[\Uparrow^i(\uparrow)][\Uparrow^{j+1}(\uparrow)] \overset{*}{\underset{v}{\longleftrightarrow}} a[\Uparrow^j(\uparrow)][\Uparrow^i(\uparrow)]$.*

Proof: By structural induction,

- a is a variable $\underline{n}$ with $n \leq i$, one applies Lemma 1

$$\underline{n}[\Uparrow^i(\uparrow)][\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{*} \underline{n}[\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{*} \underline{n},$$

$$\underline{n}[\Uparrow^j(\uparrow)][\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n}[\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n}.$$

- a is a variable $\underline{n}$ with $i < n \leq j$,

$$\underline{n}[\Uparrow^i(\uparrow)][\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{*} \underline{n+1}[\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{*} \underline{n+1},$$

$$\underline{n}[\Uparrow^j(\uparrow)][\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n+1}[\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n+1}.$$

- a is a variable $\underline{n}$ with $j < n$,

$$\underline{n}[\Uparrow^i(\uparrow)][\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{*} \underline{n+1}[\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{*} \underline{n+2},$$

$$\underline{n}[\Uparrow^j(\uparrow)][\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n+1}[\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n+2}.$$

- If a is an application it is trivial.
- If a is an abstraction $\lambda(a')$,

$$\lambda(a')[\Uparrow^i(\uparrow)][\Uparrow^{j+1}(\uparrow)] \xrightarrow[v]{} \lambda(a'[\Uparrow^{i+1}(\uparrow)][\Uparrow^{j+2}(\uparrow)])$$

by induction

$$\overset{*}{\underset{v}{\longleftrightarrow}} \lambda(a'[\Uparrow^{j+1}(\uparrow)][\Uparrow^{i+1}(\uparrow)]) \xleftarrow[v]{*} \lambda(a')[\Uparrow^j(\uparrow)][\Uparrow^i(\uparrow)].$$

□

Corollary 2 $a[\uparrow][\Uparrow^{i+1}(\uparrow)] \overset{*}{\underset{v}{\longleftrightarrow}} a[\Uparrow^i(\uparrow)][\uparrow]$, *when* $i = 0$ $a[\uparrow][\Uparrow(\uparrow)] \overset{*}{\underset{v}{\longleftrightarrow}} a[\uparrow][\uparrow]$.

Corollary 3 *For $i \geq 0$, $a[\![\uparrow^i]\!][\Uparrow^i(\uparrow)] \overset{*}{\underset{v}{\longleftrightarrow}} a[\![\uparrow^{i+1}]\!]$.*

Proof: One iterates corollary 2 i times. □

Lemma 5 $a[\Uparrow^i(\uparrow)][\Uparrow^{i+1}(s)] \overset{*}{\underset{v}{\longleftrightarrow}} a[\Uparrow^i(s)][\Uparrow^i(\uparrow)]$.

Proof: By structural induction

- a is a variable $\underline{n}$ with $n \leq i$, one applies Lemma 1.

- a is a variable $\underline{n}$ with $n > i$, one uses corollary 1.

$$\underline{n}[\Uparrow^i(\uparrow)][\Uparrow^{i+1}(s)] \xrightarrow[v]{*} \underline{n+1}[\Uparrow^{i+1}(s)] \xrightarrow[v]{*} \underline{n-i}[s][\![\uparrow^{i+1}]\!]$$

where the last derivation is Lemma 2.

$$\underline{n}[\Uparrow^i(s)][\Uparrow^i(\uparrow)] \xrightarrow[v]{*} \underline{n-i}[s][\![\uparrow^i]\!][\Uparrow^i(\uparrow)] \overset{*}{\underset{v}{\longleftrightarrow}} \underline{n-i}[s][\![\uparrow^{i+1}]\!]$$

the last statement is a consequence of corollary 3.

The two other cases, namely if a is an abstraction or an application work as previously. □

Lemma 5 has an important corollary.

Corollary 4 $a[\uparrow][\Uparrow(s)] \overset{*}{\underset{v}{\longleftrightarrow}} a[s][\uparrow]$.

By the corollary it entails, the next lemma is the key of the confluence of λv.

Lemma 6 $a[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)] \overset{*}{\underset{v}{\longleftrightarrow}} a[\Uparrow^i(b/)][\Uparrow^i(s)]$.

Proof: By structural induction

- a is a variable $\underline{n}$ with $n \leq i$, one uses routinely Lemma 1.
- a is a variable $\underline{n}$ with $n > i+1$. By Lemma 2,

$$\underline{n}[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)] \xrightarrow[v]{*} \underline{n-i-1}[s][\![\uparrow^{i+1}]\!][\Uparrow^i(b[s]/)]$$

by corollary 3

$$\overset{*}{\underset{v}{\longleftrightarrow}} \underline{n-i-1}[s][\![\uparrow^i]\!][\Uparrow^i(\uparrow)][\Uparrow^i(b[s]/)]$$

by Lemma 3

$$\xrightarrow[v]{*} \underline{n-i-1}[s][\![\uparrow^i]\!].$$

On the other hand, by Lemma 2

$$\underline{n}[\Uparrow^i(b/)][\Uparrow^i(s)] \xrightarrow[v]{*} \underline{n-i}[b/][\![\uparrow^i]\!][\Uparrow^i(s)] \xrightarrow[v]{} \underline{n-i-1}[\![\uparrow^i]\!][\Uparrow^i(s)]$$

and by corollary 4 applied i times,

$$\xrightarrow[v]{*} \underline{n-i-1}[s][\![\uparrow^i]\!].$$

- a is the variable $\underline{i+1}$. By Lemma 2 and Lemma 1,

$$\underline{i+1}[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)] \xrightarrow[v]{*} \underline{i+1}[\Uparrow^i(b[s]/)]$$

$$\xrightarrow[v]{*} \underline{1}[b[s]/][\![\uparrow^i]\!] \xrightarrow[v]{} b[s][\![\uparrow^i]\!].$$

By Lemma 2,

$$\underline{i+1}[\Uparrow^i(b/)][\Uparrow^i(s)] \xrightarrow[v]{*} \underline{1}[b/][\![\uparrow^i]\!][\Uparrow^i(s)] \xrightarrow[v]{} b[\![\uparrow^i]\!][\Uparrow^i(s)]$$

and by corollary 4 applied i times,

$$\xrightarrow[v]{*} b[s][\![\uparrow^i]\!].$$

- a is an application $a_1 a_2$,

$$(a_1 a_2)[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)] \xrightarrow[v]{*} (a_1[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)])\,(a_2[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)])$$

and by induction,

$$\overset{*}{\underset{v}{\longleftrightarrow}} (a_1[\Uparrow^i(b/)][\Uparrow^i(s)])\,(a_2[\Uparrow^i(b/)][\Uparrow^i(s)]) \xleftarrow[v]{*} (a_1 a_2)[\Uparrow^i(b/)][\Uparrow^i(s)].$$

- a is an abstraction $\lambda(a')$,

$$(\lambda a')[\Uparrow^{i+1}(s)][\Uparrow^i(b[s]/)] \xrightarrow[v]{*} \lambda(a'[\Uparrow^{i+2}(s)][\Uparrow^{i+1}(b[s]/)])$$

and by induction

$$\lambda(a'[\Uparrow^{i+1}(b/)][\Uparrow^{i+1}(s)]) \xleftarrow[v]{*} (\lambda a')[\Uparrow^i(b/)][\Uparrow^i(s)].$$

□

Corollary 5 (Substitution Lemma) $a[b/][s] \overset{*}{\underset{v}{\longleftrightarrow}} a[\Uparrow(s)][b[s]/]$.

Corollary 5 is the λv-version of the fundamental *substitution lemma* of classical λ-calculus which is a key of its confluence. In [2] Lemma 2.1.16, it reads as

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

and in [6], Exercise 1.2.7.2 as

$$M[k \leftarrow N][n \leftarrow P] \;\rightarrow\; M[n+1 \leftarrow P][k \leftarrow N[n-k \leftarrow P]].$$

For its use in the next lemma, Substitution Lemma has to be iterated.

Corollary 6 $a[b/][s_1]\ldots[s_p] \overset{*}{\underset{v}{\longleftrightarrow}} a[\Uparrow(s_1)]\ldots[\Uparrow(s_p)][b[s_1]\ldots[s_p]/]$.

Lemma 7 (Projection Lemma) *If* $a \xrightarrow[B]{} b$ *then* $v(a) \xrightarrow[\beta]{*} v(b)$. *If* $s \xrightarrow[B]{} t$ *then* $v(s) \xrightarrow[\beta]{*} v(t)$.

Proof: We prove the statement for a and s together. Let u stands either for a or s; v for either b or t. We proceed by noetherian induction on the lexicographic product of the two well-founded relations $(\xrightarrow[v]{}, \sqsupset)$, where $\sqsupset$ is the subterm relation. We distinguish cases according to the structure of u.

- If $a \equiv a_1 a_2$ is an application and if the B-redex is in a_1, since $a_1 a_2 \sqsupset a_1$ and $a_1 \xrightarrow[B]{} b_1$ by induction one gets $v(a_1) \xrightarrow[\beta]{*} v(b_1)$ and

$$v(a_1 a_2) = v(a_1)v(a_2) \xrightarrow[\beta]{*} v(b_1)v(a_2) = v(b_1 a_2).$$

We proceed likewise if the B-redex is in a_2 or if $a \equiv \lambda a_1$.

- If the B-redex is $a \equiv (\lambda a_1)a_2$ then $b \equiv a_1[a_2/]$ and $v(a) = (\lambda v(a_1))v(a_2)$. By definition of β, one has

$$v(a) \xrightarrow[\beta]{*} v(v(a_1)[v(a_2)/]) = v(b).$$

- If a is a closure then $a \equiv a'[s_1]\ldots[s_p]$ and $b \equiv b'[t_1]\ldots[t_p]$.

- $a \equiv (a_1 a_2)[s_1]\dots[s_p]$ and $b \equiv (b_1 a_2)[s_1]\dots[s_p]$. If the B redex occurs inside a_1 with $a_1 \xrightarrow[B]{} b_1$ then $a_1[s_1]\dots[s_p] \xrightarrow[B]{} b_1[s_1]\dots[s_p]$, by induction

$$v(a_1[s_1]\dots[s_p]) \xrightarrow[\beta]{*} v(b_1[s_1]\dots[s_p])$$

and

$$\begin{aligned} v((a_1a_2)[s_1]\dots[s_p]) &= v(a_1[s_1]\dots[s_p])v(a_2[s_1]\dots[s_p]) \\ &\xrightarrow[\beta]{*} v(b_1[s_1]\dots[s_p])v(a_2[s_1]\dots[s_p]) = v((b_1a_2)[s_1]\dots[s_p]), \end{aligned}$$

and the same if the B rewrite takes place inside a_2 or inside s.

- $a = ((\lambda a_3)a_2)[s_1]\dots[s_p]$ and $b = a_3[a_2/][s_1]\dots[s_p]$.

$$\begin{aligned} v(a) &= \lambda(v(a_3[\Uparrow(s_1)]\dots[\Uparrow(s_p)])v(a_2[s_1]\dots[s_p]) \\ &\xrightarrow[\beta]{*} v(v(a_3[\Uparrow(s_1)]\dots[\Uparrow(s_p)])[v(a_2[s_1]\dots[s_p])/]) \\ &= v(a_3[\Uparrow(s_1)]\dots[\Uparrow(s_p)][v(a_2[s_1]\dots[s_p])/] \end{aligned}$$

and by corollary 6,

$$= v(a_3[a_2/][s_1]\dots[s_p]) = v(b).$$

- $a = (\lambda a_1)[s_1]\dots[s_p]$. If $a_1 \xrightarrow[B]{} b_1$ or $s_i \xrightarrow[B]{} t_i$,

$$a_1[\Uparrow(s_1)]\dots[\Uparrow(s_p)] \xrightarrow[B]{} b_1[\Uparrow(t_1)]\dots[\Uparrow(t_p)]$$

and we can apply the induction hypothesis.

- $a = \underline{n}[s_1]\dots[s_p]$. The B redex is inside a s_i with $s_i = \Uparrow^{j_i}(c_i/)$. If $i > 1$, then $\underline{n}[s_1] \xrightarrow[v]{+} a_1$ where a_1 is not a closure and

$$a_1[s_2]\dots[s_p] \xrightarrow[B]{} a_1[t_2]\dots[t_p]$$

where all the t_j are equal to s_j except t_i which is $\Uparrow^{j_i}(c_i/)$ with $c_i \xrightarrow[B]{} d_i$. The result comes by induction. If the B redex is inside s_1,

$$s_1 = \Uparrow^{j_1}(c_1/) \xrightarrow[B]{} t_1 = \Uparrow^{j_1}(d_1/).$$

By case, one gets:

  * $n = 1$ and $j_1 = 0$,

$$\underline{1}[c_1/][s_2]\dots[s_p] \xrightarrow[v]{} c_1[s_2]\dots[s_p] \xrightarrow[B]{} d_1[s_2]\dots[s_p]$$

$$\underline{1}[d_1/][s_2]\dots[s_p] \xrightarrow[v]{} d_1[s_2]\dots[s_p]$$

  and the result comes by induction.

  * $n = k+1$ and $j_1 = 0$,

$$\underline{k+1}[c_1/][s_2]\dots[s_p] \xrightarrow[v]{} \underline{k}[s_2]\dots[s_p]$$

$$\underline{k+1}[d_1/][s_2]\dots[s_p] \xrightarrow[v]{} \underline{k}[s_2]\dots[s_p]$$

  and the result is immediate.

  * $n = 1$ and $j_1 = j+1$,

$$\underline{1}[\Uparrow^{j+1}(c_1/)][s_2]\dots[s_p] \xrightarrow[v]{} \underline{1}[s_2]\dots[s_p]$$

$$\underline{1}[\Uparrow^{j+1}(d_1/)][s_2]\dots[s_p] \xrightarrow[v]{} \underline{1}[s_2]\dots[s_p]$$

  and like above the result is immediate.

* $n = k+1$ and $j_1 = j+1$.

$$\underline{k+1}[\Uparrow^{j+1}(c_1/)][s_2]\dots[s_p] \underset{\upsilon}{\longrightarrow} \underline{k}[\Uparrow^{j}(c_1/)][\uparrow][s_2]\dots[s_p]$$

$$\underline{k+1}[\Uparrow^{j+1}(d_1/)][s_2]\dots[s_p] \underset{\upsilon}{\longrightarrow} \underline{k}[\Uparrow^{j}(d_1/)][\uparrow][s_2]\dots[s_p]$$

and the result comes by induction.

□

Theorem 1 (Confluence Theorem) *$\lambda\upsilon$ is confluent on closed terms.*

Proof: The proof of the theorem resembles the proof of a similar theorem by Abadi et al. in [1] which in turn was based on Hardin's interpretation method [13] with modifications due to the change of substitution calculus from σ to υ. It relies on the Projection Lemma. □

4 The typed $\lambda\upsilon$-calculus

We can type $\lambda\upsilon$. For this one introduces two new concepts, namely *types* and *contexts*. In addition we type the operator λ and we write $\lambda A.a$ instead of λa where A is a type namely the type of the variable $\underline{1}$ in a. We define now the system $\Lambda\Upsilon$ for typing $\lambda\upsilon$. A proof of termination for a simply typed calculus with *weak* reduction is in [5]

4.1 The typing system $\Lambda\Upsilon$

The grammar of $\Lambda\Upsilon$ is:

Type	A	$::=$	$\mathsf{A}_1 \mid \dots \mid \mathsf{A}_n \mid A \Rightarrow B$
Naturals	n	$::=$	$1 \mid n+1$
Terms	a	$::=$	$\underline{n} \mid ab \mid \lambda A.a \mid a[s]$
Substitutions	s	$::=$	$a/ \mid \uparrow \mid \Uparrow(s)$
Context	Γ	$::=$	$[\,] \mid A \cdot \Gamma$

where $\mathsf{A}_1 \dots \mathsf{A}_n$ is a family of atomic types.

Contexts and naturals are well-typed. Contexts are the "type" of substitutions. The rules for typing *Terms* and *Substitutions* are:

Terms

$$\frac{\Gamma \vdash a : A \Rightarrow B \quad \Gamma \vdash b : A}{\Gamma \vdash ab : B} \qquad \frac{A \cdot \Gamma \vdash a : B}{\Gamma \vdash \lambda A.a : A \Rightarrow B}$$

$$\frac{\Gamma \vdash a : A \quad \Delta \vdash s : \Gamma}{\Delta \vdash a[s] : A} \qquad \frac{}{A \cdot \Gamma \vdash \underline{1} : A} \qquad \frac{\Gamma \vdash \underline{n} : A}{B \cdot \Gamma \vdash \underline{n+1} : A}$$

Substitutions

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a/ : A \cdot \Gamma} \qquad \frac{}{A \cdot \Gamma \vdash \uparrow : \Gamma} \qquad \frac{\Gamma \vdash s : \Delta}{A \cdot \Gamma \vdash \Uparrow(s) : A \cdot \Delta}$$

4.2 Preservation of types by $\lambda\upsilon$: subject reduction theorem

We show that for each rule of $\lambda\upsilon$ the type of the left-hand side is the same as the type of the right-hand side.

B: $(\lambda a)b \to a[b/]$

$$\frac{\dfrac{B \cdot \Gamma \ \vdash \ a : \ A}{\Gamma \ \vdash \ \lambda B.a : \ B \Rightarrow A} \qquad \Gamma \ \vdash \ b : \ B}{\Gamma \ \vdash \ (\lambda B.a)b : \ A}$$

$$\frac{B \cdot \Gamma \ \vdash \ a : \ A \qquad \dfrac{\Gamma \ \vdash \ b : \ B}{\Gamma \ \vdash \ b/ : \ B \cdot \Gamma}}{\Gamma \ \vdash \ a[b/] : \ A}$$

App: $(ab)[s] \to a[s]b[s]$

$$\frac{\dfrac{\Gamma \ \vdash \ a : \ B \Rightarrow A \quad \Gamma \ \vdash \ b : \ B}{\Gamma \ \vdash \ ab : \ A} \qquad \Delta \ \vdash \ s : \ \Gamma}{\Delta \ \vdash \ (ab)[s] : \ A}$$

$$\frac{\dfrac{\Gamma \ \vdash \ a : \ B \Rightarrow A \quad \Delta \ \vdash \ s : \ \Gamma}{\Delta \ \vdash \ a[s] : \ B \Rightarrow A} \qquad \dfrac{\Gamma \ \vdash \ b : \ B \quad \Delta \ \vdash \ s : \ \Gamma}{\Delta \ \vdash \ b[s] : \ B}}{\Delta \ \vdash \ a[s]b[s] : \ A}$$

Lambda: $(\lambda a)[s] \to \lambda(a[\Uparrow(s)])$

$$\frac{\dfrac{B \cdot \Gamma \ \vdash \ a : \ A}{\Gamma \ \vdash \ \lambda B.a : \ B \Rightarrow A} \qquad \Delta \ \vdash \ s : \ \Gamma}{\Delta \ \vdash \ (\lambda B.a)[s] : \ B \Rightarrow A}$$

$$\frac{\dfrac{B \cdot \Gamma \ \vdash \ a : \ A \qquad \dfrac{\Delta \ \vdash \ s : \ \Gamma}{B \cdot \Delta \ \vdash \Uparrow(s) : \ B \cdot \Gamma}}{B \cdot \Delta \ \vdash \ a[\Uparrow(s)] : \ A}}{\Delta \ \vdash \ \lambda B.(a[\Uparrow(s)]) : \ B \Rightarrow A}$$

FVar: $\underline{1}[a/] \to a$

$$\frac{\dfrac{}{A \cdot \Gamma \ \vdash \ \underline{1} : \ A} \qquad \dfrac{\Gamma \ \vdash \ a : \ A}{\Gamma \ \vdash \ a/ : \ A \cdot \Gamma}}{\Gamma \ \vdash \ \underline{1}[a/] : \ A}$$

RVar: $\underline{n+1}[a/] \to \underline{n}$

$$\frac{\dfrac{\Gamma \ \vdash \ \underline{n} : \ A}{B \cdot \Gamma \ \vdash \ \underline{n+1} : \ A} \qquad \dfrac{\Gamma \ \vdash \ a : \ B}{\Gamma \ \vdash \ a/ : \ B \cdot \Gamma}}{\Gamma \ \vdash \ \underline{n+1}[a/] : \ A}$$

FVarLift: $\underline{1}[\Uparrow(s)] \to \underline{1}$

$$\frac{\dfrac{}{A \cdot \Gamma \ \vdash \ \underline{1} : \ A} \qquad \dfrac{\Delta \ \vdash \ s : \ \Gamma}{A \cdot \Delta \ \vdash \Uparrow(s) : \ A \cdot \Gamma}}{A \cdot \Delta \ \vdash \ \underline{1}[\Uparrow(s)] : \ A}$$

RVarLift: $\underline{n+1}[\Uparrow(s)] \to \underline{n}[s][\uparrow]$

$$\frac{\dfrac{\Gamma \vdash \underline{n} : A}{B \cdot \Gamma \vdash \underline{n+1} : A} \qquad \dfrac{\Delta \vdash s : \Gamma}{B \cdot \Delta \vdash \Uparrow(s) : B \cdot \Gamma}}{B \cdot \Delta \vdash \underline{n+1}[\Uparrow(s)] : A}$$

$$\frac{\dfrac{\Gamma \vdash \underline{n} : A \qquad \Delta \vdash s : \Gamma}{\Delta \vdash \underline{n}[s] : A} \qquad \dfrac{}{B \cdot \Delta \vdash \uparrow : \Delta}}{B \cdot \Delta \vdash \underline{n}[s][\uparrow] : A}$$

VarShift: $\underline{n}[\uparrow] \to \underline{n+1}$

$$\frac{\Gamma \vdash \underline{n} : A \qquad \dfrac{}{B \cdot \Gamma \vdash \uparrow : \Gamma}}{B \cdot \Gamma \vdash \underline{n}[\uparrow] : A}$$

$$\frac{\Gamma \vdash \underline{n} : A}{B \cdot \Gamma \vdash \underline{n+1} : A}$$

5 Typed $\lambda\upsilon$-calculus is strongly normalizing

The essential difference between $\xrightarrow[\beta]{}$ and $\xrightarrow[\lambda\upsilon]{*}$ is that β rewrites with B and then normalizes with υ in order to remove all the closures, whereas $\lambda\upsilon$ rewrites also with B but may postpone reductions of closures created by B. In the case of simply typed λ-calculus we know that β-reduction terminates or is strongly normalizing [11]. Is it the same for $\lambda\upsilon$? The answer is "yes", but is not so obvious. Ritter [19] mentions this problem as open for any calculus of explicit substitution. Indeed it could happen that $a \xrightarrow[B]{} b$ and $\upsilon(a) = \upsilon(b)$. In that case, the reduced B-redex of a lies in the substitution part of a subterm which is a closure. That closure is eliminated by rule $Rvar$ or rule $FVarLift$ which are the only rules of υ that can delete a B-redex[1]. Thus in the projection lemma, it could be the case that we perform a B-reduction that does not correspond to a β-reduction in the "projected" part, we can therefore make more (but not infinitely many more) B-reductions than β-reductions. First let us remind the reader what we call a *position* in a term. Although it has been understood in what precedes, it plays a main role in the following rule and has to be made precise.

Definition 1 (Position) *A position in a λ-term t is a sequence of numbers* 1 *or* 2*, such that*

- $t_{|\epsilon} = t$
- *If* $t_{|p} = a[s]$*, then* $t_{|p1} = a$ *and* $t_{|p2} = s$.
- *If* $t_{|p} = \lambda(a)$*, then* $t_{|p1} = a$.
- *If* $t_{|p} = a_1 a_2$*, then* $t_{|p1} = a_1$ *and* $t_{|p2} = a_2$.

Definition 2 (Replacement) *The term $t\{u\}_p$ obtained by replacing the subterm at position p by u is the term written $t\{u\}_p$ and defined by*

- $(t\{u\}_p)_{|pp'} = u_{|p'}$,
- $(t\{u\}_p)_{|p'} = t_{|p'}\{u\}_{p''}$ *if* $p = p'p''$.

[1] *App* also deletes B-redexes, but *Lambda* enables them immediately.

- $(t\{u\}_p)_{|q} = t_{|q}$ *if p and q are disjoint, i.e., q is none of the above cases.*

Rewriting the term t at the position p by the rule B into the term t' means that there exists a substitution (in the usual sense) f such that $t_{|p} = \mathsf{f}((\lambda a)b)$ and $t' = t\{\mathsf{f}(a[b/])\}_p$. We write that $t \underset{B,p}{\longrightarrow} t'$.

Before proving the next theorem, let us give two definitions and prove two lemmas.

Definition 3 (External position) *The set $Ext(a)$ of* external positions *of a term a is the set defined as:*

$$\begin{array}{rcl} Ext(ab) & = & 1Ext(a) \;\cup\; 2Ext(a) \cup \{\epsilon\} \\ Ext(\lambda a) & = & 1Ext(a) \cup \{\epsilon\} \\ Ext(a[s]) & = & 1Ext(a) \cup \{\epsilon\} \\ Ext(\underline{n}) & = & \{\epsilon\} \end{array}$$

Intuitively external positions are those under no brackets, i.e., in no substitution part of any closure. A rewrite that takes place at an external position is said *external*, otherwise it is said internal. If one wants to make precise that a rewrite $\underset{\lambda\upsilon}{\longrightarrow}$ is external (resp. internal) one writes $\underset{\lambda\upsilon}{\longrightarrow}{}^{ext}$ (resp. $\underset{\lambda\upsilon}{\longrightarrow}{}^{int}$).

Lemma 8 *If $p \in Ext(a)$ and if $a \underset{B,p}{\longrightarrow} b$, then $\upsilon(a) \overset{+}{\underset{\beta}{\longrightarrow}} \upsilon(b)$. In particular, $\upsilon(a) \neq \upsilon(b)$.*

Definition 4 (Minimal $\lambda\upsilon$ derivation) *An infinite $\lambda\upsilon$ derivation*

$$a_1 \underset{B,p_1}{\longrightarrow} a'_1 \overset{*}{\underset{\upsilon}{\longrightarrow}} a_2 \ldots a_i \underset{B,p_i}{\longrightarrow} a'_i \overset{*}{\underset{\upsilon}{\longrightarrow}} a_{i+1} \ldots .$$

is minimal *if for $a_1 \ldots a_n \underset{B,q}{\longrightarrow} b \overset{*}{\underset{\upsilon}{\longrightarrow}} \ldots$ another infinite derivation, then for all p', $q \neq p_n p'$ (see Figure 3).*

That means that either p_n and q are disjoint positions or q is above p_n (i.e., $p_n = qq'$). In other words, one rewrites always the lowest possible redex to keep non termination.

Lemma 9 *In an infinite derivation containing only internal B-rewrites, one can assume that there is no external υ-rewrite either. More specifically if*

$$a_1 \underset{\lambda\upsilon}{\longrightarrow} a_2 \underset{\lambda\upsilon}{\longrightarrow} \ldots a_n \underset{\lambda\upsilon}{\longrightarrow} a_{n+1} \ldots$$

then there exists an infinite derivation

$$a_1 \overset{*}{\underset{\upsilon}{\longrightarrow}} b_1 \underset{\lambda\upsilon}{\longrightarrow} b_2 \underset{\lambda\upsilon}{\longrightarrow} \ldots b_n \underset{\lambda\upsilon}{\longrightarrow} b_{n+1} \ldots$$

where the rewrites from a_1 to b_1 are the only external rewrites. Moreover this construction does not destroy the minimality of the B-rewrites.

> **Proof:** One can prove that $a \underset{\lambda\upsilon}{\longrightarrow}{}^{int} \cdot \underset{\upsilon}{\longrightarrow}{}^{ext} b$ implies $a \underset{\upsilon}{\longrightarrow}{}^{ext} \cdot \overset{*}{\underset{\lambda\upsilon}{\longrightarrow}}{}^{int} b$. The proof is by induction on the structure of a. If the external rewrite does not take place at the root, the result comes by induction. If the external rewrite takes place at the root of a this means that a is a closure and the proof comes by case on the rule of υ which is applied in the external rewrite. If the rule is *App* the internal rewrite in $a \underset{\lambda\upsilon}{\longrightarrow}{}^{int} \cdot \underset{\upsilon}{\longrightarrow}{}^{ext} b$ is replaced by two internal rewrites in $a \underset{\upsilon}{\longrightarrow}{}^{ext} \cdot \overset{*}{\underset{\lambda\upsilon}{\longrightarrow}}{}^{int} b$. If the rule is *Lambda* the result is evident. For the other rules, one notices that the internal rewrite never applies to a term of sort substitution, therefore the $\underset{\upsilon}{\longrightarrow}{}^{ext}$ rewrite is preserved.
>
> The termination (strong normalization) of υ limits the number of υ-rewrites one can perform starting from a_1, thus completes the proof of the lemma. □

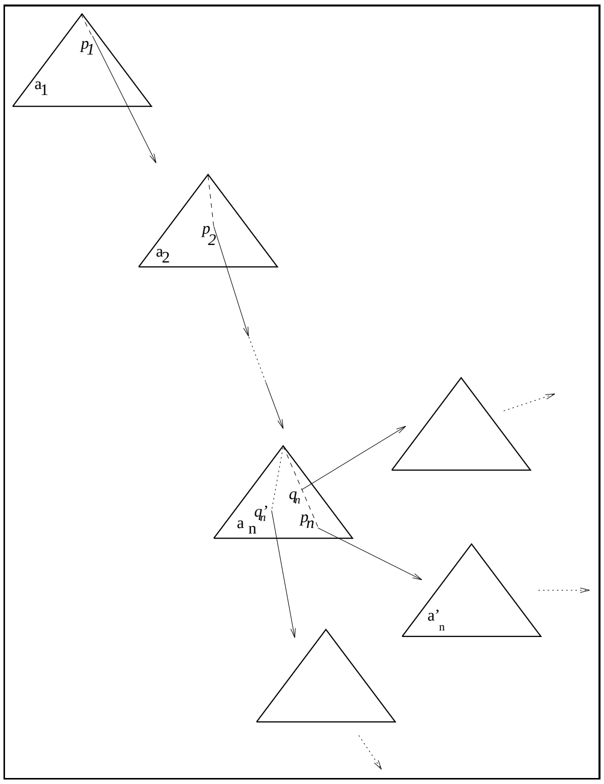

Figure 3: A minimal λv derivation

Theorem 2 *Every pure well-typed term of $\Lambda\Upsilon$ is strongly normalizable.*

Proof: We use a method based on a least counter-example. Let us consider a *minimal* infinite $\lambda\upsilon$ derivation starting from a pure term a_1

$$a_1 \underset{B,p_1}{\longrightarrow} a'_1 \overset{*}{\underset{\upsilon}{\longrightarrow}} a_2 \ldots a_i \underset{B,p_i}{\longrightarrow} a'_i \overset{*}{\underset{\upsilon}{\longrightarrow}} a_{i+1} \ldots .$$

From projection lemma and termination of β, there is some N such that $i \geq N$ implies $\upsilon(a_i) = \upsilon(a_N)$. This means that after N, υ-normal forms remain the same.

According to Lemma 8, for $i \geq N$, B-rewrites take place always at internal positions. By Lemma 9, we can assume that for $i \geq N$ all the $\lambda\upsilon$-rewrites are internal. We have then a closure at an external position p such that

$$a_N \overset{*}{\underset{\lambda\upsilon}{\longrightarrow}} a_{j_1} = a\{b_{j_1}[c_{j_1}/]\}_p \underset{\lambda\upsilon}{\longrightarrow} a_{j_2} = a\{b_{j_1}[c_{j_2}/]\}_p \underset{\lambda\upsilon}{\longrightarrow} \ldots \underset{\lambda\upsilon}{\longrightarrow} a_{j_k} = a\{b_{j_k}[c_{j_k}/]\}_p \ldots$$

The indices j_k's are chosen such that $c_{j_1} \underset{\lambda\upsilon}{\longrightarrow} c_{j_2} \underset{\lambda\upsilon}{\longrightarrow} \ldots \underset{\lambda\upsilon}{\longrightarrow} c_{j_k} \ldots$ is an infinite sequence. By Lemma 8, we know that the closure $b_{j_1}[\cdot]$ has been created sometime before N, say at rank J, by a B-rewrite

$$a_J = d\{(\lambda e)c\}_{p_J} \underset{B,p_J}{\longrightarrow} d\{e[c/]\}_{p_J} = a_{J+1}$$

$e \overset{*}{\underset{\lambda\upsilon}{\longrightarrow}} b_{j_1}$ and $c \overset{*}{\underset{\lambda\upsilon}{\longrightarrow}} c_{j_1}$. By rewriting a subterm of c at position say q in $d\{(\lambda e)c\}_q$ we can create an infinite sequence that B-rewrites a_J at position q lower than p_J, which is a contradiction with the minimality of the sequence (a_i). $\square$

6 The U-machine

In this section we consider *weak head* normalization and *strong* normalization in λ-calculus, they can be done by an abstract machine. We define the U-machine which is an *environment* machine, more precisely a SEC machine, similar to Krivine's machine described by Curien in his book [6]. A state of the machine has three components: a term, an environment in which the term must be evaluated and a stack which contains *closures*. A closure is a pair of a term and of an associated environment, whose evaluation is postponed. Transitions allow to go from a state to another one. The U-machine is deterministic, each non final state is matched by only one transition rule. Each rule corresponds to one rule of the $\lambda\upsilon$-calculus, except for one (denoted $LBA - BET$) which is connected with the two rules (Lambda) and (B).

Thus the structure of the U-machine is

$$\begin{array}{lcl} \mathsf{state} & = & \mathsf{term} \times \mathsf{env} \times \mathsf{stack} \\ \mathsf{env} & = & ((\uparrow \cup \mathsf{closure}) \times \mathbb{N})\ \mathsf{list} \\ \mathsf{closure} & = & \mathsf{term} \times \mathsf{env} \\ \mathsf{stack} & = & \mathsf{closure\ list} \end{array}$$

term's are de Bruijn's pure lambda-terms, i.e. they have no substitution part and use de Bruijn's indices. Elements of stack are denoted by p (for the French word "pile"). Since term is the *code* and env is the *environment* the U-machine is SEC. The U-machine is defined by

$$\begin{array}{rcll} (ab, e, p) & \underset{U}{\longrightarrow} & (a, e, \langle b, e\rangle :: p) & (APP) \\ (\lambda a, e, \langle b, e'\rangle :: p) & \underset{U}{\longrightarrow} & (a, \mathsf{Lift\_env}(e) \oplus [(\langle b, e'\rangle, 0], p & (LBA - BET) \\ (\underline{1}, (c, i+1) :: e, p) & \underset{U}{\longrightarrow} & (\underline{1}, e, p) & (FVARLIFT) \\ (\underline{n+1}), (c, i+1) :: e, p) & \underset{U}{\longrightarrow} & (\underline{n}, (c, i) :: (\uparrow, 0) :: e, p) & (RVARLIFT'') \\ (\underline{1}, (\langle a, e\rangle, 0) :: e', p) & \underset{U}{\longrightarrow} & (a, e \oplus e', p) & (FVAR) \\ (\underline{n+1}, (\langle a, e\rangle, 0) :: e', p) & \underset{U}{\longrightarrow} & (\underline{n}, e', p) & (RVAR) \\ (\underline{n}, (\uparrow, 0) :: e, p) & \underset{U}{\longrightarrow} & (\underline{n+1}, e, p) & (VARSHIFT) \end{array}$$

where Lift_env is defined by

$$\begin{array}{rcl} \mathsf{Lift\_env}([\,]) & \to & [\,] \\ \mathsf{Lift\_env}((c,i) :: e) & \to & (c, i+1) :: \mathsf{Lift\_env}(e) \end{array}$$

and the operator $\oplus$ appends one environment to another one.

The U-machine stops in a state of the form $(\lambda a, e, [])$ or $(\underline{n}, [], p)$. For the computation of the strong normal form of a term we need to call recursively the U-machine. Thus we introduce two inference rules

$$\frac{(a, e, []) \xrightarrow[U]{*} (\lambda b, e', []) \quad (b, \mathsf{Lift\_env}(e')) \underset{nf}{\longrightarrow} c}{(a, e) \underset{nf}{\longrightarrow} \lambda c} \quad (L)$$

$$\frac{(a, e, []) \xrightarrow[U]{*} (\underline{n}, [], [\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle]) \quad \langle b_i, e_i\rangle \underset{nf}{\longrightarrow} c_i \text{ for } 1 \leq i \leq q}{(a, e) \underset{nf}{\longrightarrow} \underline{n} c_1 \ldots c_q} \quad (V)$$

We will prove in the next section that the strong normal form of a term a, if it exists, is reached by application of $\underset{nf}{\longrightarrow}$ until no more reduction can be applied.

7 Correction of the U-machine

In order to prove the correction of the U-machine with respect to λv-calculus, we define two translation functions τ from *environments* to *substitutions* and ξ from *states* to *terms*:

$$\begin{array}{rrcl} \tau : & \mathsf{env} & \to & \mathsf{list(Substitutions)} \\ & [\,] & \mapsto & [\,] \\ & (c,i) :: e & \mapsto & \rho((c,i)) :: \tau(e) \end{array}$$

where $\rho : (\uparrow \cup \mathsf{closure}) \times \mathsf{N} \to \mathsf{Substitutions}$

$$\begin{array}{rcll} (\langle a, e\rangle, 0) & \mapsto & a[\tau(e)]/ & (a \text{ stands for } a[]) \\ (\uparrow, 0) & \mapsto & \uparrow & \\ (c, i+1) & \mapsto & \Uparrow(\rho((c,i))) & \end{array}$$

and $\xi :$

$$\begin{array}{rcl} \mathsf{state} & \to & \mathsf{Terms} \\ (a, e, [\langle b_1, e_1\rangle; \ldots; \langle b_q, c_q\rangle]) & \mapsto & a[\tau(e)] b_1[\tau(e_1)] \ldots b_q[\tau(e_q)] \end{array}$$

We extend $\Uparrow$ to lists of substitutions by $\Uparrow([s_1; \ldots; s_k]) = [\Uparrow(s_1); \ldots; \Uparrow(s_k)]$ and we define @ as $a[e@e'] = a[e][e']$. Proofs of Lemma 10 and Lemma 11 are by induction on e.

Lemma 10 *$\tau(e \oplus e') = \tau(e)@\tau(e')$.*

Lemma 11 *$\tau(\mathsf{Lift\_env}(e)) = \Uparrow(\tau(e))$.*

Proposition 2 *For all the states (a, e, p) and (a', e', p') of the U-machine,*

$$(a, e, p) \underset{U}{\longrightarrow} (a', e', p') \Rightarrow \xi(a, e, p) \xrightarrow[\lambda v]{*} \xi(a', e', p').$$

Proof: Let p be $[\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle]$ with $q \geq 0$. We have to consider each rule of the U-machine.

- (APP) $(ab, e, p) \underset{U}{\longrightarrow} (a, e, \langle b, e\rangle :: p)$.

$$\begin{array}{rcl} \xi(ab, e, [\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle]) & = & (ab)[\tau(e)] b_1[\tau(e_1)] \ldots b_q[\tau(e_q)] \\ & \underset{\lambda v}{\longrightarrow} & a[\tau(e)] b[\tau(e)] b_1[\tau(e_1)] \ldots b_q[\tau(e_q)] \\ & = & \xi(a, e, [\langle b, e\rangle; \langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle]) \\ & = & \xi(a, e, \langle b, e\rangle :: p). \end{array}$$

- (LBA-BET) $(\lambda a, e, \langle b, e'\rangle :: p) \underset{U}{\longrightarrow} (a, \mathsf{Lift\_env}(e) \oplus [(\langle b, e'\rangle, 0)], p)$.
  - case $e = [\ ]$. $\xi(\lambda a, e, \langle b, e'\rangle :: [\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle])$

$$\begin{aligned}
&= (\lambda a)[\tau(e)]b[\tau(e')]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&= (\lambda a)b[\tau(e')]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&\underset{\lambda v}{\longrightarrow} a[b[\tau(e')]/]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&= a[\rho(\langle b, e'\rangle, 0)]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&= a[\tau([(\langle b, e'\rangle, 0)])]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&= \xi(a, [(\langle b, e'\rangle, 0)], p)\\
&= \xi(a, \mathsf{Lift_env}(e) \oplus [(\langle b, e'\rangle, 0)], p) \quad \text{since } \mathsf{Lift_env}([])=[].
\end{aligned}$$

  - case $e \neq [\ ]$.

$\xi(\lambda a, e, \langle b, e'\rangle :: [\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle])$

$$\begin{aligned}
&= (\lambda a)[\tau(e)]b[\tau(e')]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&\underset{\lambda v}{\longrightarrow} \lambda(a[\Uparrow(\tau(e))])]b[\tau(e')]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&\underset{\lambda v}{\longrightarrow} a[[\Uparrow(\tau(e))][b[\tau(e')]/]b_1[\tau(e_1)] \ldots [b_q(\tau(e_q)]\\
&= \xi(a, \mathsf{Lift_env}(e) \oplus [(\langle b, e'\rangle, 0)], [\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle])
\end{aligned}$$

  by Lemma 10 and Lemma 11.

  Remark: Since U works only on the highest part of the stack and leaves the lowest part unchanged, it is sufficient to prove

$$(a, e, []) \underset{U}{\longrightarrow} (a', e', []) \Rightarrow \xi(a, e, []) \overset{*}{\underset{\lambda v}{\longrightarrow}} \xi(a', e', [])$$

  to claim

$$(a, e, p) \underset{U}{\longrightarrow} (a', e', p) \Rightarrow \xi(a, e, p) \overset{*}{\underset{\lambda v}{\longrightarrow}} \xi(a', e', p).$$

  We use this in the following.

- (FVARLIFT) $(\underline{1}, (c, i+1) :: e, p) \underset{U}{\longrightarrow} (\underline{1}, e, p)$.

$$\begin{aligned}
\xi(\underline{1}, (c, i+1) :: e, []) &= \underline{1}[\tau((c, i+1) :: e)]\\
&= \underline{1}[\rho(c, i+1) :: \tau(e)]\\
&= \underline{1}[\rho(c, i+1)][\tau(e)]\\
&= \underline{1}[\Uparrow(\rho(c, i))][\tau(e)] \underset{\lambda v}{\longrightarrow} \underline{1}[\tau(e)]\\
&= \xi(\underline{1}, e, [])
\end{aligned}$$

- (RVARLIFT") $(\underline{n+1}, (c, i+1) :: e, p) \underset{U}{\longrightarrow} (\underline{n}, (c, i) :: (\uparrow, o) :: e, p)$.

$$\begin{aligned}
\xi(\underline{n+1}, (c, i+1) :: e, []) &= \underline{n+1}[\tau((c, i+1) :: e)]\\
&= \underline{n+1}[\rho(c, i+1) :: \tau(e)]\\
&= \underline{n+1}[\rho(c, i+1)][\tau(e)]\\
&= \underline{n+1}[\Uparrow(\rho(c, i))][\tau(e)]\\
&\underset{\lambda v}{\longrightarrow} \underline{n}[\rho(c, i)][\uparrow][\tau(e)]\\
&= \underline{n}[\rho(c, i)][\rho(\uparrow, 0)][\tau(e)]\\
&= \underline{n}[\rho(c, i) :: \rho(\uparrow, 0) :: \tau(e)]\\
&= \underline{n}[\tau((c, i) :: (\uparrow, 0) :: e)]\\
&= \xi(\underline{n}, (c, i) :: (\uparrow, 0) :: e, []).
\end{aligned}$$

- (FVAR) $(\underline{1}, (\langle a, e\rangle, 0) :: e', p) \underset{U}{\longrightarrow} (a, e \oplus e', p)$.

$$\begin{aligned}
\xi(\underline{1}, (\langle a, e\rangle, 0) :: e', []) &= \underline{1}[\tau((\langle a, e\rangle, 0) :: e')]\\
&= \underline{1}[\rho(\langle a, e\rangle, 0) :: \tau(e')]
\end{aligned}$$

$$
\begin{array}{rl}
= & \underline{1}[\rho(\langle a,e\rangle,0)][\tau(e')] \\
= & \underline{1}[a[\tau(e)]/][\tau(e')] \\
\underset{\lambda v}{\longrightarrow} & a[\tau(e)][\tau(e')] \\
= & a[\tau(e)@\tau(e')] \\
= & a[\tau(e\oplus e')] \\
= & \xi(a, e\oplus e', []).
\end{array}
$$

- (RVAR) $(\underline{n+1}, (\langle a,e\rangle,0) :: e', p) \underset{U}{\longrightarrow} (\underline{n}, e', p)$.

$$
\begin{array}{rcl}
\xi(\underline{n+1}, (\langle a,e\rangle,0) :: e', []) & = & \underline{n+1}[\tau((\langle a,e\rangle,0) :: e')] \\
& = & \underline{n+1}[\rho(\langle a,e\rangle,0) :: \tau(e')] \\
& = & \underline{n+1}[\rho(\langle a,e\rangle,0)][\tau(e')] \\
& = & \underline{n+1}[a[\tau(e)]/][\tau(e')] \\
& \underset{\lambda v}{\longrightarrow} & \underline{n}[\tau(e')] \\
& = & \xi(\underline{n}, e', []).
\end{array}
$$

- (VARSHIFT) $(\underline{n}, (\uparrow,0) :: e, p) \underset{U}{\longrightarrow} (\underline{n+1}, e, p)$.

$$
\begin{array}{rcl}
\xi(\underline{n}, (\uparrow,0) :: e, []) & = & \underline{n}[\tau((\uparrow,0) :: e)] \\
& = & \underline{n}[\rho(\uparrow,0) :: \tau(e)] \\
& = & \underline{n}[\rho(\uparrow,0)][\tau(e)] \\
& = & \underline{n}[\uparrow][\tau(e)] \\
& \underset{\lambda v}{\longrightarrow} & \underline{n+1}[\tau(e)] \\
& = & \xi(\underline{n+1}, e, []).
\end{array}
$$

□

Proposition 3 *For every terms a and b and every environment e*

$$(a, e) \underset{nf}{\longrightarrow} b \;\Rightarrow\; \xi(a, e, []) \overset{*}{\underset{\lambda v}{\longrightarrow}} b.$$

Proof: By structural induction on b.

- case $b = \lambda c$.
  By Proposition 2, we have

$$
\begin{array}{rcl}
(a, e, []) \overset{*}{\underset{U}{\longrightarrow}} (\lambda c, e', []) \Rightarrow a[\tau(e)] & \overset{*}{\underset{\lambda v}{\longrightarrow}} & (\lambda c)[\tau(e')] \\
& \underset{\lambda v}{\longrightarrow} & \lambda(c[\Uparrow(\tau(e'))]) = \lambda(c[\tau(\mathsf{Lift\_env}(e'))])
\end{array}
$$

If $(c, \mathsf{Lift\_env}(e')) \underset{nf}{\longrightarrow} d$, then $\xi(c, \mathsf{Lift\_env}(e'), []) \overset{*}{\underset{\lambda v}{\longrightarrow}} d$ by induction hypothesis. Thus

$$c[\tau(\mathsf{Lift\_env}(e'))] \overset{*}{\underset{\lambda v}{\longrightarrow}} d$$

and

$$(\lambda c)[\tau(e')] \overset{*}{\underset{\lambda v}{\longrightarrow}} \lambda d$$

, i.e. $\xi(a, e, []) \overset{*}{\underset{\lambda v}{\longrightarrow}} b$.

- case $b = \underline{n}c_1 \ldots c_q$.
  We have
  $$(a, e, []) \xrightarrow[U]{*} (\underline{n}, [], [\langle b_1, e_1\rangle; \ldots; \langle b_q, e_q\rangle])$$
  and
  $$(b_i, e_i) \xrightarrow[nf]{} c_i \text{ for } 1 \leq i \leq q.$$
  By Proposition 2 and by induction hypothesis,
  $$a[\tau(e)] \xrightarrow[\lambda v]{*} \underline{n}b_1[\tau(e_1)] \ldots b_q[\tau(e_q)] \xrightarrow[\lambda v]{*} \underline{n}c_1 \ldots c_q.$$

□

Theorem 3

1. If $(a, []) \xrightarrow[nf]{!} b$ then $a \xrightarrow[\lambda v]{!} b$ and b is the β normal form of a.

2. If a admits a β normal form b then $a \xrightarrow[\lambda v]{!} b$ and $(a, []) \xrightarrow[nf]{!} b$.

Proof:

- By Proposition 3, $(a, []) \xrightarrow[nf]{!} b \Rightarrow a \xrightarrow[\lambda v]{*} b$ since $a([\tau(e)]) = a([]) = a$. Let us prove that b is actually the λv-normal form of a. If a is $\underline{n}$ then $(\underline{n}, []) \xrightarrow[nf]{!} \underline{n}$ and $\underline{n}$ is its own λv normal form. If b is of the form λc or of the form $\underline{n}c_1 \ldots c_q$ we can prove by induction on b, that b is the λv normal form of a.
  By Proposition 1, we have $\mu(a) \xrightarrow[\beta]{*} \mu(b)$. a is a pure λ-term, thus $\mu(a) = a$. b is irreducible by λv, therefore it has no closure and no redex, hence $\mu(b) = b$ and b is irreducible by β. Therefore $a \xrightarrow[\beta]{!} b$.
- Assume a admits a β normal form b. Then $a \xrightarrow[\lambda v]{*} b$ because $a \xrightarrow[\beta]{} b$ means $a \xrightarrow[B]{} b'$ and $b = v(b')$. Moreover a pure λ-term which is irreducible by $\xrightarrow[\beta]{}$ is also irreducible by $\xrightarrow[\lambda v]{}$, thus $a \xrightarrow[\lambda v]{!} b$.
  If (a,[]) $\xrightarrow[nf]{!}$ c then the first part of the theorem and the confluence of the λv-calculus imply $c = b$. We just have to prove that $\xrightarrow[nf]{}$ terminates in order to achieve the proof of the second part of the theorem. The proof of termination of $\xrightarrow[nf]{}$ is similar to the proof of termination of the KN-machine in [3, 4]. By composition of ξ and μ where μ is the function defined in section 2, we have a translation function γ of a state of the U-machine into a pure lambda term.
  Let $(a, e, p) \xrightarrow[U]{} (a', e', p')$.
  - If $\xrightarrow[U]{}$ is not the rule $(LBA - BET)$ then
    - $\xi(a, e, p) \xrightarrow[v]{} \xi(a', e', p')$
    - and $\gamma(a, e, p) = \gamma(a', e', p')$.

    Since the system v is noetherian, the set of rules of the U-machine without $(LBA - BET)$ is also noetherian.
  - If $\xrightarrow[U]{}$ is $(LBA - BET)$ then $\gamma(a, e, p) \xrightarrow[\beta]{} \gamma(a', e', p')$. We can verify that the rules of the U-machine imply that we always reduce the leftmost outermost redex of $\gamma(a, e, p)$. This strategy is known to be terminating, hence we can apply $(LBA - BET)$ only finitely many times.
  - The application of $\xrightarrow[nf]{}$ builds a part of the β normal form of the starting λ-term, if it exists. Thus we have only a finite number of $\xrightarrow[nf]{}$ steps.

Thus the noetherianity of υ and the safety of the leftmost outermost strategy reduction in the λ-calculus yield the second part of the theorem.

$\square$

8 Conclusion

Explicit substitutions are a main tool for describing lazy evaluation of functional programming languages, but the system $\lambda\upsilon$ has many other useful applications. For instance, as a first order system it allows for the application of methods based on or derived from narrowing to higher-order unification. $\lambda\upsilon$ describes only the substitution mechanism and makes no hypothesis on implementation choices like composing substitutions or representing compound substitutions by lists. Therefore it may be a tool to explore several implementations of λ-calculus taking advantage of sharing or parallelism. If implementation choices have to be made, the will be done at the right time and necessary properties will be proved then.

Acknowledgment. We would like to thank Prof. N.G. de Bruijn for an interesting discussion and for mentioning us his papers, Georges Gonthier and Luc Maranget for suggestions improving proof of Theorem 2, Roberto Amadio and Paul Zimmermann for interesting discussions and Pierre-Louis Curien for encouragements.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] H. P. Barendregt. *The Lambda-Calculus, its syntax and semantics*. Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1984. Second edition.

[3] P. Crégut. An abstract machine for the normalization of λ-calculus. In *Proc. Conf. on Lisp and Functional Programming*, pages 333–340. ACM, 1990.

[4] P. Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université de PARIS 07, 1991.

[5] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[6] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhäuser, 1993. 2nd edition.

[7] P.-L. Curien, Th. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. RR 1617, INRIA, Rocquencourt, February 1992.

[8] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Koninkl. Nederl. Akademie van Wetenschappen*, 75(5):381–392, 1972.

[9] N. G. de Bruijn. A namefree lambda calculus with facilities for internal definition of expressions and segments. TH-Report 78-WSK-03, Technological University Eindhoven, Netherlands, Department of Mathematics, 1978.

[10] J. Field. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proceedings of the 17th Annual ACM Symposium on Principles Of Programming Languages, Orlando (Fla., USA)*, pages 1–15, San Fransisco, 1990. ACM.

[11] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[12] T. Hardin. Eta-conversion for the languages of explicit substitutions. In H. Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, volume 632 of *Lecture Notes in Computer Science*, pages 306–321. Springer-Verlag, September 1992.

[13] Th. Hardin. Confluence results for the pure strong categorical combinatory logic CCL: λ-calculi as subsystems of CCL. *Theoretical Computer Science*, 65:291–342, 1989.

[14] Th. Hardin and J.-J. Lévy. A confluent calculus of substitutions. In *France-Japan Artificial Intelligence and Computer Science Symposium*, Izu, 1989.

[15] P. Lescanne. Termination of rewrite systems by elementary interpretations. In Hélène Kirchner and G. Levi, editors, *Proceedings 3rd International Conference on Algebraic and Logic Programming, Volterra (Italy)*, volume 632 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, September 1992.

[16] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In Hans Boehm, editor, *Proceedings of the 21st Annual ACM Symposium on Principles Of Programming Languages, Portland (Or., USA)*, pages 60–69. ACM, 1994.

[17] N. Martì-Oliet and J. Meseguer. Rewriting logic as a logical and semantical framework. Technical report, SRI International, May 1993.

[18] A. Ríos. *Contributions à l'étude des λ-calculs avec des substitutions explicites.* Thèse de Doctorat d'Université, U. Paris VII, 1993.

[19] E. Ritter. Normalization for typed lambda calculi with explicit substitution. In *Conference on Symbolic Logic*, 1993. Also available as University of Cambridge, Computer Laboratory, Technical Report.

[20] T. Strahm. Partial applicative theories and explicit substitutions. Technical Report IAM 93–008, Univerität Bern, Institut für Informatik und angewandte Mathematik, June 1993.

An ML implementation of the U-machine

The following programs are an extremely naive and unefficient implementation in ML of the U-machine. None of the possible optimizations has been made like sharing of environments or lazy function for `Lift_env` or `append_env`. Here we have chosen Caml-light.

```
type term = L of term | App of term * term | V of nat
and nat = I | S of nat ;;

type env = Empty_env | Cons of (item * int) * env
and item = Shift | Cl of term * env
and closure = C of term * env ;;

type stack == closure list ;;
```

```
type state == term * env * stack ;;

let rec Lift_env = function
      Empty_env → Empty_env
    | Cons((c,i), e) → Cons((c,i+1), Lift_env(e)) ;;

let rec append_env = function
      (Empty_env,e) → e
    | (Cons(it,e), f) → Cons(it, append_env (e,f)) ;;

let rec machine = function

      (App(a,b), e, p) → machine (a, e, C(b,e) :: p)

    | (L(a), e, C(b,f) :: p) →
            machine(a, append_env(Lift_env(e), Cons((Cl(b,f), 0), Empty_env)), p)

    | (V(I), Cons ((Cl(a,e),0), f), p) → machine(a, append_env(e,f), p)

    | (V(S(n)), Cons ((Cl(a,e),0), f), p) → machine(V(n), f,p)

    | (V(n), Cons((Shift,0), e),p) → machine(V(S(n)), e, p)

    | (V(I), Cons((c,i), e), p) → machine(V(I), e,p)

    | (V(S(n)), Cons((c,i), e), p) → machine(V(n), Cons((c,i-1), Cons((Shift,0), e)), p)

    | x → x ;;

let rec nf (a, e) = match machine (a, e, [ ]) with

      (L(a), e, [ ]) → L (nf (a, Lift_env(e)))

    | (V(n), Empty_env, p) → let rec map_nf = function
                                   [ ] → V(n)
                                 | C(b,f) :: p → App(map_nf(p), nf(b,f))
                             in map_nf(rev(p))

    | _ → failwith "not_possible" (* this case cannot be a result of machine *) ;;
```