ƛ

Dimitri Hendriks and Vincent van Oostrom

Department of Philosophy, Universiteit Utrecht
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
{hendriks,oostrom}@phil.uu.nl

Abstract. We make the notion of scope in the λ-calculus explicit. To that end, the syntax of the λ-calculus is extended with an end-of-scope operator ƛ, matching the usual opening of a scope due to λ. Accordingly, β-reduction is extended to the set of scoped λ-terms by performing *minimal* scope extrusion before performing replication as usual. We show confluence of the resulting scoped β-reduction. Confluence of β-reduction for the ordinary λ-calculus is obtained as a corollary, by extruding scopes *maximally* before forgetting them altogether. Only in this final forgetful step, α-equivalence is needed. All our proofs have been verified in Coq.

1 Introduction

Performing a substitution $M[x{:=}N]$ in the λ-calculus can be decomposed into two subtasks: replicating N an appropriate number of times, and renaming in M in order to prevent unintended capture of variables of N. Indeed, the defining clauses of Curry's definition of substitution ([3, C.1 DEFINITION]) can be neatly partitioned into those dealing with replication (the variable and application clauses) and those dealing with renaming (the abstraction clauses). In this paper we will focus on trying to understand the latter subtask. We do so, by extending λ-calculus with an explicit operator representing the (end of the) scope of a name, while leaving replication implicit.

In the λ-calculus the scope of the binder λx in $\lambda x.M$ is (implicitly) assumed to extend to the whole of M. Hence to make the notion of scope explicit, it suffices to introduce an operator expressing the end of the scope of λx. This operator is denoted by ƛ (adbmal). ƛ$x.M$ expresses that the scope of x is ended 'above' M. For instance, in the ƛ-term $\lambda x.$ƛ$x.\underline{x}$ the underlined occurrence of the variable x is free, since the binding effect of the λx is undone by the subsequent ƛx. For another example, only the underlined occurrence of x is free in $\lambda x.x(\text{ƛ}x.\underline{x})x$; the first and third occurrences of x are in scope of the λx (see Figure 1).

Definition 1. *The set* $(M, N, P \in)\Lambda$ *of* ƛ-terms *is defined by:*

$$\Lambda ::= \mathcal{V} \mid \text{ƛ}x.\Lambda \mid \lambda x.\Lambda \mid \Lambda\Lambda$$

where $(x, y, z \in)\mathcal{V}$ is a collection of variable(name)s with decidable equality:

Axiom 1 (Names with decidable equality) $x = y \vee x \neq y$, *for all* $x, y : \mathcal{V}$

F. Baader (Ed.): CADE-19, LNAI 2741, pp. 136–150, 2003.

We adopt the usual notational conventions for the λ-calculus [3], treating ƛ analogously to λ. λ-terms are embedded as ƛ-terms without occurrences of ƛ.

In order to extend the notions of α-equivalence and β-reduction, we should first try to make some semantic sense of ƛs. Thinking of λx and λx as (named) opening '$[_x$' and closing '$]_x$' brackets[1], it is clear that ƛ-terms may come in different degrees of balancedness. For instance, scopes could seemingly be crossing one another as indicated by the boxes in:

$$P = \boxed{\lambda x.\boxed{\lambda y.\lambda x.}\lambda y.}\, Q$$

This would obviously cause semantical problems (try to define substitution). To overcome this problem we assume a simple minded jump semantics: an occurrence of $\lambda x.M$ implicitly ends the scopes of all (non-matching) λs inbetween that occurrence and its matching λx, just as the occurrence of the variable x in $\lambda x.\lambda y.x$ can be thought of as implicitly ending the scope of the λy. Hence P is semantically equivalent to $\lambda x.\lambda y.\lambda y.\lambda x.\lambda y.Q$. Our definitions of α-equivalence and β-reduction and hence our definition of substitution, as will be presented below, are meant to reflect this intuitive (operational) semantics.

Apart from such *jump* terms we identify the useful subclasses of scope-balanced and balanced terms, both of which are closed under α-equivalence and β-reduction. Balanced terms can be used to represent nameless λ-terms using de Bruijn-indices, by using only a single name. Ordinary λ-terms are not (necessarily) balanced, however they always are scope-balanced.

Definition 2. *A term M is* scope-balanced *if $\langle\Box\rangle M$, where $\langle X\rangle M$ is defined by:*

$$\frac{}{\langle X\rangle x} \qquad \frac{\langle Xx\rangle M}{\langle X\rangle \lambda x.M} \qquad \frac{\langle X\rangle M}{\langle Xx\rangle \lambda x.M} \qquad \frac{\langle X\rangle M \quad \langle X\rangle N}{\langle X\rangle MN}$$

Balancedness *is defined as scope-balancedness restricting the first clause to*

$$\frac{}{\langle Xx\rangle x}$$

Here $\Box$ is the empty stack and Xx is the result of pushing x on the stack X.

For instance, $\lambda x.x$ is not scope-balanced (no λ to match ƛ), $\lambda x.y$ is scope balanced but not balanced (x not closed before y), and $\lambda x.x$ is balanced. Scopes in balanced ƛ-terms can be neatly visualized as boxes in their abstract syntax tree, as shown in Figure 1 [2]. Vice versa, in the term representation of a box, only its 'doors' are kept. That is, λs and ƛs are used to demarcate all places where the boundary of the box is crossed by the abstract syntax tree. In fact, there is a strong similarity (see Figure 1) between balanced terms and the context-free string language of *matching brackets* as presented by the grammar:

$$P ::= \epsilon \mid [P] \mid PP$$

[1] But note that brackets (parentheses) usually apply 'horizontally' to the textual representation of terms, whereas λ and ƛ apply 'vertically' to their abstract syntax trees (where brackets do not even occur).

[2] Scopes in non-balanced terms can be drawn as floorless boxes ($\lambda x.\lambda y.x$ in Figure 1).

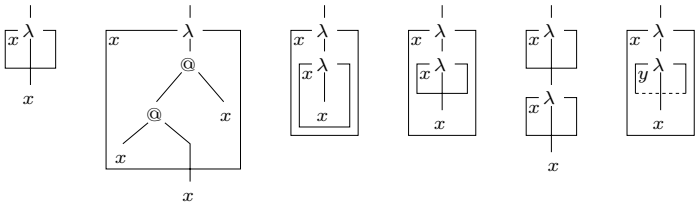

Fig. 1. $\lambda x.\lambda x.x$, $\lambda x.x(\lambda x.x)x$, $\lambda x.\lambda x.x$, $\lambda x.\lambda x.\lambda x.x$, $\lambda x.\lambda x.\lambda x.\lambda x.x$, and $\lambda x.\lambda y.x$

- Scopes can be nested (similar to $[P]$). In the λ-term $\lambda x.\lambda x.x$, the occurrence of x is implicitly assumed to be bound by the rightmost λx. Similarly, the scope of the rightmost λx is ended by the λx in $\lambda x.\lambda x.\lambda x.x$.
- Scopes can be concatenated (similar to PP). In the λ-term $\lambda x.\lambda x.\lambda x.\lambda x.x$, the scopes of the two λxs do not have overlap/are not nested, in spite of the latter being 'to the right' of the former.

Indeed, the set of balanced λ-terms can be generated by a so-called context-free term grammar, where contect-free term grammars are the natural generalization of context-free string grammars (see e.g. [12, Section 2.5]). A difference between matching bracket strings and balanced λ-terms is that, due to the branching structure of terms, *several* λ's may match the same λ as in $\lambda x.(\lambda x.\underline{x})(\lambda x.\underline{x})$, with both underlined occurrences of x free.

The outline of the rest of this paper is as follows. We provide several definitions of α-equivalence for λ-terms in Section 2, extending classical definitions as found in the literature on the λ-calculus, prove them to be decidable congruence relations, and show them to be equivalent. Then we present a definition of β-reduction for λ-terms in Section 3, extending the usual definition for the λ-calculus, and prove this notion of reduction to be confluent *without* α-equivalence. In both (α and β) cases it is shown how the results on the λ-calculus entail the corresponding results for the ordinary λ-calculus, e.g. confluence of β-reduction modulo α-equivalence. Applications are presented in Section 4.

The results in this paper pertain to scope-balanced and, hence, balanced terms. Coq proofs are available at `http://preprints.phil.uu.nl/lgpr/`. Although the definition of substitution pertains to the jump calculus as well, proving confluence seems to require a more general form of the substitution lemmata (which do however hold in their present form), based on 'subtracting' stacks, but at the moment of writing proofs were not finished yet.

Related Work. When application of λx is restricted to variables (and end-of-scopes), it corresponds to Berkling's lambda-bar ([5]), which is in turn seen to be a named version of de successor operator in De Bruijn's nameless (more precisely: single name) calculus ([14]). Their calculi do not allow successions of boxes, only nestings of boxes. This corresponds to the sublanguage of the language of matching brackets (see above) generated by the grammar: $B ::= \epsilon \mid [B]$.

Restricting to a single name, i.e. to De Bruijn-indices, λx corresponds to the shift substitution $[\uparrow]$ in the λ-calculus with explicit substitutions $\lambda\sigma$ of [1], or the shift operation *Shi* of [10], and Bird and Paterson show in [6] that in the

balanced (single name) case the term language of the ƛ-calculus is context-free by presenting it by means of the following context-free term grammar:

$$Term\ a ::= Var\ a \mid App(Term\ a, Term\ a) \mid Abs(Term(Incr(Term\ a)))$$
$$Incr\ a ::= Zero \mid Succ\ a$$

the idea being that *Term*s are balanced by generating *Incr*s, i.e. variables (*Zero*s) or end-of-scopes (*Succ*s), at the same time as their matching *Abs* (abstraction)[3].

When restricting to the balanced case, our boxes correspond closely to boxes in MELL proof nets for linear logic (see e.g. [20]). In fact, in our optimal implementation (see Section 4) λx disintegrates into a λ (a par in MELL) and (part of the boundary of) an x-box ((Asperti's version of) a box in MELL), upon encountering an application. One can think of these two phases of abstraction as turning a free variable x into a bound one by closing it off from the outside world inside an x-box, but providing a handle to x to the outside world again in the form of the λ. Many proposals for decomposing abstraction into more elementary notions can be found in the literature, a recent one being [4]. Similarly, notions of enclosure abound. Analogous to the conflation of the enclosure with the enclosed as found in (the etymology of) words such as town, garden, park and paradise, these formalisations may or may not make the boundary explicit (see e.g. [9,23,7]).

In the area of dynamic semantics for natural language, a stack-based semantics for a variant of predicate logic is presented in [17]. Although, the exact relationship is not clear to us yet, a difference seems to be that in their semantics every variable has its own stack, whereas we have a single stack. However, also in [5] variables have their own stack.

2 α

We present three distinct definitions of α-equivalence for the λ-calculus known from the literature, in historical order. We then compare these notions, present our adaptations of each of them to the ƛ-calculus, and prove them to be equivalent. For this the existence of fresh variables is required.

Axiom 2 (Fresh name) $\forall X : list(\mathcal{V}).\exists x{:}\mathcal{V}.x \notin X$,

2.1 $\lambda\alpha$

Church. Our first notion of α-equivalence is the usual one based on Church's Postulate I for the λ-calculus [11], which reads (page 355):

> *If* J *is true, if* L *is well-formed, if all the occurrences of the variable* x *in* L *are occurrences as a bound variable, and if the variable* y *does not occur in* L*, then* K*, the result of substituting* $\mathsf{S}^{\mathsf{x}}_{\mathsf{y}}\,\mathsf{L}|$ *for a particular occurrence of* L *in* J*, is also true.*

where $\mathsf{S}^{\mathsf{X}}_{\mathsf{Y}}\,\mathsf{U}|$ *represents the formula which results when we operate on the formula* U *by replacing* X *by* Y *throughout, where* Y *may be any symbol or formula but* X *must be a single symbol, not a combination of symbols* [11, page 350].

[3] This does not work (directly) for non-balanced terms in the many-variable case.

Due to Curry, Postulate I is nowadays known as the α-conversion rule. An α-conversion *step* is obtained from the α-conversion rule by allowing its application to any subterm of a term. An α-*conversion* consists of a sequence of α-conversion steps. Finally, a term is said to be α-equivalent to another one, if there exists an α-conversion from the former to the latter.

An advantage of this definition is that it is operational and fine-grained; each α-conversion step itself is easy to understand since it does only little work. A disadvantage of this fine-grainedness is that it is at first sight not clear whether structural properties such as symmetry and decidability of α-conversion hold. Moreover, it needs the Fresh name axiom due to the *Extra-hand principle*: if both your hands are full, you need a third hand in order to swap their contents[4].

Example 1. The terms $\lambda x.\lambda y.xy$ and $\lambda y.\lambda x.yx$ are α-equivalent. However, both α-conversion steps replacing x by y and vice versa are forbidden. Hence, an α-conversion needs to introduce a third, fresh, variable, say z, first:

$$\lambda x.\lambda y.xy \to_\alpha \lambda z.\lambda y.zy \to_\alpha \lambda z.\lambda x.zx \to_\alpha \lambda y.\lambda x.yx$$

Schroer. In order to prove symmetry and decidability of α-equivalence as defined in the previous paragraph, one may try to find a strategy for α-conversion such that the number of α-conversion steps needed in a conversion from s to t is bounded by, say, the sum of the sizes of s and t. An obvious way to bound this number is by restricting α-conversion by:

Never rename twice.

However, from Example 1 we immediately see that this is too strict a restriction; the leftmost λ-abstraction needs to be renamed twice. Hence renaming once is not enough, but, as the example suggests our assumption may be replaced by:

Never rename thrice.

Such an idea appears at least as early as Schoer's PhD thesis [24, page 384]:

> Scholium 3.44. The proof of Theorem 3.44 below has as its germ the following procedure to determine of A,B ε Wocc whether or not A adj B: Let Z_1, Z_2,... be singleton expressions of the alphabetically earliest variables not occurring at all in either of A,B , enumerated without repetitions. In each of A,B , change quantifiers from left to right, replacing the given variables by the Z's in order. There will result A',B' such that A adj A' . B adj B' , and such that A adj B . $\equiv$. A' = B' .

where `adj` is his notion of α-equivalence and `Theorem 3.44` states decidability.

[4] There's the well-known way to swap the contents of two registers *in situ* by performing three exclusive-or's (xor); in Java: `r1 ^= r2; r2 ^= r1; r1 ^= r2` where `op1 ^= op2` is equivalent to `op1 = op1 ^ op2` and `^` is bitwise xor. Here, we will *not* assume the structure needed for this, e.g. a Boolean ring on the variables.

Example 2. Applied to Example 1 Schroer's procedure yields:

$$\lambda x.\lambda y.xy \to_\alpha \lambda z_1.\lambda y.z_1 y \to_\alpha \lambda z_1.\lambda z_2.z_1 z_2 \leftarrow_\alpha \lambda z_1.\lambda x.z_1 x \leftarrow_\alpha \lambda y.\lambda x.yx$$

Of course, to prove that this is an α-conversion one needs to prove that the last two backward α-steps are forward α-steps as well; they are.

Symmetry of a definition based on Schroer's procedure is trivial, decidability and reflexivity are also not too difficult, but now transitivity is not so simple because of the choosing of the `alphabetically earliest variables not occurring at all in either of A,B` which may differ for `A,B` and `B,C`, when proving `A adj C` [5]. Also note that the procedure is *not* very parsimonious; it allocates as many fresh variables as there are λ-abstractions (quantifiers) in a term, where a single one (one extra hand) would suffice. This latter fact may be seen by proceeding in a top-down fashion, the only interesting case being abstraction.

Kahrs. Both the problem of showing transitivity and the need for the Fresh name axiom can be overcome by making renaming implicit. That is, instead of explicitly relating terms by explicitly renaming variables, one may set up an (implicit) correspondence between their respective variables. For instance, the two terms in Example 1 are shown α-equivalent by letting x and y in the first correspond to y and x in the second. However, the correspondence needs more structure than just a bijection between the sets of variables in both terms.

Example 3. The terms $\lambda x.x\lambda y.y$ and $\lambda x.x\lambda x.x$ are α-equivalent, but this cannot be shown by means of a bijection between variables.

To define α-equivalence inductively, one has to set up a correspondence between *stacks* of variables. Such an idea appears in Kahrs' paper [18]; to quote from it:

> We also define a notion of α-congruence for our terms. It is the usual one, but we shall use it in a slightly more general setting, based on proof rules.
> **Definition 11**. *Sentences* are of the form $\Gamma \vdash t \equiv u$ or $\Gamma \vdash x = y$, where x and y are variables, t and u are terms of the same type and arity, and Γ is an *environment*. An environment is a list $x_1 = y_1, \cdots, x_n = y_n$ of equations between variables. We write ϵ for the empty environment ($n = 0$). A sentence *holds*, if it can be derived by the proof rules in figure 2. (see Figure 2)

One easily proves by induction that α-congruence defined in this way, has all the desired structural properties, e.g. transitivity and decidability. But, of course, it is less clear how to decompose α-equivalence into 'atomic' renaming steps.

2.2 ƛα

We show that each of the three definitions of α-equivalence can be straightforwardly extended from λ-terms to ƛ-terms. In each case, we highlight the key aspect of our formalisation in Coq.

[5] Compared to Church's α-conversion Schroer's procedure needs variables to be alphabetically sorted. Here, we will *not* assume the structure needed for this (e.g. a well-order) on the collection of variables.

$$\frac{}{\epsilon \vdash x = x} \qquad \frac{}{\Gamma, x = y \vdash x = y} \qquad \frac{v \neq x \quad y \neq z \quad \Gamma \vdash v = z}{\Gamma, x = y \vdash v = z}$$

$$\frac{x, y \in \mathit{Var} \quad \Gamma \vdash x = y}{\Gamma \vdash x \equiv y} \qquad \frac{F \in \mathit{Sym}}{\Gamma \vdash F \equiv F} \qquad \frac{\Gamma, x = y \vdash t \equiv u}{\Gamma \vdash [x]t \equiv [y]u} \qquad \frac{\Gamma \vdash A \equiv C \quad \Gamma \vdash B \equiv D}{\Gamma \vdash AB \equiv CD}$$

Fig. 2. Proof rules for α-congruence (Kahrs [18])

Church. We have defined α-conversion `alpha_conv_hat` as:

```
Definition church := (Rhat alpha_conv).
```

where `Rhat` yields the reflexive, symmetric, and transitive closure of its argument `alpha_conv`, which is (inductively defined) single-step α-renaming. All the work in it is performed by the clause for abstraction, which reads:

```
| alpha_conv_rule : (M:sterm;x,y:name)
    ~(In y (names M))->(alpha_conv (abs x M)(abs y (rename M x y Nil)))
```

which should be self-explanatory. The clause dealing with λ is just a compatibility clause ([3, 3.1.1. Definition]), since at the time one comes across an λ, all the work has already been performed by its matching abstraction.

Schroer. Our definition of α-equivalence à la Schoer (`schroer`):

```
Definition schroer := [M,N:sterm](EX Z:(list name)|(schroer' M N Z)).
```

makes use of an auxiliary stack `Z` which records the variables chosen thusfar for renaming. `schroer'` is inductively defined, and again all the work is performed in the clause for abstraction. Compared to α-conversion above, the variable chosen for renaming is now much fresher: not only must it be fresh for `M`, but also for `N` and for the variables `Z` chosen thusfar:

```
| schroer_rule : (M,N:sterm;x,y,z:name;Z:(list name))
                ~(In z (names M)) ->~(In z (names N)) ->~(In z Z)
                ->(schroer' (rename M x z Nil)(rename N y z Nil) Z)
                  ->(schroer' (abs x M)(abs y N)(cons z Z))
```

The clause dealing with λ is just a compatibility clause, as above.

Kahrs. Our definition of α-equivalence à la Kahrs (`kahrs`) reads:

```
Definition kahrs := [M,N:sterm](kahrs' M Nil N Nil).
```

and makes use of two auxiliary stacks (both initially empty (`Nil`)), to set up the correspondence between the variables in `M` and `N` mentioned above. `kahrs'` just implements the clauses of Figure 2, extended with clauses for λ, which are analogous to the clauses for variables, and are displayed in Figure 3.

$$\frac{\epsilon \vdash t \equiv u}{\epsilon \vdash ʎx.t \equiv ʎx.u} \qquad \frac{\Gamma \vdash t \equiv u}{\Gamma, x = y \vdash ʎx.t \equiv ʎy.u} \qquad \frac{v \neq x \quad y \neq z \quad \Gamma \vdash ʎv.t \equiv ʎz.u}{\Gamma, x = y \vdash ʎv.t \equiv ʎz.u}$$

Fig. 3. Proof rules for α-congruence of ʎ in Kahrs' notation

Results on α-Equivalences.

Theorem 1. *All three notions of α-equivalence are equivalent.*

Note that to prove that ʎ-terms which are α-equivalent à la Kahrs are α-equivalent according to the other two definitions, one essentially uses the Fresh name axiom. (It is not needed in the other direction.)

Theorem 2. *α-equivalence is a congruent equivalence relation.*

Proof. Taking the inductive definition of Kahrs, the results are all proven by straightforward inductions on the definition, loading them appropriately with stacks. For instance, to prove that the relation is a congruence, one needs to show that inserting the same variable on the bottom of both stacks is irrelevant: $\Gamma \vdash A \equiv B$ iff $z = z, \Gamma \vdash A \equiv B$.

Lemma 1. *α-equivalence preserves ʎ-terms, scope-balancedness, balancedness, and λ-terms.*

Preservation of λ-terms implies that also for the ordinary λ-calculus, the three notions of α-equivalence are equivalent (in the way we have formalised them), yielding as far as we know the first formal such results, e.g. of transitivity and decidability (only assuming decidability of equality of names).

During proof development, (the generalization of) Kahrs' definition was by far the easiest to work with, because of it being defined inductively. Note that his definition 'works' directly for the infinitary ʎ-calculus as well (defined, say, analogously to [25, Chapter 12]).

3 β

We extend β-reduction to ʎ-terms, and show it to be confluent without renaming. Confluence of β-reduction modulo α-equivalence is obtained as a corollary, by defining suitable projections and liftings of their respective reductions.

3.1 $\lambda\beta$

In [3, Chapter 3], the binary relation $\to_\beta$ on Λ is defined as the compatible closure of the notion of reduction $\beta = \{((\lambda x.M)N, M[x{:=}N]) | M, N \in \Lambda\}$. The substitution $M[x{:=}N]$ in the rhs of β is the naive one, i.e. up to α-congruence which is denoted by $\equiv_\alpha$. The naive approach is in turn justified by showing α-congruence to be a congruence for Curry's definition of substitution:

Let M, $N \in \Lambda$. Then $M[x{:=}N]$ is defined inductively as follows (even if the variable convention is not observed).

M	$M[x{:=}N]$
x	N
$y \not\equiv x$	y
$M_1 M_2$	$M_1[x{:=}N]M_2[x{:=}N]$
$\lambda x.M_1$	$\lambda x.M_1$
$\lambda y.M_1$, $y \not\equiv x$	$\lambda z.M_1[y{:=}z][x{:=}N]$
	where $z \equiv y$ if $x \notin \mathrm{FV}(M_1)$ or $y \notin \mathrm{FV}(N)$,
	else z is the first variable in the sequence
	v_0, v_1, v_2, ... not in M_1 or N.

Our notion of substitution on Λ differs from Curry's in several ways[6].

The first difference is 'under the hood'. Curry's definition is *not* an inductive one (to Coq) because of its final clause. Instead, we base our inductive definition on the skeleton $\mathtt{skl}(M)$, which is obtained from the term by forgetting names.

The second difference is more important and serves to 'make α-congruence explicit'. The point is that the last clause in Curry's definition of substitution is neither perspicuous nor technically convenient. On the one hand, it encodes several cases at once relying on the 'coding trick' that $M[y{:=}y]$ equals M, in case $x \notin \mathrm{FV}(M_1)$ or $y \notin \mathrm{FV}(N)$. On the other hand, renaming of bound variables is not incorporated in a modular way. Our definition addresses both issues by performing renaming first on $\lambda y.M_1$ in case there's the threat of confusion of variables. The definition is inductive (to Coq) if one decrees 'threat of confusion of variables' larger than 'no confusion'.

Definition 3. *Substitution on λ-terms is defined as above, except for the clauses of λ-abstraction, which are to be replaced by:*

$\lambda y.M_1$	$\lambda y.M_1[x{:=}N]$
	if $x \neq y$ and $y \notin FV(N)$
$\lambda y.M_1$	$(\lambda z.M_1[y{:=}z])[x{:=}N]$
	otherwise, with z obtained via an α-step from $\lambda y.M_1$
	such that $x \neq z$ and $z \notin FV(N)$

Despite the apparent differences, this definition is seen (proven) to be more liberal than Curry's (it does not need the variables to be linearly ordered).

3.2 $\lambda\!\!\!\!\!\!\backslash\beta$

We present the definition of β-reduction and the salient points of its proof of confluence. Compared to the ordinary λ-calculus, the β-rule must now take care of an arbitrary number of $\lambda\!\!\!\!\!\!\backslash$s which are 'inbetween' the application and the abstraction. In such cases, the scopes of the $\lambda\!\!\!\!\!\!\backslash$s are *extruded* in a minimal way so as to contain the scope of the abstraction, after which β-reduction proceeds as usual (see Figure 4, where it is irrelevant where scopes are in N). In order to

[6] Apart from that we do not assume variables to be ordered, as mentioned above.

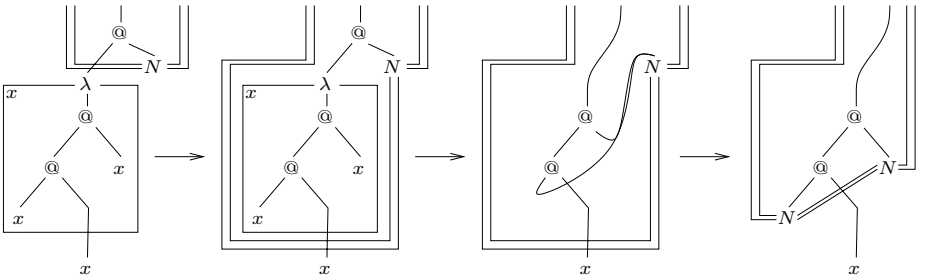

Fig. 4. β-reduction: scope extrusion, rewiring and x-box removal, and replication

perform all these operations in one go, our notion of substitution as employed by β-reduction has three arguments, of which the second one is as usual.

Definition 4. *The β-rule is $(\lambda\!\!\!\!\backslash X.\lambda x.M)N \to M[X, x{:=}N, \Box]$. The relation $\to_\beta$ is the compatible closure of the β-rule.*

The third argument of substitution, which initially is the empty stack, serves to determine whether an occurrence of x in M matches with the x to be substituted for. In particular, during substitution, this stack is pushed upon when encountering an abstraction, and popped from when meeting an end-of-scope:

Definition 5. Substitution $M[X, x{:=}N, Y]$ *is defined by:*

$$
\begin{aligned}
y[X, x{:=}N, Y] &= y, \text{ if } y \in Y \\
y[X, x{:=}N, Y] &= \lambda\!\!\!\!\backslash Y.N, \text{ if } y \notin Y,\ x = y \\
y[X, x{:=}N, Y] &= \lambda\!\!\!\!\backslash Y.\lambda\!\!\!\!\backslash X.y, \text{ if } y \notin Y,\ x \neq y \\
(\lambda y.M)[X, x{:=}N, Y] &= \lambda y.M[X, x{:=}N, yY] \\
(\lambda\!\!\!\!\backslash y.M)[X, x{:=}N, YyY'] &= \lambda\!\!\!\!\backslash y.M[X, x{:=}N, Y'], \text{ if } y \notin Y \\
(\lambda\!\!\!\!\backslash y.M)[X, x{:=}N, Y] &= \lambda\!\!\!\!\backslash Y.\lambda\!\!\!\!\backslash X.M, \text{ if } y \notin Y,\ x = y \qquad (6) \\
(\lambda\!\!\!\!\backslash y.M)[X, x{:=}N, Y] &= \lambda\!\!\!\!\backslash Y.\lambda\!\!\!\!\backslash X.\lambda\!\!\!\!\backslash y.M, \text{ if } y \notin Y,\ x \neq y \qquad (7) \\
(M_1 M_2)[X, x{:=}N, Y] &= M_1[X, x{:=}N, Y]M_2[X, x{:=}N, Y]
\end{aligned}
$$

The important clauses are the sixth and seventh, which explain the end-of-scope. Basically they say that if we have reached an end-of-scope, which matches (6) or jumps (7) the variable (x) to be subtituted for, then we can just throw the argument (N) away; this is safe since we know that x does not occur free in M.

Theorem 3. *$\to_\beta$ is confluent on $\lambda\!\!\!\!\backslash$.*

Proof. Our proof strategy is the usual Tait and Martin-Löf proof ([3]), hence is essentially based on the so-called substitution lemma on page 27 of [3]:

2.1.16. Substitution Lemma. If $x \not\equiv y$ and $x \notin \mathrm{FV}(L)$, then

$$M[\,x{:=}N][\,y{:=}L] \equiv M[\,y{:=}L]\Big[\,x{:=}N[\,y{:=}L]\Big]$$

which arises when computing the critical pair for the λ-term $(\lambda y.(\lambda x.M)N)L$. Interestingly, the substitution lemma now splits into two lemmata, depending on whether the scope of y is ended by some $\curlywedge y$ just in front of the λx, or not. We will comment on this below. Otherwise, the proof is entirely standard, (inductively) introducing multi-steps and proving that multi-steps have the diamond property.

What is interesting to note is that *no* α-conversion is needed. One might say that this is no surprise, since explicitly dealing with end-of-scopes constitutes a renaming mechanism in itself. Still, it *is* in our opinion surprising that the minimal scope-extrusion mechanism works nicely on non-balanced terms.

Let us now present our two versions of the substitution lemma. The *closed* substitution lemma arises when the scope of y *is* ended by some end-of-scope in front of the λx, e.g. in $(\lambda y.(\curlywedge y.\lambda x.M)N)L$,

Lemma 2 (Closed SL).

$$M[Y'yZ', x{:=}N, X'][Y, y{:=}P, X'Y'] = M[Y'YZ', x{:=}N[Y, y{:=}P, Y'], X']$$

Note that the substitution for y in M has disappeared from the rhs, corresponding to the erasing effect of the $\curlywedge y$ in front of it.

The *open* substitution lemma arises when the scope of y *is not* ended by some end-of-scope in front of the λx. Then we obtain the usual substitution lemma, appropriately enriched with scoping information.

Lemma 3 (Open SL).

$$\begin{aligned} &M[X, x{:=}N, X'][Y, y{:=}P, X'XY'] \\ &\quad = M[Y, y{:=}P, X'xY'][X, x{:=}N[Y, y{:=}P, XY'], X'] \end{aligned}$$

As a corollary we obtain confluence of the ordinary λ-calculus (see Figure 5).

Theorem 4. *$\to_\beta$ / $=_\alpha$ is confluent on Λ.*

Proof. Consider two diverging $\lambda\beta$-reductions $M \to \cdots \to N$ and $M \to \cdots \to P$.

Lift these stepwise to diverging $\curlywedge\beta$-reductions $M \to \cdots \to N'$ and $M \to \cdots \to P'$. (Note that M being a λ-term, it is a (scope-balanced) $\curlywedge$-term .)

By confluence of $\curlywedge\beta$-reduction, we can find some $\curlywedge$-term Q' such that $N' \to \cdots \to Q'$, $P' \to \cdots \to Q'$.

Projecting $N' \to \cdots \to Q'$ and $P' \to \cdots \to Q'$ back to $\lambda\beta$-reduction yields $N \to \cdots \to Q_1$ and $P \to \cdots \to Q_2$, for some α-equivalent λ-terms Q_1 and Q_2, establishing the desired confluence of $\lambda\beta$ modulo α-equivalence.

Let us comment on the proof steps. Both projection and lifting of reductions are performed stepwise. That is, a single $\lambda\beta$-step lifts to a single $\curlywedge\beta$-step and vice versa (not to reduction sequences, as in calculi with explicit substitutions). The forgetful mapping (projection) from $\curlywedge$-terms to λ-terms is the composition of first performing an α-equivalence step followed by a so-called ω-step removing all $\curlywedge$'s in one go[7]. For instance, no ω-step is possible from $\lambda x.\curlywedge x.x$ since removing $\curlywedge x$

[7] ω could be decomposed itself by first pushing $\curlywedge$s to the variables, i.e. performing *maximal* scope extrusion as mentioned in the abstract, before forgetting.

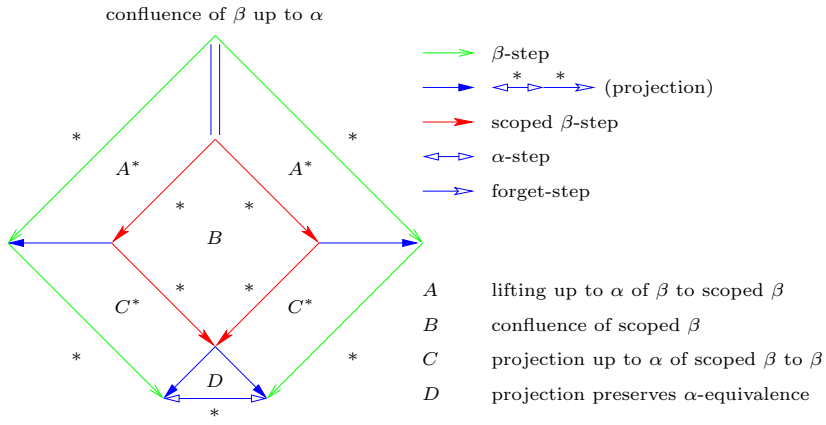

Fig. 5. Confluence of ƛ-calculus implies confluence of λ-calculus

would turn the free variable x into a bound variable in $\lambda x.x$. Obviously, uniquely renaming all variables would guarantee that an ω-step can be performed.

Remark 1. In $\lambda\beta$-reduction renamings are performed, as soon as there's a confusion threat. However, such a threat may turn out to be innocuous, as in:

$$(\lambda y.\lambda x.(\lambda z.I)yx)x \to \lambda x'.(\lambda z.I)xx' \to \lambda x'.Ix'$$

The renaming is caused by the substitution for the variable x which is erased later anyway. On the other hand, no renaming takes place during ƛβ-reduction:

$$(\lambda y.\lambda x.(\lambda z.I)yx)x \to \lambda x.(\lambda z.I)(\lambda x.x)x \to \lambda x.Ix$$

Observe that despite the final term of this λƛ-reduction being an ordinary λ-term, α-conversion *is* needed to project it.

As far as we know the only formalised proof of confluence of β-reduction modulo α, in our setting, i.e. with a single variable space is [26]. However, their proof technique is entirely different, uniquely renaming all variables, before performing β-steps, whereas our schema, which works via the ƛ-calculus, only performs the necessary updates (in the sense of [13]).

4 Applications

We think that the ƛ-calculus provides an intuitive understanding of scoping in the λ-calculus. We claim it can provide solutions to problems which are known to be hard for the λ-calculus. We present some (conjectured) points in case.

Expressing Free Variable Conditions. In the λ-calculus one often has use for free variable conditions. Not only are these necessary to *express* e.g. the η-rule:

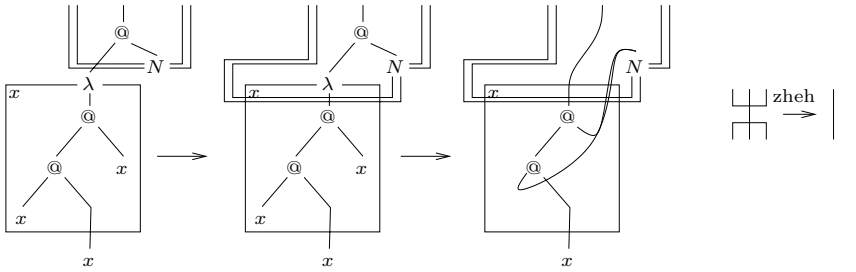

Fig. 6. Left: β-reduction: local scope extrusion and rewiring. Right: scope fusion

$$\lambda x.Mx \to M, \text{ if } x \notin \mathsf{FV}(M),$$

but knowing that x does not occur in the free variables of M would also *speed up* reduction of the β-redex $(\lambda x.M)N$; in that case one may simply erase N. Rather than reifying the negative concept of *a variable not occurring free in a subterm* (cf. e.g. [16]), our $\lambda\!\!\!/$-operator makes the positive concept of *the ending of the scope of a variable* explicit. Using it, the free-variable condition of the η-rule can be expressed in the object language as:

$$\lambda x.(\lambda\!\!\!/ x.M)x \to M,$$

and the β-redex becomes $(\lambda x.\lambda\!\!\!/ x.M)N$, which indeed executes more efficiently.

Optimal Reductions. Lamping provided in [21] the first implementation of the λ-calculus which was optimal in the sense of Lévy [22]. His implementation was based on a translation of λ-terms to graphs having nodes (fan-in and fan-out) for both explicit sharing and unsharing. In order for sharing and unsharing nodes to match up properly (the 'oracle'), he had to introduce two further types of nodes, the control nodes (square bracket and croissant). These control nodes had an *ad hoc* justification and their definitive understanding was considered to be the main open problem of this technique according to [2, Chapter 9].

The oracle, can be understood to arise from making β-reduction in the $\lambda\!\!\!/$-calculus *local* in the sense of [19]. That is scope extrusion and x-box removal as in Figure 4 are to be made local (replication is dealt with by the sharing nodes). A way in which this can be implemented is shown on the left in Figure 6. In fact, a key insight (cf. the second step of Figure 6) is that x-box removal is superfluous as long as scopes can always be moved out of the way (of a β-redex). We have a working optimal implementation of the λ-calculus based on rules achieving just that, such as the zheh-rule in Figure 6 for fusing two adjacent scopes. The implementation performs well on the examples in [2], without the need for either their safe nodes or heuristics (we have only one control node). E.g. computing their most complex example ((f ten) in [2, Figure 9.23]) takes us roughly 5 times as many interactions (compared to BOHM 1.1)[8].

[8] The difference might be explainable by that we do not employ compound nodes.

Explicit Substitution Calculi which Are PSN. This work arose from trying to understand Chapter 4 of [8] on perpetuality in David and Guillaume's calculus with explicit substitutions λ_{ws}, in a named setting (cf. [15]) and in an atomic way. The λ_{ws} calculus was introduced as a calculus having, among other desirable properties, the preservation of normalisation (PSN) property. From [13] we understand that λ_{ws} arose in a seemingly *ad hoc* way from barring counterexamples to PSN for existing calculi with explicit substitutions. We think the ƛ-calculus offers an easy insight why the calculus works as follows.

The problem with PSN arises when one tries to orient, as a reduction rule, the critical pair arising from (an explicit version of) the substitution lemma (see above). The problem with orienting the ensuing critical pair from right to left is that the resulting rule is non-left-linear (L occurs twice in its left-hand side), causing non-confluence, which is undesirable. However, orienting the critical pair from left to right is also problematic since the resulting rule is non-terminating, just by itself, since the left-hand side can be embedded into the right-hand side. (Note that this orientation corresponds to transforming from inside-out to outside-in (standard) order of contraction of the β-redexes.)

The key insight is that in the ƛ-calculus, we can recognise the fact that we are already in outside-in order: consider the substitution lemma above oriented from left to right and enriched with end-of-scope information (but for the moment forgetting the first component of ƛ-substitutions which are empty in this example):

$$M[\, x{:=}N, \Box][\, y{:=}L, \Box] \to M[\, y{:=}L, \underline{x}]\big[\, \underline{x}{:=}N[\, y{:=}L, \Box], \Box\big]$$

Now we *recognise* that the two underlined xs in the rhs match with one another, hence that these substitutions are already in standard order. Forbidding further applications of the rule in such situations, should break the infinite reduction and regain PSN.

Acknowledgments

We would like to thank the participants of the TCS seminar at the Vrije Universiteit Amsterdam, PAM and the 7th Dutch Proof Tools Day both at CWI, Amsterdam, ZIC at the Technische Universiteit Eindhoven, the CS seminar at the University of Leicester, and the TF lunch seminar at the Universiteit Utrecht, for feedback. Eduardo Bonelli, Marko van Eekelen, Joost Engelfriet, Stefan Kahrs, Kees Vermeulen, Albert Visser, and the CADE referees provided useful comments and pointers to the literature.

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. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages.* Cambridge University Press, 1998.

3. H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
4. S. Baro and F. Maurel. The $q\nu$ and $q\nu k$ calculi : name capture and control. PPS prépublication 16, Université Denis Diderot, 2003.
5. K.J. Berkling. A symmetric complement to the lambda-calculus. Interner Bericht ISF-76-7, GMD, D-5205, St. Augustin 1, West Germany, 1976.
6. R.S. Bird and R.A. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.
7. S.C.C. Blom. *Term Graph Rewriting, syntax and semantics*. PhD thesis, Vrije Universiteit Amsterdam, 2001.
8. E. Bonelli. *Substitutions explicites et réécriture de termes*. PhD thesis, Université Paris XI, 2001.
9. L. Cardelli and A.D. Gordon. Mobile ambients. In M. Nivat, editor, *FOSSACS '98*, volume 1378 of *LNCS*, pages 140–155. Springer, 1998.
10. C. Chen and H. Xi. Meta-programming through typeful code representation. `http://www.cs.bu.edu/$\sim$hwxi/`.
11. A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.
12. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://www.grappa.univ-lille3.fr/tata`.
13. R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structures for Computer Science*, 11:169–206, 2001.
14. N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
15. R. Di Cosmo, D. Kesner, and E. Polonovski. Proof nets and explicit substitutions. In *FOSSACS) '00*, volume 1784 of *LNCS*, pages 63–81. Springer, 2000.
16. A.D. Gordon and T.F. Melham. Five axioms of alpha-conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *TPHOLs '96*, volume 1125 of *LNCS*, pages 173–190. Springer, 1996.
17. M. Hollenberg and C.F.M Vermeulen. Counting variables in a dynamic setting. *Journal of Logic and Computation*, 6(5):725–744, 1996.
18. S. Kahrs. Context rewriting. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92*, volume 656 of *LNCS*, pages 21–35. Springer, 1993.
19. Y. Lafont. Interaction nets. In *POPL '90*, pages 95–108. ACM Press, 1990.
20. Y. Lafont. From proof-nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*, pages 225–248. Cambridge University Press, 1995.
21. J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90*, pages 16–30. ACM Press, 1990.
22. J.-J. Lévy. *Réductions correctes et optimales dans le λ-calcul*. Thèse de doctorat d'état, Université Paris VII, 1978.
23. J. Parrow. The fusion calculus: Expressiveness and symmetry in mobile processes. In *LICS '98*, pages 176–185. IEEE Computer Society, 1998.
24. D.E. Schroer. *The Church–Rosser Theorem*. PhD thesis, Cornell University, 1965.
25. Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.
26. R. Vestergaard and J. Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. In A. Middeldorp, editor, *RTA '01*, LNCS, pages 306–321. Springer, 2001.