Chapter 3

$\lambda\!\!\!/$

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 $\lambda\!\!\!/$, 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.

Authors: Dimitri Hendriks and Vincent van Oostrom

3.1 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, see e.g. C.1 DEFINITION of [4], can be neatly partitioned into those dealing with replication (the variable and application clauses) and those dealing with renaming (the abstraction clauses). In this chapter 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.

Abstractions in the λ-calculus can be viewed as being composed of two parts: one part which is dual to application, and another which causes the opening of the scope of the bound variable. 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 $\lambda\!\!\!/$ (adbmal). $\lambda\!\!\!/x.M$ expresses that the scope of x is ended 'above' M. For instance, in the $\lambda\!\!\!/$-term $\lambda x.\lambda\!\!\!/x.\underline{x}$ the underlined occurrence of the variable x is free, since the binding effect of the λx is undone by the subsequent $\lambda\!\!\!/x$. For another example, only the underlined

occurrence of x is free in $\lambda x.x(\not\leftarrow x.\underline{x})x$; the first and third occurrences of x are in scope of the λx (see Figure 3.1).

Definition 3.1.1 *The set* $(M, N, P \in)\Lambda$ *of* $\not\leftarrow$-terms *is defined by:*

$$\Lambda ::= \mathcal{V} \mid \not\leftarrow x.\Lambda \mid \lambda x.\Lambda \mid \Lambda\Lambda$$

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

Axiom 3.1 (Names with decidable equality) $x = y \vee x \neq y$, *for all* $x, y : \mathcal{V}$

We stress that Definition 3.1.1 is an *inductive* one without any reference to α-equivalence. That is, we do *not* assume the variable convention (2.1.13 of [4]). In fact, we don't need it, since in the process of β-reduction, we will not rename offending binders to avoid capturing of free variables. Instead, $\not\leftarrow$s will be inserted in an appropriate way, as will become clear in the sequel.

We adopt the usual notational conventions for the λ-calculus [4] (but *not* the variable convention), treating $\not\leftarrow$ analogously to λ. We use the notation $\lambda X.M$ and $\not\leftarrow X.M$ where X is a stack of variables $x_0, \ldots, x_n$, to denote $\lambda x_0 \cdots x_n.M$ resp. $\not\leftarrow x_0 \cdots x_n.M$.

Remark 3.1.1 *One way in which the usefulness of the* $\not\leftarrow$*-calculus is shown, is by deriving confluence of the standard* λ*-calculus from confluence of the* $\not\leftarrow$*-calculus. One way to do this would be to define standard* λ*-terms as* $\not\leftarrow$*-terms without occurrences of* $\not\leftarrow$*, and then prove some kind of conservativity result. Instead, in our* Coq *implementation* λ*-terms are defined separately from* $\not\leftarrow$*-terms and we use a canonical embedding from the former to the latter. In this way we hope to make it clear that it is really the* standard λ*-calculus we are proving confluent, and also to take away any suspicion of cheating (e.g. employing notions for* $\not\leftarrow$*-terms in the* λ*-calculus). In order to improve readability, mention of the function embedding* λ*-terms into* $\not\leftarrow$*-terms will be suppressed.*

In order to extend the notions of α-equivalence and β-reduction, we should first try to make some semantic sense of $\not\leftarrow$s. Thinking of λx and $\not\leftarrow x$ as (named) opening '$[_x$' and closing '$]_x$' brackets,[1] it is clear that $\not\leftarrow$-terms may come in different degrees of balancedness. For instance, scopes could seemingly be crossing one another as indicated by the boxes in:

$$P = \boxed{\lambda x.\boxed{\lambda y.\not\leftarrow x.}\not\leftarrow y.}\,Q$$

This would obviously cause semantical problems (try to define substitution). To overcome this problem we assume a simple minded jump semantics: an occurrence of $\not\leftarrow x.M$ 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

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

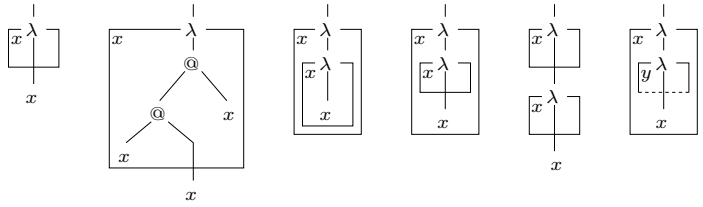

Figure 3.1: $\lambda x.\curlywedge x.x$, $\lambda x.x(\curlywedge x.x)x$, $\lambda x.\lambda x.x$, $\lambda x.\lambda x.\curlywedge x.x$, $\lambda x.\curlywedge x.\lambda x.\curlywedge x.x$, and $\lambda x.\lambda y.x$.

semantically equivalent to $\lambda x.\lambda y.\curlywedge y.\curlywedge x.\curlywedge y.Q$. 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 (see the discussion in the paragraph on related work below). Ordinary λ-terms are not (necessarily) balanced, however they always are scope-balanced.

Definition 3.1.2 *A term is* scope-balanced *if it is scope-balanced* under *some stack X. A term M being* scope-balanced under *a stack X is denoted by $\langle X\rangle M$ and defined by:*

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

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

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

Here xX is the result of pushing x on the stack X.

For instance, $\lambda x.\lambda y.\curlywedge x.M$ is not scope-balanced (λy not closed before $\curlywedge x$), $\lambda y.x$ is scope-balanced but not balanced (λy not closed before x), and $\lambda x.\curlywedge x.M$ and $\lambda x.x$ are balanced (if M is in the former case).

It is easy to see that closed λ-terms are scope-balanced under any stack, hence in particular under the empty stack $\Box$. Scopes in balanced $\curlywedge$-terms can be neatly visualised as boxes in their abstract syntax tree, as shown in Figure 3.1.[2] Vice versa, in the term representation of a box, only its 'doors' are kept. That is, λs and $\curlywedge$s 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 3.1) between balanced terms and the context-free string language of *matching brackets* as presented by the grammar:

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

[2]Scopes in non-balanced terms can be drawn as floorless boxes ($\lambda x.\lambda y.x$ in Figure 3.1).

- 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 $\nleftarrow x$ in $\lambda x.\lambda x.\nleftarrow x.x$.
- Scopes can be concatenated (similar to PP). In the $\nleftarrow$-term $\lambda x.\nleftarrow x.\lambda x.\nleftarrow x.x$, 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 $\nleftarrow$-terms can be generated by a so-called context-free term grammar, where context-free term grammars are the natural generalisation of context-free string grammars, see e.g. Section 2.5 of [23]. A difference between matching bracket strings and balanced $\nleftarrow$-terms is that, due to the branching structure of terms, *several* $\nleftarrow$'s may match the same λ as in $\lambda x.(\nleftarrow x.\underline{x})(\nleftarrow x.\underline{x})$, with both underlined occurrences of x free.

Related work This chapter is the full version of [34], and is under consideration for publication in the *Journal of Functional Programming*. Compared to [34] the results in the present chapter are more general, in particular the key substitution lemmas. Moreover, we have supplied more formal definitions and some typical proof ideas. For the complete proof development we refer the reader to [35].

When application of $\nleftarrow x$ is restricted to variables (and end-of-scopes), it corresponds to Berkling's lambda-bar [11], which is in turn seen to be a named version of the successor operator in De Bruijn's nameless (more precisely: single name) calculus [19]. 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, $\nleftarrow x$ corresponds to the shift substitution $[\uparrow]$ in the λ-calculus with explicit substitutions $\lambda\sigma$ of [1], or the shift operation *Shi* of [21]. The earliest generalisation of De Bruijn indices seems to be due to Paterson [56]. The idea is to allow the successor S to appear on subterms, instead of just on indices; as it is written in [15]:

> Substitution on de Bruijn terms transforms arguments as well as function bodies, thus precluding sharing. Consider the example term from Section 1, with the variables written in unary notation:
>
> $$\lambda.0\,(\lambda.\mathsf{S}0\,0\,(\lambda.\mathsf{SS}0\,\mathsf{S}0\,0))$$
>
> If this term is applied to the term $\lambda.0\,\mathsf{S}0$, the result is
>
> $$(\underline{\lambda.0\,\mathsf{S}0})\,(\lambda.(\underline{\lambda.0\,\mathsf{SS}0})\,0\,(\lambda.(\underline{\lambda.0\,\mathsf{SSS}0})\,\mathsf{S}0\,0))$$
>
> where the three versions of the argument are underlined. There is a generalisation of de Bruijn notation in which S can be applied to any term, not just a variable (Paterson, 1991). Its effect is to escape the scope of the matching λ. With this looser representation of terms, one can avoid transforming arguments while substituting. In the above example, substitution yields
>
> $$(\underline{\lambda.0\,\mathsf{S}0})\,(\lambda.\mathsf{S}(\underline{\lambda.0\,\mathsf{S}0})\,0\,(\lambda.\mathsf{SS}(\underline{\lambda.0\,\mathsf{S}0})\,\mathsf{S}00))$$
>
> In effect, we have postponed pushing the S's down to the variables.

where the 'example term from Section 1' is the λ-term $\lambda x.x\ (\lambda y.x\ y\ (\lambda z.x\ y\ z))$ which translates to $\lambda 0.\ (\lambda.1\ 0\ (\lambda.2\ 1\ 0))$ in (their) De Bruijn notation. Applying the λ-term to a named version, say $\lambda x.xy$ of $\lambda.0\,\mathsf{S}0$, yields (see Subsection 3.4.2) $(\underline{\lambda x.xy})\ (\lambda y.(\curlywedge y.\underline{\lambda x.xy})\ y\ (\lambda z.(\curlywedge z.\curlywedge y.\underline{\lambda x.xy}))\ y\ z)$. So obviously $\curlywedge$ corresponds to the successor S in De Bruijn notation.

Bird and Paterson go on to show that in the balanced (single name) case the term language of the $\curlywedge$-calculus is context-free by presenting it by means of the following context-free term grammar:

$$\begin{aligned} \mathit{Term}\ a &::= \mathit{Var}\ a \mid \mathit{App}(\mathit{Term}\ a, \mathit{Term}\ a) \mid \mathit{Abs}(\mathit{Term}(\mathit{Incr}(\mathit{Term}\ a))) \\ \mathit{Incr}\ a &::= \mathit{Zero} \mid \mathit{Succ}\ a \end{aligned}$$

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. [46]. In fact, in our optimal implementation (see Section 3.5) 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 [5]. 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. [20, 55, 16] for some recent ones.

In the area of dynamic semantics for natural language, a stack-based semantics for a variant of predicate logic is presented in [37]. 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 [11] variables have their own stack.

Implementation All results informally presented here are formalised in Coq. The source files are available from [35]. The size of the development is 9543 lines of Coq-code, 259160 bytes; 324 lemmas are proved. The Coq proofs and the $\curlywedge$-calculus were developed concurrently. The total development time is estimated one man-year approximately.

Outline The outline of the rest of this chapter is as follows. In Section 3.2 we define some preliminary notions on abstract rewriting systems. We provide several definitions of α-equivalence for $\curlywedge$-terms in Section 3.3, extending classical

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

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 $\lambda\!\!\!\lambda$-terms in Section 3.4, 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 $\lambda\!\!\!\lambda$-calculus entail the corresponding results for the ordinary λ-calculus, e.g. confluence of β-reduction up to α-equivalence. Applications are presented in Section 3.5. Finally, in Section 3.6, we conclude, and discuss upon the relationship of the $\lambda\!\!\!\lambda$-calculus to explicit substitution calculi having the property of preservation of normalisation.

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.

3.2 Preliminaries

Definition 3.2.1 *We define the n-fold composition $\rightarrow^n$ of a binary relation $\rightarrow$ as the smallest relation satisfying the following clauses:*

$$\frac{}{x \rightarrow^0 x}\ \textit{reflexivity} \qquad \frac{x \rightarrow y}{x \rightarrow^1 y}\ \textit{embedding} \qquad \frac{x \rightarrow^n y \quad y \rightarrow^m z}{x \rightarrow^{n+m} z}\ \textit{transitivity}$$

The reflexive–transitive closure $x \rightarrow^* y$ of $\rightarrow$ is defined as $\exists n.x \rightarrow^n y$. The equivalence closure *$\leftrightarrow^*$ of $\rightarrow$ is defined as $\exists n.x \leftrightarrow^n y$, where $\leftrightarrow^n$ is defined inductively by the former three clauses (replacing all occurrences $\rightarrow^k$ with $\leftrightarrow^k$) plus the following one:*

$$\frac{y \leftrightarrow^n x}{x \leftrightarrow^n y}\ \textit{symmetry}$$

Definition 3.2.2 *A binary relation R on a set A has the* diamond property*, if for all $a, b, c : A$, $a\ R\ b$ and $a\ R\ c$ implies there exists $d : A$, such that $c\ R\ d$ and $b\ R\ d$. R is* confluent *if its reflexive–transitive closure R^* has the diamond property. We say R has the* diamond property up to S *if for all $a, b, c : A$, $a\ R\ b$ and $a\ R\ c$ implies there exist $d, d' : A$, such that $c\ R\ d$, $b\ R\ d'$, and $d\ S\ d'$. R is* confluent up to S *if R^* has the diamond property up to S.*

Note that the ordinary diamond property is equivalent to the diamond property up to identity.

Definition 3.2.3 *We say R* transits *S if $R \subseteq S \subseteq R^*$, where $R_1 \subseteq R_2$ is defined by $\forall xy.x\ R_1\ y \Rightarrow x\ R_2\ y$.*

Lemma 3.2.1 *If R has the diamond property, then it is confluent.*

Proof. (See, for example, [67].) By induction on the complexity of the diverging steps, loading it by: converging steps have the same complexity as opposite diverging steps. Here we express the complexity by the number of R-steps. The following statement is proved by induction on the diverging steps:

$$x\ R^n\ y \wedge x\ R^m\ z \Rightarrow \exists u.y\ R^m\ u \wedge z\ R^n\ u$$

Lemma 3.2.2 *If R transits S and S has the diamond property, then R is confluent.*

Proof. By monotonicity (if $R \subseteq R'$, then $R^* \subseteq R'^*$), idempotence ($R^{**} = R^*$) of *, and the first assumption, we have $R^* \subseteq S^* \subseteq R^*$. We conclude from the previous lemma and the second assumption.

3.3 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 $\lambda\!\!\!\!\backslash$-calculus, and prove them to be equivalent. For this the existence of fresh variables is required:

Axiom 3.2 (Fresh variable) *For any given (finite) stack of variables, there is a variable not among these, i.e. a fresh variable, $\forall X.\exists x.x \notin X$.*

3.3.1 $\lambda\alpha$

Church

Our first notion of α-equivalence is the usual one based on Church's Postulate I for the λ-calculus [22], 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* (page 350 of [22]).

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 relating 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 variable 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 3.3.1 *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:*

$$\lambda\underline{x}.\lambda y.\underline{x}y \to_\alpha \lambda z.\lambda\underline{y}.z\underline{y} \to_\alpha \lambda\underline{z}.\lambda x.\underline{z}x \to_\alpha \lambda y.\lambda x.yx$$

where we have underlined in each term the variables that are converted in the subsequent step.

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 the number of steps is by restricting α-conversion by:

> Never rename twice.

However, from Example 3.3.1 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, see page 384 of his [62]:

> 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 is 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 3.3.2 *Applied to Example 3.3.1 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, as noted, e.g., by [27]. This fact may be seen by proceeding in a top-down fashion, the only interesting case being abstraction:

(abstraction) Suppose $s = \lambda x.s'$ and $t = \lambda y.t'$, such that the variables 'above' them have already been made identical. We proceed by first converting *every* x in s into z. Next, *every* y is converted into x and finally, *every* z is converted into y, resulting in, say, $\hat{s}$. Now $\hat{s}$ and t have the same initial binder, and we proceed on the respective subterms. To prove that this procedure is correct, one uses that y does not occur free in s' since otherwise s and t would not be α-equivalent. Symmetrically, x does not occur free in t'.

Example 3.3.3 *α-converting, say, $\lambda x_1x_2x_3.x_1x_2x_3$ into $\lambda x_2x_3x_1.x_2x_3x_1$ using this procedure proceeds as follows. First, we swap, using a fresh variable y, x_1 with x_2 yielding $\lambda x_2x_1x_3.x_2x_1x_3$. Hence the first variable has been appropriately renamed, and we may proceed on the respective subterms. In order to α-convert $\lambda x_1x_3.x_2x_1x_3$ into $\lambda x_3x_1.x_2x_3x_1$, we swap, using the same y, x_1 and x_3 in the former yielding the latter and we are done (formally one needs to continue with twice the subterm $\lambda x_1.x_2x_3x_1$, but nothing 'happens' anymore.)*

Kahrs

Both the problem of showing transitivity and the need for the Fresh variable 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 3.3.1 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 3.3.4 *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.*

[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.

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

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

Figure 3.2: Proof rules for α-congruence [43].

To define α-equivalence inductively, one has to set up a correspondence between *stacks* of variables. Such an idea appears in Kahrs' paper [43]; 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.

where we present the proof rules of 'figure 2' in Figure 3.2. It is to be understood that two terms A and B are α-equivalent, if $\epsilon \vdash A \equiv B$ can be derived by the proof rules in Figure 3.2. 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.

3.3.2 $\lambda\!\!\!\lambda\alpha$

We show that each of the three definitions of α-equivalence can be straightforwardly extended from λ-terms to $\lambda\!\!\!\lambda$-terms. In each case, we highlight the key aspect of our formalisation in Coq. We start with some necessary technicalities. Apart from the 'renaming of boxes' (Definition 3.3.2), these may be skipped by the experienced reader who is referred to page 61.

We shall need to test whether a variable is fresh with respect to a term.

Definition 3.3.1 *The relation $x \in M$, saying whether x occurs in M (free or bound) is defined by:*

$$\begin{array}{l} x \in y, \textit{ if } x = y \\ x \in \lambda y.M, \textit{ if } x = y \textit{ or } x \in M \\ x \in \lambda\!\!\!\lambda y.M, \textit{ if } x = y \textit{ or } x \in M \\ x \in MN, \textit{ if } x \in M \textit{ or } x \in N \end{array}$$

We say x is fresh for *M if $x \notin M$.*

As usual, α-conversion is defined using a renaming function to identify expressions that differ only in the names assigned to their bound variables. For $\curlywedge$-terms this means the renaming of *boxes* (viz. Figure 3.1). Renaming the (outer) x-box in $\lambda x.M$ into a y-box, means to replace λx by λy and to replace all *matching* occurrences of variables x and end-of-scopes $\curlywedge x$ in M by y and $\curlywedge y$ respectively. The first argument of renaming ($x{:=}y$) is fixed (as usual). Matching is performed using a stack (the second argument of renaming, initially empty) to record the binders encountered while descending recursively; it is pushed upon when passing an abstraction, and popped from when meeting an end-of-scope:

Definition 3.3.2 Renaming the outer x-box in $\lambda x.M$ into a y-box, $\lambda y.M[x{:=}y]$, *is defined using* $M[x{:=}y] = M[x{:=}y, \Box]$, *where* $M[x{:=}y, Z]$ *is defined by the following recursive equations:*

$$\begin{array}{rcll} z[x{:=}y, Z] & = & z, \text{ if } z \in Z \text{ or } z \neq x & (1)\\ z[x{:=}y, Z] & = & y, \text{ if } z \notin Z \text{ and } z = x & (2)\\ (\lambda z.M)[x{:=}y, Z] & = & \lambda z.M[x{:=}y, zZ] & (3)\\ (\curlywedge z.M)[x{:=}y, \Box] & = & \curlywedge y.M, \text{ if } x = z & (4)\\ (\curlywedge z.M)[x{:=}y, \Box] & = & \curlywedge z.M, \text{ if } x \neq z & (5)\\ (\curlywedge z.M)[x{:=}y, z'Z] & = & \curlywedge z.M[x{:=}y, Z], \text{ if } z = z' & (6)\\ (\curlywedge z.M)[x{:=}y, z'Z] & = & (\curlywedge z.M)[x{:=}y, Z], \text{ if } z \neq z' & (7)\\ (M_1M_2)[x{:=}y, Z] & = & M_1[x{:=}y, Z]M_2[x{:=}y, Z] & (8) \end{array}$$

In clause (5) $\curlywedge z$ implicitly closes the scope of x; therefore, in the result, we can think of $\curlywedge z$ as implicitly closing the scope of y.

Example 3.3.5 $(x\curlywedge x.x)(\lambda x.(x\curlywedge x.x))[x{:=}y] = (y\curlywedge y.x)\lambda x.(x\curlywedge x.y)$

In case of ordinary λ-terms the stack Z only grows during renaming, rendering the variable to be renamed inaccessible once it is abstracted from again:

Lemma 3.3.1 *If* M *is a* λ*-term, then* $(\lambda x.M)[x{:=}y] = \lambda x.M$.

Proof. Note that $(\lambda x.M)[x{:=}y] = \lambda x.M[x{:=}y, x]$. One proves by induction on the λ-term M, that if x occurs in X, then $M[x{:=}y, X] = M$.

The following two properties of renaming will be needed in the proof of commutation of β-reduction and α-equivalence. The first states commutativity of renaming:

Lemma 3.3.2 $M[x{:=}z, XyY][y{:=}z', X] = M[y{:=}z', X][x{:=}z, Xz'Y]$, *if* z *fresh for* y, M, X *and* z' *fresh for* M, X.

Secondly, renaming x into z, followed by renaming z into z', amounts to the same as directly renaming x into z':

Lemma 3.3.3 $M[x{:=}z, X][z{:=}z', X] = M[x{:=}z', X]$, *if* $z \notin M, X$.

When we project β-reductions in the $\curlywedge$-calculus to β-reductions in the λ-calculus (Section 3.4.4), we need to reason about the set of free variables of a term:

Definition 3.3.3 *The set of free variables of a term M, $\mathsf{FV}(M)$, is defined as $\mathsf{FV}(M, \Box)$, where $\mathsf{FV}(M, X)$ is defined by the following recursive equations.*

$$\begin{array}{rcll} \mathsf{FV}(x, X) & = & \{x\} - X & (1) \\ \mathsf{FV}(\lambda x.M, X) & = & \mathsf{FV}(M, xX) & (2) \\ \mathsf{FV}(\curlywedge x.M, \Box) & = & \mathsf{FV}(M, \Box) & (3) \\ \mathsf{FV}(\curlywedge x.M, x'X) & = & \mathsf{FV}(M, X), \text{ if } x = x' & (4) \\ \mathsf{FV}(\curlywedge x.M, x'X) & = & \mathsf{FV}(\curlywedge x.M, X), \text{ if } x \neq x' & (5) \\ \mathsf{FV}(MN, X) & = & \mathsf{FV}(M, X) \cup \mathsf{FV}(N, X) & (6) \end{array}$$

Note that, in clause (5), $\curlywedge x$ implicitly closes the scope x', which is therefore popped from the stack of currently open scopes.

Remark 3.3.1 *Note that if M is balanced under X we have $\mathsf{FV}(M, X) = \emptyset$. This implies that terms balanced under the empty stack are closed.*

In the sequel it will sometimes be useful to forget about the binding structure of terms. To that end, we map terms to first-order terms by simply forgetting names (equivalently: mapping all names to a single one):

Definition 3.3.4 *First we define a set T of first-order terms:*

$$T ::= \odot \mid \curlywedge T \mid \lambda T \mid @TT$$

The skeleton $[M]_{\mathsf{skel}}$ *of an $\curlywedge$-term M is such a first-order term defined inductively by:*

$$\begin{array}{rcl} [x]_{\mathsf{skel}} & = & \odot \\ [\curlywedge x.M]_{\mathsf{skel}} & = & \curlywedge [M]_{\mathsf{skel}} \\ [\lambda x.M]_{\mathsf{skel}} & = & \lambda [M]_{\mathsf{skel}} \\ [M_1 M_2]_{\mathsf{skel}} & = & @[M_1]_{\mathsf{skel}} [M_2]_{\mathsf{skel}} \end{array}$$

For instance, the proof of Lemma 3.4.20 proceeds by induction over the skeleton of a term, using the fact that renaming preserves skeletons:

Lemma 3.3.4 $[M[x{:=}y]]_{\mathsf{skel}} = [M]_{\mathsf{skel}}$.

Unary contexts are used to express congruences.

Definition 3.3.5 Unary contexts *are typed $\curlywedge \to \curlywedge$ and built from*

$$[\,], C \circ C', \lambda x.[\,], \curlywedge x.[\,], M[\,], [\,]M$$

where C, C' are unary contexts. We write $C[M]$ to denote the result of filling the hole in C by term M; $(C \circ C')[M] = C[C'[M]]$.

Now that we have defined some technical preliminaries, we are ready to extend the three notions of α-equivalence given in Subsection 3.3.1 to the $\curlywedge$-calculus.

Church

The notion of α-conversion is extended to the $\lambda\!\!\!/$-calculus.

Definition 3.3.6 *The α-rule is $\lambda x.M \to \lambda y.M[x{:=}y]$ if $y \notin M$. Single-step α-renaming, $\to_\alpha$, is defined as the compatible closure of the α-rule:*

$$\frac{(M,N) \in \alpha}{M \to_\alpha N} \qquad \frac{M \to_\alpha N}{\lambda x.M \to_\alpha \lambda x.N} \qquad \frac{M \to_\alpha N}{\lambda\!\!\!/ x.M \to_\alpha \lambda\!\!\!/ x.N}$$

$$\frac{M \to_\alpha M'}{MN \to_\alpha M'N} \qquad \frac{N \to_\alpha N'}{MN \to_\alpha MN'}$$

The relation $=_\alpha^{\mathsf{c}}$, which we define as $\leftrightarrow_\alpha^*$, i.e. the equivalence closure of $\to_\alpha$, is called α-conversion.

The clause dealing with $\lambda\!\!\!/$ is just a compatibility clause, cf. 3.1.1. Definition of [4], since at the time one comes across an $\lambda\!\!\!/$, all the (renaming) work has already been performed by its matching abstraction. Due to Lemma 3.3.1, our definitions of α-step and α-conversion coincide with that of [4] in the case of λ-terms.

Schroer

Our definition of α-equivalence à la Schroer makes use of an auxiliary stack Z which records the variables chosen thusfar for renaming.

Definition 3.3.7 α-equivalence à la Schroer*, $M =_\alpha^{\mathsf{s}} N$, is defined as $\exists Z.M =_\alpha^Z N$, where $M =_\alpha^Z N$ is defined by:*

$$\frac{M[x{:=}z] =_\alpha^Z N[y{:=}z] \quad z \notin M,N,Z}{\lambda x.M =_\alpha^{zZ} \lambda y.N}$$

$$\frac{}{x =_\alpha^Z x} \qquad \frac{M =_\alpha^Z N}{\lambda\!\!\!/ x.M =_\alpha^Z \lambda\!\!\!/ x.N} \qquad \frac{M =_\alpha^Z M' \quad N =_\alpha^Z N'}{MN =_\alpha^Z M'N'}$$

Again all the work is performed in the clause for abstraction. Compared to α-conversion $=_\alpha^{\mathsf{c}}$ 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. The clause dealing with $\lambda\!\!\!/$ is just a compatibility clause, as above.

Kahrs

Our definition of α-equivalence à la Kahrs reads as follows. It makes use of two auxiliary stacks (both initially empty), to set up the correspondence between the variables in M and N mentioned above.

Definition 3.3.8 *We define $M =^{\mathsf{k}}_{\alpha} N$, if $\langle\Box\rangle M =^{\mathsf{k}}_{\alpha} \langle\Box\rangle N$, where for stacks of variables X and Y, $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ is inductively defined as follows:*

$$\begin{array}{rcl}
\langle\Box\rangle x & =^{\mathsf{k}}_{\alpha} & \langle\Box\rangle x \\
\langle xX\rangle x & =^{\mathsf{k}}_{\alpha} & \langle yY\rangle y, \text{ if } |X| = |Y| \\
\langle x'X\rangle x & =^{\mathsf{k}}_{\alpha} & \langle y'Y\rangle y, \text{ if } x' \neq x,\ y' \neq y, \text{ and } \langle X\rangle x =^{\mathsf{k}}_{\alpha} \langle Y\rangle y \\
\langle X\rangle \lambda x.M & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle \lambda y.N, \text{ if } \langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N \\
\langle\Box\rangle \mathcal{K} x.M & =^{\mathsf{k}}_{\alpha} & \langle\Box\rangle \mathcal{K} x.N, \text{ if } \langle\Box\rangle M =^{\mathsf{k}}_{\alpha} \langle\Box\rangle N \\
\langle xX\rangle \mathcal{K} x.M & =^{\mathsf{k}}_{\alpha} & \langle yY\rangle \mathcal{K} y.N, \text{ if } \langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \\
\langle x'X\rangle \mathcal{K} x.M & =^{\mathsf{k}}_{\alpha} & \langle y'Y\rangle \mathcal{K} y.N, \text{ if } x' \neq x,\ y' \neq y \text{ and } \langle X\rangle \mathcal{K} x.M =^{\mathsf{k}}_{\alpha} \langle Y\rangle \mathcal{K} y.N \\
\langle X\rangle M_1 M_2 & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle N_1 N_2, \text{ if } \langle X\rangle M_1 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_1 \text{ and } \langle X\rangle M_2 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_2
\end{array}$$

where $|X|$ denotes the length of stack X.

The variable, abstraction and application clauses in the definition above can easily be seen to be corresponding to the clauses in Figure 3.2. The end-of-scope clauses are analogous to the clauses for variables. Unique reading holds up to α-equivalence:

Lemma 3.3.5 *If $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$, then*

- *M and N have the same skeleton: $[M]_{\mathsf{skel}} = [N]_{\mathsf{skel}}$.*
- *X and Y have the same length: $|X| = |Y|$.*
- *M and N have the same set of free variables: $\mathsf{FV}(M, X) = \mathsf{FV}(N, Y)$.*

As a consequence we have conservativity of α-equivalence over the ƛ-calculus (Lemma 3.3.7). Note that the third end-of-scope clause of Definition 3.3.8 expresses that ending the scope of some variable x automatically ends the scope of the variables which were bound later than x. By conservativity of $=^{\mathsf{k}}_{\alpha}$, this clause can be omitted for scope-balanced terms. For balanced terms we can do with only four clauses:

$$\begin{array}{rcl}
\langle x\rangle x & =^{\mathsf{k}}_{\alpha} & \langle y\rangle y \\
\langle xX\rangle \mathcal{K} x.M & =^{\mathsf{k}}_{\alpha} & \langle yY\rangle \mathcal{K} y.N \\
\langle X\rangle \lambda x.M & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle \lambda y.N, \text{ if } \langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N \\
\langle X\rangle M_1 M_2 & =^{\mathsf{k}}_{\alpha} & \langle Y\rangle N_1 N_2, \text{ if } \langle X\rangle M_1 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_1 \text{ and } \langle X\rangle M_2 =^{\mathsf{k}}_{\alpha} \langle Y\rangle N_2
\end{array}$$

If $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ holds, then the pair of stacks (X,Y) can be seen as an update on the identity relation (which obviously is a bijection) on variable names, the result of which is a bijection between the 'free' variables of M and N. Indeed, as stated by the following lemma, inserting the same variable on the bottom of both stacks is irrelevant. This lemma is needed, e.g., to show that $=^{\mathsf{k}}_{\alpha}$ is a congruence (abstraction case).

Lemma 3.3.6 *$\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ iff $\langle Xz\rangle M =^{\mathsf{k}}_{\alpha} \langle Yz\rangle N$*

Proof. The proof is by induction on the derivations. The only interesting cases are the variable and end-of-scope cases, which are similar. So, suppose $M = x$ and $N = y$.

($\Leftarrow$) By induction on stack X, and inversion[6] of the instances of $\langle Xz\rangle x =^{\mathsf{k}}_{\alpha} \langle Yz\rangle y$.

- If $X = \Box$, then $Y = \Box$ (by Lemma 3.3.5), and $\langle\Box\rangle x =^{\mathsf{k}}_{\alpha} \langle\Box\rangle y$ follows, since either $x = z = y$ or $x \neq z \neq y$.
- If $X = x'X'$, then $Y = y'Y'$ (by Lemma 3.3.5). Inverting $\langle x'X'z\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Y'z\rangle y$ gives two possibilities. Either $x = x'$, $y = y'$ and $|X'z| = |Y'z|$, then $\langle x'X'\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Y'\rangle y$ follows by application of the second clause; or $x \neq x'$ and $y \neq y'$, then our goal follows by application of the third clause of $=^{\mathsf{k}}_{\alpha}$ and the induction hypothesis.

($\Rightarrow$)
- Case $\langle\Box\rangle x =^{\mathsf{k}}_{\alpha} \langle\Box\rangle x$. If $x = z$, then we conclude by application of the second defining clause of $=^{\mathsf{k}}_{\alpha}$. If $x \neq z$, then $\langle z\rangle x =^{\mathsf{k}}_{\alpha} \langle z\rangle x$ follows from the third clause of $=^{\mathsf{k}}_{\alpha}$ and the assumption.
- Case $\langle xX\rangle x =^{\mathsf{k}}_{\alpha} \langle yY\rangle y$ with $|X| = |Y|$; so $|Xz| = |Yz|$ and $\langle xXz\rangle x =^{\mathsf{k}}_{\alpha} \langle yYz\rangle y$ follows from application of the second clause of $=^{\mathsf{k}}_{\alpha}$.
- Case $\langle x'X\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Y\rangle y$, where $x \neq x'$ and $y \neq y'$; then $\langle x'Xz\rangle x =^{\mathsf{k}}_{\alpha} \langle y'Yz\rangle y$ follows from application of the third rule of $=^{\mathsf{k}}_{\alpha}$ and the induction hypothesis.

Results on α-equivalences

Theorem 3.3.1 *All three notions of α-equivalence are equivalent:*

$$=^{\mathsf{c}}_{\alpha} = =^{\mathsf{s}}_{\alpha} = =^{\mathsf{k}}_{\alpha}$$

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

Theorem 3.3.2 *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.

Lemma 3.3.7 *α-equivalence preserves λ-terms, scope-balancedness, balancedness, and λ-terms.*

[6] *Inverting* a statement $P(t)$, where P is an inductive predicate, means to derive for each possible constructor $c_i : A_1 \to \cdots \to A_n \to P(t)$ all the necessary conditions $A_1, \ldots, A_n$ that should hold for the instance $P(t)$ to be proved by c_i.

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).

Remark 3.3.2 *Proving the three definitions of α-equivalence to be equivalent served mainly as sanity check for our extension of α-equivalence from λ- to $\lambda\!\!\!\lambda$-calculus. We do not (cl)aim to have covered all definitions of α-equivalence in the literature, see e.g. [68], but, based on the above, we strongly believe that the notion we have captured is the proper one. During proof development, (the generalisation 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 $\lambda\!\!\!\lambda$-calculus as well (defined, say, analogously to Chapter 12 of [67]).*

3.4 β

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

3.4.1 $\lambda\beta$

In Chapter 3 of [4], the binary relation $\rightarrow_\beta$ 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 right-hand side 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.[7]

The first difference is 'under the hood'. Curry's definition is *not* a recursive one (to Coq) because of its final clause. Instead, we base our recursive definition on the skeleton $[M]_{\mathsf{skel}}$ (Definition 3.3.4).

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

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

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 is the threat of confusion of variables. The definition is recursive (to Coq) if one decrees 'threat of confusion of variables' larger than 'no confusion'.

Definition 3.4.1 *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 *such that* $\lambda y.M_1 =_\alpha \lambda z.M_1[y{:=}z]$,
	$x \neq z$, *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).

Remark 3.4.1 *The proviso that names are ordered linearly only serves to make Curry's definition definite. However, the definiteness assumption doesn't cause β-reduction to be definite. That is, different β-reductions possibly result in α-different normal forms, as in the following example from [36]; abbreviate $M = (\lambda x.(\lambda y.\lambda x.xy)x)y$:*

$$M \to_\beta (\lambda x.\lambda z.zx)y \to_\beta \lambda z.zy$$
$$M \to_\beta (\lambda y.\lambda x.xy)y \to_\beta \lambda x.xy$$

Thus, the definiteness of Curry's definition still gives an arbitrary choice. Even stronger, as no definite α-renaming scheme would cause β-reduction to be definite, we don't assume definiteness. Of course, for implementation purposes often a choice function *is needed.*

3.4.2 $\lambda\beta$

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 λs which are 'inbetween' the application and the abstraction. In such cases, the scopes of the λs are *extruded* in a minimal way so as to contain the scope of the abstraction, after which β-reduction proceeds as usual (see Figure 3.3, where it is irrelevant where scopes are in N). In order to perform all these operations in one go, our notion of substitution as employed by β-reduction has three arguments, of which the second corresponds to the usual one.

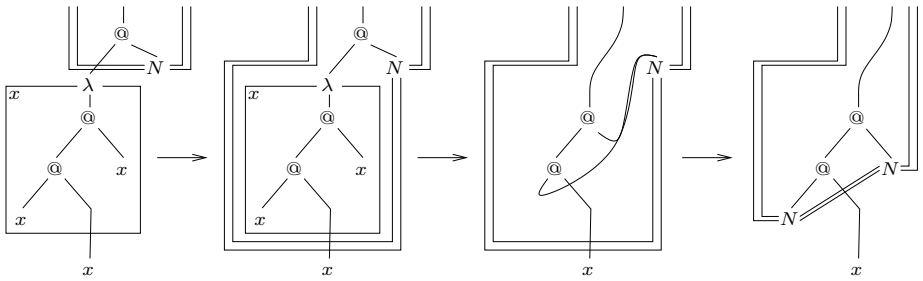

Figure 3.3: β-reduction: scope extrusion, rewiring and x-box removal, and replication.

Definition 3.4.2 *The β-rule is $(\lambda\!\!\!\lambda X.\lambda x.M)N \to M[X, x{:=}N, \Box]$. The relation $\to_\beta$ is the compatible closure of the β-rule:*

$$\frac{(M,N) \in \beta}{M \to_\beta N} \qquad \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N} \qquad \frac{M \to_\beta N}{\lambda\!\!\!\lambda x.M \to_\beta \lambda\!\!\!\lambda x.N}$$

$$\frac{M \to_\beta M'}{MN \to_\beta M'N} \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}$$

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

$$\begin{array}{rcll}
y[X, x{:=}N, Y] & = & y, \text{ if } y \in Y & (1)\\
y[X, x{:=}N, Y] & = & \lambda\!\!\!\lambda Y.N, \text{ if } y \notin Y,\ x = y & (2)\\
y[X, x{:=}N, Y] & = & \lambda\!\!\!\lambda Y.\lambda\!\!\!\lambda X.y, \text{ if } y \notin Y,\ x \neq y & (3)\\
(\lambda y.M)[X, x{:=}N, Y] & = & \lambda y.M[X, x{:=}N, yY] & (4)\\
(\lambda\!\!\!\lambda y.M)[X, x{:=}N, JyY'] & = & \lambda\!\!\!\lambda y.M[X, x{:=}N, Y'], \text{ if } y \notin J & (5)\\
(\lambda\!\!\!\lambda y.M)[X, x{:=}N, Y] & = & \lambda\!\!\!\lambda Y.\lambda\!\!\!\lambda X.M, \text{ if } y \notin Y,\ x = y & (6)\\
(\lambda\!\!\!\lambda y.M)[X, x{:=}N, Y] & = & \lambda\!\!\!\lambda Y.\lambda\!\!\!\lambda X.\lambda\!\!\!\lambda y.M, \text{ if } y \notin Y,\ x \neq y & (7)\\
(M_1M_2)[X, x{:=}N, Y] & = & M_1[X, x{:=}N, Y]M_2[X, x{:=}N, Y] & (8)
\end{array}$$

Capture of free variables in the argument N is avoided by closing all open scopes Y, as expressed in clause (2). In case $\lambda\!\!\!\lambda X$ is to be put, the open scopes Y have to be closed first $(3, 6, 7)$, as will be explained below. Important clauses are (6) and (7), 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.

To explain clauses $(5, 6, 7)$, consider an initial call:

$$(\lambda\!\!\!\lambda y.M)[X, x{:=}N, \Box]$$

Firstly, note that jumps *within* the body M remain untouched. To see this, imagine a recursive call:

$$(\curlywedge y.M')[X, x{:=}N, JyY']$$

on a subterm $\curlywedge y.M'$ of M, and suppose $y \notin J$. According to our jump semantics, $\curlywedge y$ implicitly ends the scopes J; this implicitness is kept in the resulting $\curlywedge y.M'[X, x{:=}N, Y']$ (clause (5)).

Secondly, in order for minimal scope extrusion of the X (originating from a β-redex) to be safe, the opened scopes J in $(\curlywedge y.M')[X, x{:=}N, J]$ have to be closed explicitly in case $y \notin J$, i.e. in case $\curlywedge y$ ends the scope of some λy *outside* the body M. In that case, jump semantics prescribes that the scope of the substitution variable x is closed either explicitly ($x = y$, (6)) or implicitly ($x \neq y$, (7)) by $\curlywedge y$ and we want to put the $\curlywedge X$. Now if we do not put the $\curlywedge J$ first, there is the threat that X *explicitly* closes J, which is unintended. This point is demonstrated by the following example:

Example 3.4.1 *Consider the term $P = (\curlywedge z.\lambda y.(\lambda x.\curlywedge y.M)N)L$ and note that both scopes of the abstractions λx and λy are closed in front of M, as $\curlywedge y$ implicitly closes λx. Therefore both N and L vanish when reducing P, as shown by the following two reduction paths from P. Substitutions are written out explicitly; the numbers stacked above '=' refer to the clauses in Definition 3.4.3. Assume $x \neq y$. The sequence starting with contraction of the inner redex of P runs as follows:*

$$\begin{array}{rcl} P & \to_\beta & (\curlywedge z.\lambda y.(\curlywedge y.M)[\Box, x{:=}N, \Box])L \\ & \overset{7}{=} & (\curlywedge z.\lambda y.\curlywedge y.M)L \\ & \to_\beta & (\curlywedge y.M)[z, y{:=}L, \Box] \\ & \overset{6}{=} & \curlywedge z.M \end{array}$$

The sequence starting with contraction of the outer redex of P runs as follows:

$$\begin{array}{rcl} P & \to_\beta & ((\lambda x.\curlywedge y.M)N)[z, y{:=}L, \Box] \\ & \overset{8,4}{=} & (\lambda x.\underline{(\curlywedge y.M)[z, y{:=}L, x]})N[z, y{:=}L, \Box] = P' \\ & \overset{6}{=} & (\lambda x.\underline{\curlywedge x.\curlywedge z.M})N[z, y{:=}L, \Box] \\ & \to_\beta & (\curlywedge x.\curlywedge z.M)[\Box, x{:=}N[z, y{:=}L, \Box], \Box] \\ & \overset{6}{=} & \curlywedge z.M \end{array}$$

Focus on the underlined substitution in the second sequence:

$$(\curlywedge y.M)[z, y{:=}L, x] = \curlywedge x.\curlywedge z.M$$

and note that it would be wrong to forget that $\curlywedge y$ implicitly closes the λx in front, and put $(\curlywedge y.M)[z, y{:=}L, x] \overset{\text{wrong!}}{=} \curlywedge z.M$. That this is wrong shows up if,

by coincidence, $x = z$. We would then get:

$$\begin{array}{rcl} P' & \overset{\text{wrong!}}{=} & (\lambda x.\lambda\!\!\!\backslash z.M)N[z, y{:=}L, \Box] \\ & \to_\beta & (\lambda\!\!\!\backslash z.M)[\Box, x{:=}N[z, y{:=}L, \Box], \Box] \\ & \overset{6}{=} & M \textit{ (assuming } x = z\textit{)} \end{array}$$

and confluence would be broken: $\lambda\!\!\!\backslash z.M \neq M$.

Remark 3.4.2 *The defining clauses $(5, 6, 7)$ for $(\lambda\!\!\!\backslash y.M)[X, x{:=}N, Y]$ are exhaustive, because if $y \in Y$, then $Y = JyY'$ for some J, Y' such that $y \notin J$. The implementation actually uses a nested recursion on Y with an auxiliary stack J (initially empty) recording the opening scopes in Y that are jumped (and closed) by $\lambda\!\!\!\backslash y$ (thus, the invariant is: $y \notin J$). Define $(\lambda\!\!\!\backslash y.M)[X, x{:=}N, Y] = (\lambda\!\!\!\backslash y.M)[X, x{:=}N, Y](\Box)$, where $(\lambda\!\!\!\backslash y.M)[X, x{:=}N, Y](J)$ is defined as follows.*

$$\begin{array}{rcll} (\lambda\!\!\!\backslash y.M)[X, x{:=}N, \Box](J) & = & \lambda\!\!\!\backslash J.\lambda\!\!\!\backslash X.M, \textit{ if } x = y & (J_1) \\ (\lambda\!\!\!\backslash y.M)[X, x{:=}N, \Box](J) & = & \lambda\!\!\!\backslash J.\lambda\!\!\!\backslash X.\lambda\!\!\!\backslash y.M, \textit{ if } x \neq y & (J_2) \\ (\lambda\!\!\!\backslash y.M)[X, x{:=}N, zY](J) & = & \lambda\!\!\!\backslash y.M[X, x{:=}N, Y], \textit{ if } y = z & (J_3) \\ (\lambda\!\!\!\backslash y.M)[X, x{:=}N, zY](J) & = & (\lambda\!\!\!\backslash y.M)[X, x{:=}N, Y](Jz), \textit{ if } y \neq z & (J_4) \end{array}$$

Note that z is inserted at the bottom of J in clause (J_4), to maintain the original order of scopes. The clauses for $\lambda\!\!\!\backslash y.M$ in Definition 3.4.3 are derived from these J-clauses.

Confluence of $\lambda\!\!\!\backslash$-calculus

We discuss our formalised proof of confluence of $\to_\beta$. Our proof strategy is the usual Tait and Martin-Löf proof [4], hence is essentially based on the so-called substitution lemma on page 27 of [4]:

> 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, in our case the substitution lemma splits into three lemmas: the *closed* substitution lemmas arise when the scope of y is ended (either explicitly or implicitly) in front of the λx; the *open* substitution lemma is the usual one, enriched with scoping information. We will comment on this below. Otherwise, the proof is entirely standard, (inductively) introducing multi-steps, proving that multi-steps have the diamond property and that β-reduction transits multi-steps.

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 (cf. the discussion of confluence of MELL proof net reduction in [46]).

Adapting the substitution lemma to our calculus, we end up simplifying expressions of the shape $(\lambda\!\!\!/X.M)[Z, x{:=}N, Y]$. First, to get an understanding of what is going on, consider the case of scope-balanced terms. Suppose that $\lambda\!\!\!/X.M$ is scope-balanced under YxW. Then we know, since end-of-scopes X 'balance' (a part of) YxW, that X and Y are *overlapping*, that is, either:

- Z exceeds Y, $X = YxX'$, then $(\lambda\!\!\!/X.M)[Z, x{:=}N, Y] = \lambda\!\!\!/YZX'.M$; or:
- X is part of Y, $Y = XY'$, then $(\lambda\!\!\!/X.M)[Z, x{:=}N, Y] = \lambda\!\!\!/X.M[Z, x{:=}N, Y']$.

Jump terms demand extra care and we need a different (more general) notion of comparison between opening scopes Y and closing scopes X. Consider, once more, $(\lambda\!\!\!/X.M)[Z, x{:=}N, Y]$. We distinguish three cases.

1. End-of-scopes X close more scopes than opened by Y, thus X includes the scope of the substitution variable x, which is closed either
   (a) explicitly; or
   (b) implicitly.
2. End-of-scopes X close some (possible all) of the open scopes Y.

To formalise this case distinction, we define *scope subtraction*. Subtraction $Y - X$ results in a pair of stacks, of which the first is either negative (item 1) or positive (the complementary case, item 2). The second stack of the pair computed by $Y - X$ is a stack J used only if the first stack is negative, say $-zX'$. In that case the substitution variable x is either matched (item 1a) or jumped (item 1b) by z, as will be shown below.

Definition 3.4.4 *Define $Y - X = Y -_{\Box} X$, where $X -_J Y$ is defined by the following clauses.*

$$\begin{array}{rcll} Y -_J \Box & = & (Y, J) & (1)\\ \Box -_J xX & = & (-xX, J) & (2)\\ yY -_J xX & = & Y -_{\Box} X, \text{ if } x = y & (3)\\ yY -_J xX & = & Y -_{Jy} xX, \text{ if } x \neq y & (4) \end{array}$$

The auxiliary stack J (initially empty) consists of the scopes in Y jumped by the current top of X. This can be inferred by clause (4), where y is inserted at the bottom of J, to maintain the original order of scopes. The argument J is reset to $\Box$ in case the top of X matches the top of Y (clause (3)). It's easy to see that if $Y - X$ is positive, then J is the empty stack.

Note that the idea of an auxiliary stack J is similar to the idea in Remark 3.4.2, only in a more general form to cope with $\lambda\!\!\!/X$ instead of $\lambda\!\!\!/y$. Indeed, clauses $(5, 6, 7)$ of Definition 3.4.3 can also be defined using scope-subtraction:

$$(\lambda\!\!\!/y.M)[Z, x{:=}N, Y] = \begin{cases} \lambda\!\!\!/y.M[Z, x{:=}N, Y'], \text{ if } Y - y = (Y', \Box)\\ \lambda\!\!\!/Y.\lambda\!\!\!/Z.M, \text{ if } Y - y = (-y, Y),\ x = y\\ \lambda\!\!\!/Y.\lambda\!\!\!/Z.\lambda\!\!\!/y.M, \text{ if } Y - y = (-y, Y),\ x \neq y \end{cases}$$

To see the equivalence, note that:

- if $y \in Y$, then $Y = JyY'$ for some J, Y' such that $y \notin J$, and $JyY' - y = yY' -_J y = (Y', \Box)$;
- if $y \notin Y$, then $Y - y = (-y, Y)$.

If X and Y are overlapping, clause (4) in Definition 3.4.4 never applies (cf. the case distinction for scope-balanced terms on page 69):

Lemma 3.4.1

$$\begin{array}{rcll} Y_1Y_2 - Y_1 & = & (Y_2, \Box) & (Y \geq X, Y = Y_1Y_2, X = Y_1) \\ X_1 - X_1xX_2 & = & (-xX_2, \Box) & (Y < X, Y = X_1, X = X_1xX_2) \end{array}$$

For arbitrary stacks X, Y, we can decide whether the outcome of subtracting X from Y is positive or negative:

Lemma 3.4.2 *The result $Y - X$ of scope-subtracting X from Y is either*

$$\begin{array}{l} (Y_2, \Box), \text{ and then } Y = Y_1Y_2; \text{ or} \\ (-zX_2, J), \text{ and then } X = X_1zX_2. \end{array}$$

Proof. By appropriately loaded induction over Y. We refer to our Coq formalisation for more details.

Now we are ready to simplify expressions $(\lambda\!\!\!\lambda X.M)[Z, x{:=}N, Y]$ based on the case distinction mentioned above.

Lemma 3.4.3 *If $Y - X_1zX_2 = (-zX_2, J)$ and $x = z$, then*

$$(\lambda\!\!\!\lambda X_1zX_2.M)[Z, x{:=}N, Y] = \lambda\!\!\!\lambda X_1JZ.\lambda\!\!\!\lambda X_2.M$$

Lemma 3.4.4 *If $Y - X_1zX_2 = (-zX_2, J)$ and $x \neq z$, then*

$$(\lambda\!\!\!\lambda X_1zX_2.M)[Z, x{:=}N, Y] = \lambda\!\!\!\lambda X_1JZzX_2.M$$

Lemma 3.4.5 *If $Y_1Y_2 - X = (Y_2, \Box)$, then*

$$(\lambda\!\!\!\lambda X.M)[Z, x{:=}N, Y_1Y_2] = \lambda\!\!\!\lambda X.M[Z, x{:=}N, Y_2]$$

Let us now present the substitution lemmas. We use the notation S^- to denote the reversal of a stack S, i.e. if $S = x_0 \dots x_n$, then $S^- = x_n \dots x_0$. In general, we want to compute the critical pair(s) from:

$$P = (\lambda\!\!\!\lambda Z.\lambda y.\lambda Y^-.(\lambda\!\!\!\lambda X.\lambda x.\lambda W^-.M)N)L$$

Inside-out reduction, $P \to^2_\beta P_{\mathsf{in,out}}$, gives:

$$P_{\mathsf{in,out}} = \lambda Y^-.\lambda W^-.M[X, x{:=}N, W][Z, y{:=}L, WY]$$

First β-reducing the outer redex, gives:

$$P_{\mathsf{out}} = \lambda Y^-.\underbrace{(\lambda\!\!\!\lambda X.\lambda x.\lambda W^-.M)[Z, y{:=}L, Y]}_{Q}N[Z, y{:=}L, Y]$$

We distinguish three cases for $P_{\mathsf{out}} \to_\beta P_{\mathsf{out,in}}$:

- $X = X_1zX_2$, $Y - X = (-zX_2, J)$ and $y = z$, then, by Lemma 3.4.3:

  $Q = \curlywedge X_1JZX_2.\lambda x.\lambda W^-.M$; and we get:
  $P_{\mathsf{out,in}} = \lambda Y^-.\lambda W^-.M[X_1JZX_2, x{:=}N[Z, y{:=}L, Y], W]$.

  Then, $P_{\mathsf{in,out}} = P_{\mathsf{out,in}}$ by Lemma 3.4.6.

- $X = X_1zX_2$, $Y - X = (-zX_2, J)$ and $y \neq z$, then by Lemma 3.4.4:

  $Q = \curlywedge X_1JZzX_2.\lambda x.\lambda W^-.M$; and we get:
  $P_{\mathsf{out,in}} = \lambda Y^-.\lambda W^-.M[X_1JZzX_2, x{:=}N[Z, y{:=}L, Y], W]$.

  Then, $P_{\mathsf{in,out}} = P_{\mathsf{out,in}}$ by Lemma 3.4.7.

- $Y = Y_1Y_2$ and $Y - X = (Y_2, \Box)$; then, by Lemma 3.4.5:

  $Q = \curlywedge X.\lambda x.\lambda W^-.M[Z, y{:=}L, WxY_2]$; and we get:
  $P_{\mathsf{out,in}} = \lambda Y^-.\lambda W^-.M[Z, y{:=}L, WxY_2][X, x{:=}N[Z, y{:=}L, Y_1Y_2], W]$.

  Then, $P_{\mathsf{in,out}} = P_{\mathsf{out,in}}$ by Lemma 3.4.8.

The *closed* substitution lemmas arise when the scope of y *is* ended by some (possibly implicit) $\curlywedge y$ in front of the λx, e.g. in $(\lambda y.(\curlywedge y.\lambda x.M)N)L$.

Example 3.4.2 *As an illustration, we compute the critical pair arising from* $P = (\lambda y.(\curlywedge y.\lambda x.M)N)L$. *If we start with the inner redex, we get:*

$$\begin{array}{rcl} P & \to_\beta & (\lambda y.M[y, x{:=}N, \Box])L \\ & \to_\beta & M[y, x{:=}N, \Box][\Box, y{:=}L, \Box] \end{array}$$

Performing the outer redex first:

$$\begin{array}{rcl} P & \to_\beta & ((\curlywedge y.\lambda x.M)N)[\Box, y{:=}L, \Box] \\ & = & (\curlywedge y.\lambda x.M)[\Box, y{:=}L, \Box]N[\Box, y{:=}L, \Box] \\ & = & (\lambda x.M)N[\Box, y{:=}L, \Box] \\ & \to_\beta & M[\Box, x{:=}N[\Box, y{:=}L, \Box], \Box] \end{array}$$

Note that the substitution for y *in* M *has disappeared from the right-hand side, corresponding to the erasing effect of the* $\curlywedge y$ *in front of it. Indeed,*

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

follows from Lemma 3.4.6.

If the scope of the substitution variable y is ended *explicitly* by some $\curlywedge y$ in front of the λx, the following lemma springs up.

Lemma 3.4.6 (Closed substitution lemma (match))

$$\begin{array}{l} M[X_1zX_2, x{:=}N, W][Z, y{:=}L, WY] \\ \quad = \quad M[X_1JZX_2, x{:=}N[Z, y{:=}L, Y], W], \text{ if } Y - X_1zX_2 = (-zX_2, J),\ y = z \end{array}$$

Remark 3.4.3 *By the previous lemma and Lemma 3.4.1 we obtain:*

$$\begin{aligned} & M[X_1 y X_2, x{:=}N, W][Z, y{:=}L, W X_1] \\ = \quad & M[X_1 Z X_2, x{:=}N[Z, y{:=}L, X_1], W] \end{aligned}$$

which is applicable when proving the multi-step substitution lemma for scope-balanced terms.

If the scope of the substitution variable y is ended *implicitly* by some $\nleftarrow z$ in front of the λx, the following lemma springs up.

Lemma 3.4.7 (Closed substitution lemma (jump))

$$\begin{aligned} & M[X_1 z X_2, x{:=}N, W][Z, y{:=}L, WY] \\ = \quad & M[X_1 J Z z X_2, x{:=}N[Z, y{:=}L, Y], W], \text{ if } Y - X_1 z X_2 = (-z X_2, J),\ y \neq z \end{aligned}$$

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 3.4.8 (Open substitution lemma) *If* $Y_1 Y_2 - X = (Y_2, \Box)$*, then*

$$\begin{aligned} & M[X, x{:=}N, W][Z, y{:=}L, W Y_1 Y_2] \\ = \quad & M[Z, y{:=}L, W x Y_2][X, x{:=}N[Z, y{:=}L, Y_1 Y_2], W] \end{aligned}$$

Remark 3.4.4 *By the previous lemma and Lemma 3.4.1 we obtain:*

$$\begin{aligned} & M[Y_1, x{:=}N, W][Z, y{:=}L, W Y_1 Y_2] \\ = \quad & M[Z, y{:=}L, W x Y_2][Y_1, x{:=}N[Z, y{:=}L, Y_1 Y_2], W] \end{aligned}$$

which is applicable when proving the multi-step substitution lemma for scope-balanced terms.

We introduce *multi-steps* that contract all β-redexes in a given term simultaneously.

Definition 3.4.5 Multi-steps $\multimap\!\!\!\circ\!\!\!\rightarrow$ *are defined by:*

$$\frac{M_1 \multimap\!\!\!\circ\!\!\!\rightarrow N_1 \quad M_2 \multimap\!\!\!\circ\!\!\!\rightarrow N_2}{(\nleftarrow X.\lambda x.M_1) M_2 \multimap\!\!\!\circ\!\!\!\rightarrow N_1[X, x{:=}N_2, \Box]}$$

$$\frac{}{x \multimap\!\!\!\circ\!\!\!\rightarrow x} \qquad \frac{M_1 \multimap\!\!\!\circ\!\!\!\rightarrow N_1 \quad M_2 \multimap\!\!\!\circ\!\!\!\rightarrow N_2}{M_1 M_2 \multimap\!\!\!\circ\!\!\!\rightarrow N_1 N_2}$$

$$\frac{M \multimap\!\!\!\circ\!\!\!\rightarrow N}{\lambda x.M \multimap\!\!\!\circ\!\!\!\rightarrow \lambda x.N} \qquad \frac{M \multimap\!\!\!\circ\!\!\!\rightarrow N}{\nleftarrow x.M \multimap\!\!\!\circ\!\!\!\rightarrow \nleftarrow x.N}$$

In a multi-step from M, multiple β-redexes in M may be contracted. In particular, β-redexes occurring in the *parallel* branches M_1 and M_2 of any application subterm M_1M_2 of M, may be contracted 'simultaneously' (cf. the single 'parallel' compatibility clause for application of $\multimap$ to the two 'sequential' compatibility clauses for application of $\to_\beta$ of Definition 3.4.2). All β-redexes occurring *nested* inside the body M_1 or argument M_2 of a redex subterm M' of M, may be contracted 'at the same time' as M' itself. (cf. the 'nested' β-redex *inference rule* of $\multimap$ to the β-redex *axiom* of $\to_\beta$.) Finally, note that the number of β-redexes may be zero, i.e. $M \multimap M$ for any term. As it turns out, it is enough to assume this for variables only (cf. the clause $x \multimap x$).

Remark 3.4.5 *Note that the relation $\multimap$ is* not *transitive. Contraction might create redexes not yet present in the starting term. e.g. we have $I^3 \multimap I^2 \multimap I$, but not $I^3 \multimap I$.*

The reason for the switch from single steps to multi-steps is that the former do not have the diamond property whereas the latter do. This is because contraction of a β-redex may replicate other redexes. Hence, for a notion of reduction extending β to possess the diamond property it must be 'closed under replication'. Multi-steps are just the least extension of single steps fitting the bill. At the technical level, closure under replication corresponds to the so-called substitution lemma:

Lemma 3.4.9 (Multi-step substitution lemma)

$$M \multimap M' \wedge N \multimap N' \Rightarrow M[Z, y{:=}N, Y] \multimap M'[Z, y{:=}N', Y]$$

Proof. By induction on $M \multimap M'$. In case $M = (\curlywedge X.\lambda x.M_1)M_2$, the proof obligation is:

$$(\curlywedge X.\lambda x.M_1)[Z, y{:=}N, Y]M_2[Z, y{:=}N, Y] \multimap M_1'[X, x{:=}M_2', \Box][Z, y{:=}N', Y]$$

where $M_1 \multimap M_1'$, $M_2 \multimap M_2'$ and $N \multimap N'$. The proof proceeds distinguishing cases in a similar way as on page 70, where we computed the general form of critical pairs and application of the substitution lemmas 3.4.6, 3.4.7 and 3.4.8.

Lemma 3.4.10 (Multi-step diamond property) *Multi-steps satisfy the diamond property.*

Proof. By induction on the diverging steps. All cases are trivial, except for the so-called coherence case when the starting term is a redex $(\curlywedge X.\lambda x.M)N$, and at least one step is a β-step.

- If both are β-steps, the result follows from the induction hypothesis, using the multi-step substitution lemma (Lemma 3.4.9).
- If only one of them is a β-step, then the results are $M_1[X, x{:=}N_1, \Box]$ and $(\curlywedge X.\lambda x.M_2)N_2$ respectively, for $M \multimap M_i$ and $N \multimap N_i$. By the

induction hypothesis $M_i \multimap M'$ and $N_i \multimap N'$ for some M' and N'. Hence the result follows since

$$M_1[X, x{:=}N_1, \Box] \multimap M'[X, x{:=}N', \Box]$$

by the multi-step substitution lemma (Lemma 3.4.9), and

$$(\lambda\!\!\!\lambda X.\lambda x.M_2)N_2 \multimap M'[X, x{:=}N', \Box]$$

by definition of $\multimap$.

Lemma 3.4.11 *$\to_\beta$ transits $\multimap$.*

Proof. By induction on the definitions of $\to_\beta$ and $\multimap$, respectively. The $\multimap \subseteq \to_\beta^*$ part follows from simulating inside-out developments of $\multimap$. For this one needs congruence of $\to_\beta^*$: if $M \to_\beta^* N$, then $C[M] \to_\beta^* C[N]$, which is proved by induction on unary contexts C.

Theorem 3.4.1 (Confluence of $\to_\beta$) *$\to_\beta$ is confluent on Λ.*

Proof. From Lemmas 3.2.2, 3.4.11 and 3.4.10.

3.4.3 α and β

We prove that α and β commute on scope-balanced terms, which is enough for present purposes. We are confident that commutation of α and β holds for *all* $\lambda\!\!\!\lambda$-terms, but leave this for future work. In particular, this would require more general formulations (using scope subtraction) of the substitution/renaming lemmas (Lemmas 3.4.16 and 3.4.17).

During the proof development, we experimented with all three α-equivalences $=_\alpha^{\mathsf{c}}$, $=_\alpha^{\mathsf{s}}$, and $=_\alpha^{\mathsf{k}}$. For the commutation lemma of $\to_\beta$ and $=_\alpha$ (Lemma 3.4.20), we took $=_\alpha^{\mathsf{s}}$ for $=_\alpha$. We first present some lemmas used in the proof of that lemma. If we rename the free occurrences of a variable x (and the matching occurrences of $\lambda\!\!\!\lambda x$) in α-equivalent terms M and N, the results are still α-equivalent:

Lemma 3.4.12 *$M =_\alpha^Z N \Rightarrow M[x{:=}z, Y] =_\alpha^Z N[x{:=}z, Y]$ if z fresh for M, N, Z.*

Proof. By induction on $M =_\alpha^Z N$, using Lemma 3.3.2 in the abstraction case.

In renaming expressions $M[x{:=}z]$ it is safe to replace z by a z' just as fresh:

Lemma 3.4.13 *$M[x{:=}z] =_\alpha^Z N[y{:=}z] \Rightarrow M[x{:=}z'] =_\alpha^Z N[y{:=}z']$, if z fresh for M, N and z' fresh for M, N, Z.*

Proof. Let $z \neq z'$ (the statement trivially holds if $z = z'$), then $z' \notin M[x{:=}z, Y]$ and $z' \notin N[y{:=}z, Y]$. By the previous lemma we obtain

$$M[x{:=}z, Y][z{:=}z', Y] =_\alpha^Z N[y{:=}z, Y][z{:=}z', Y]$$

Conclude by rewriting Lemma 3.3.3 twice.

The relation $=_\alpha^Z$ depends on the 'freshness' of Z and on Z being sufficiently long only:

Lemma 3.4.14 *If $|Z_2| \geq |Z_1|$, Z_2 fresh for M, N, Z_1, and all elements of Z_2 are distinct, then: $M =_\alpha^{Z_1} N \Rightarrow M =_\alpha^{Z_2} N$.*

Proof. By induction over $M =_\alpha^{Z_1} N$ and Lemma 3.4.13.

Lemma 3.4.14 solves the difficulty of proving transitivity of $=_\alpha^{\mathsf{s}}$ mentioned on page 57.

Lemma 3.4.15 *The relation $=_\alpha^{\mathsf{s}}$ is transitive.*

Proof. First prove that, for given Z, $=_\alpha^Z$ is transitive (*). Then, given $M =_\alpha^{Z_1} N =_\alpha^{Z_2} P$, choose Z_3 of length $\mathsf{max}(|Z_1|, |Z_2|)$ fresh for Z_1, Z_2, M, N, P. Then, by Lemma 3.4.14 we obtain $M =_\alpha^{Z_3} N =_\alpha^{Z_3} P$. Finally $M =_\alpha^{Z_3} P$ follows from (*).

Next, we present the substitution/renaming lemmas:

Lemma 3.4.16 (Open substitution/renaming lemma)

$$\begin{array}{ll} & M[X_1, y{:=}N, X_0][x{:=}z, X_0X_1X_2] \\ = & M[x{:=}z, X_0yX_2][X_1, y{:=}N[x{:=}z, X_1X_2], X_0] \end{array}$$

if z fresh for y, M, X_0.

Lemma 3.4.17 (Closed substitution/renaming lemma)

$$\begin{array}{ll} & M[X_1xX_2, y{:=}N, X_0][x{:=}z, X_0X_1] \\ = & M[X_1zX_2, y{:=}N[x{:=}z, X_1], X_0] \end{array}$$

Note that, for balanced terms, as we have that $M[x{:=}y, X] = M[y, x{:=}y, X]$, Lemmas 3.4.16 and 3.4.17 follow from Lemmas 3.4.6 and 3.4.8.

It is safe to rename in β-reductions:

Lemma 3.4.18 *If $z \notin M$ and M is scope-balanced under YxW, then $M \to_\beta N$ implies $M[x{:=}z, Y] \to_\beta N[x{:=}z, Y]$.*

Proof. Consider the case $(\lambda\!\!\!\lambda X.\lambda x.M_1)M_2 \to_\beta M_1[X, x{:=}M_2, \Box]$. Because of the assumption $\langle YxW\rangle(\lambda\!\!\!\lambda X.\lambda y.M_1)$, either X exceeds Y, and then the closed substitution/renaming lemma applies, or X is part of Y, and then the open substitution/renaming lemma applies.

If $\lambda x.M$ and $\lambda y.M'$ are α-equivalent (so also the outer (x- resp. y-)boxes have the same shape) and N and N' are α-equivalent, then the β-contractum of $(\lambda\!\!\!\lambda X.\lambda x.M)N$ is α-equivalent to the β-contractum of $(\lambda\!\!\!\lambda X.\lambda y.M')N'$:

Lemma 3.4.19

$$\begin{array}{ll} & M[x{:=}z, Y] =_\alpha^{Z_1} M'[y{:=}z, Y] \wedge N =_\alpha^{Z_2} N' \\ \Rightarrow & \exists Z_3.M[X, x{:=}N, Y] =_\alpha^{Z_3} M'[X, y{:=}N', Y] \end{array}$$

if Z_1 fresh for X, Y, Z_2, M, M', N, N', and z fresh for M, M'.

On scope-balanced terms, β-reduction and α-equivalence commute (Schema E in Figure 3.4). Here, we take $=_\alpha^{\mathsf{s}}$ for $=_\alpha$.

Lemma 3.4.20 *If $\langle X\rangle M$, $M \to_\beta N$, and $M =_\alpha^Z M'$, then there exists a term N' and a stack Z' such that $M' \to_\beta N'$ and $N =_\alpha^{Z'} N'$.*

Proof. By induction on the skeleton of M. We show some interesting cases. Recall that renaming doesn't alter the skeleton of a term and that scope-balancedness is closed under $=_\alpha$.

- Case $\lambda x.M_0 \to_\beta \lambda x.N_0$. By inverting $\lambda x.M_0 =_\alpha^Z M'$, we obtain $M' = \lambda y.M_0'$ and $Z = zZ_0$ such that $M_0[x{:=}z] =_\alpha^{Z_0} M_0'[y{:=}z]$ for $z \notin M_0, M_0', Z_0$. By Lemma 3.4.18, we have $M_0[x{:=}z] \to_\beta N_0[x{:=}z]$. Then, by the induction hypothesis, there exist P and Z_0' such that $M_0'[y{:=}z] \to_\beta P$ and $N_0[x{:=}z] =_\alpha^{Z_0'} P$. We are able prove that, for some N_0', $P = N_0'[y{:=}z]$ and $M_0' \to_\beta N_0'$. Choose z' fresh for N_0, N_0', Z_0' (the Fresh variable axiom (Axiom 3.2) guarantees the existence of z'). From Lemma 3.4.13, we obtain $N_0[x{:=}z'] =_\alpha^{Z_0'} N_0'[x{:=}z']$ ($\to_\beta$ doesn't introduce new names, therefore $z \notin N_0, N_0'$ follows from $z \notin M_0, M_0'$). Then, $N' = \lambda y.N_0'$ and $Z' = z'Z_0'$ witness the goal $\exists N'.\exists Z'.\lambda y.M_0' \to_\beta N' \wedge \lambda x.N_0 =_\alpha^{Z'} N'$.

- Case $M_1M_2 \to_\beta N_1M_2$. So $M_1 \to_\beta N_1$, $M' = M_1'M_2'$, $M_1 =_\alpha^{Z_1} M_1'$, and $M_2 =_\alpha^{Z_1} M_2'$. By the induction hypothesis, there exist N_1' and Z_2 such that $M_1' \to_\beta N_1'$ and $N_1 =_\alpha^{Z_2} N_1'$. By Lemma 3.4.14, we obtain $\exists Z'.M_1'M_2' =_\alpha^{Z'} N_1'M_2'$.

- Case $(\lambda\!\!\!\downarrow X.\lambda x.M_1)M_2 \to_\beta M_1[X, x{:=}M_2, \Box]$. Derive $M' = (\lambda\!\!\!\downarrow X.\lambda y.M_1')M_2'$, $M_1[x{:=}z] =_\alpha^Z M_1'[y{:=}z]$, and $M_2 =_\alpha^{zZ} M_2'$ with $z \notin M_1, M_2, Z$. Choose Z_0 fresh for $M_1, M_2, M_1', M_2', X, Z$ and such that $|Z_0| = |Z|$ (apply the Fresh variable axiom (Axiom 3.2) $|Z_0|$ times). Then Z_0 is provably fresh for $M_1[x{:=}z]$ and $M_1'[y{:=}z]$ as well. By Lemma 3.4.14 we obtain $M_1[x{:=}z] =_\alpha^{Z_0} M_1'[y{:=}z]$ Take $N' = M_1'[X, y{:=}M_2', \Box]$ as witness for the existential statement we are proving. Finally, by Lemma 3.4.19, there exists Z' such that $M_1[X, x{:=}M_2, \Box] =_\alpha^{Z'} M_1'[X, y{:=}M_2', \Box]$.

3.4.4 Confluence of λ-calculus

As a corollary we obtain confluence of the ordinary λ-calculus (see Figure 3.4). The exposition of the proof proceeds in a top-down fashion, forward referring to lifting and projection lemmas. We use $\to_{\lambda\beta}$ and $\to_{\lambda\!\!\!\downarrow\beta}$ to distinguish β-reduction in the λ-calculus from β-reduction in the $\lambda\!\!\!\downarrow$-calculus, respectively.

Theorem 3.4.2 *$\to_{\lambda\beta}$ is confluent up to $=_\alpha$.*

Proof.

1. Consider two diverging $\lambda\beta$-reductions $M \to_{\lambda\beta}^* N$ and $M \to_{\lambda\beta}^* P$.

2. Lift these stepwise to diverging $\lambda\beta$-reductions $M \twoheadrightarrow^*_{\lambda\beta} N'$ and $M \twoheadrightarrow^*_{\lambda\beta} P'$ (Lemma 3.4.27). (Note that M being a λ-term, it is a scope-balanced λ-term.)

3. By confluence of $\lambda\beta$-reduction, we can find some λ-term Q' such that $N' \twoheadrightarrow^*_{\lambda\beta} Q'$, $P' \twoheadrightarrow^*_{\lambda\beta} Q'$ (Theorem 3.4.1).

4. Projecting $N' \twoheadrightarrow^*_{\lambda\beta} Q'$ and $P' \twoheadrightarrow^*_{\lambda\beta} Q'$ back to $\lambda\beta$-reduction yields $N \twoheadrightarrow^*_{\lambda\beta} Q_1$ and $P \twoheadrightarrow^*_{\lambda\beta} Q_2$ (Lemma 3.4.28), for some α-equivalent λ-terms Q_1 and Q_2 (Corollary 3.4.1), establishing the desired confluence of $\lambda\beta$ up to α-equivalence.

Remark 3.4.6 *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 [68]. However, their proof technique is entirely different, uniquely renaming all variables, before performing β-steps, whereas our schema, which works via the λ-calculus, only performs the necessary updates (in the sense of [25]).*

Lifting and Projection of β-reduction

Projection of λ-terms to λ-terms is the composition of first performing an α-equivalence step followed by a so-called ω-step removing all λs in one go.[8] For instance, no ω-step is possible from $\lambda x.\lambda x.x$ since removing λx 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 could be performed. However, we rename only if necessary.

Definition 3.4.6 *We define $M \multimap\!\!\bullet\!\!\to_\omega N$, if $\langle\Box\rangle M \multimap\!\!\bullet\!\!\to_\omega N$, where $\langle X\rangle M \multimap\!\!\bullet\!\!\to_\omega N$ is defined by the following clauses.*

$$\frac{}{\langle X\rangle x \multimap\!\!\bullet\!\!\to_\omega x} \qquad \frac{\langle xX\rangle M \multimap\!\!\bullet\!\!\to_\omega M'}{\langle X\rangle \lambda x.M \multimap\!\!\bullet\!\!\to_\omega \lambda x.M'}$$

$$\frac{\langle X\rangle M_1 \multimap\!\!\bullet\!\!\to_\omega N_1 \quad \langle X\rangle M_2 \multimap\!\!\bullet\!\!\to_\omega N_2}{\langle X\rangle M_1 M_2 \multimap\!\!\bullet\!\!\to_\omega N_1 N_2} \qquad \frac{\langle X\rangle M \multimap\!\!\bullet\!\!\to_\omega M' \quad x \notin \mathsf{FV}(M)}{\langle xX\rangle \lambda x.M \multimap\!\!\bullet\!\!\to_\omega M'}$$

Thus, ω-steps are maximal, in the sense that one ω-step removes all λs in one go. We write $\langle X\rangle M =^{\mathsf{k}}_\alpha \langle Y\rangle N \multimap\!\!\bullet\!\!\to_\omega P$ to abbreviate $\langle X\rangle M =^{\mathsf{k}}_\alpha \langle Y\rangle N \wedge \langle Y\rangle N \multimap\!\!\bullet\!\!\to_\omega P$, and, conversely, we write $\langle X\rangle M \multimap\!\!\bullet\!\!\to_\omega N =^{\mathsf{k}}_\alpha \langle Y\rangle P$ to abbreviate $\langle X\rangle M \multimap\!\!\bullet\!\!\to_\omega N \wedge \langle X\rangle N =^{\mathsf{k}}_\alpha \langle Y\rangle P$. Note that the source of the $\multimap\!\!\bullet\!\!\to_\omega$ is, by definition, forced to be scope-balanced:

$$\langle X\rangle M \multimap\!\!\bullet\!\!\to_\omega N \Rightarrow \langle X\rangle M$$

Also note that $\multimap\!\!\bullet\!\!\to_\omega$ doesn't alter the set of free variables:

$$\langle XY\rangle M \multimap\!\!\bullet\!\!\to_\omega M' \Rightarrow \mathsf{FV}(M, X) = \mathsf{FV}(M', X)$$

[8] ω could be decomposed itself by first pushing λs to the variables, i.e. performing *maximal* scope extrusion before omitting λs.

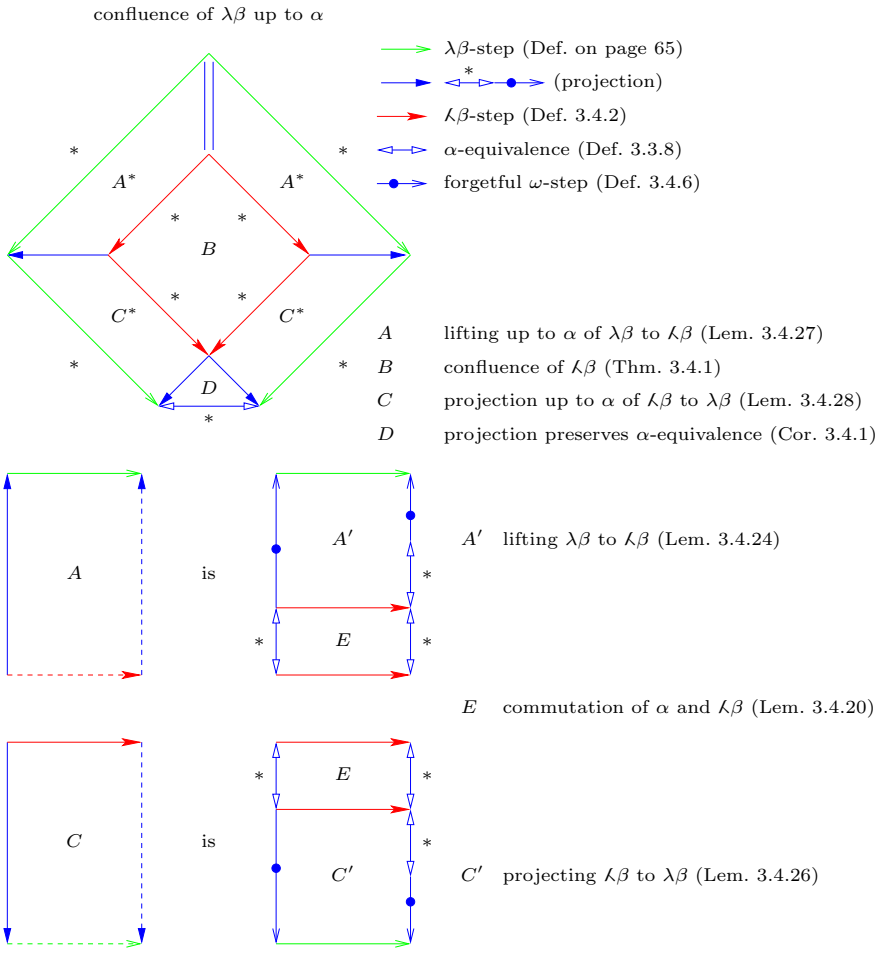

Figure 3.4: Confluence of ƛ-calculus implies confluence of λ-calculus.

Example 3.4.3 *If we first rename the bound xs in $\lambda x.\curlywedge x.x$, by some $x' \neq x$, then it is safe to forget the $\curlywedge x'$: $\langle x\rangle\lambda x.\curlywedge x.x =^{\mathsf{k}}_{\alpha} \langle x\rangle\lambda x'.\curlywedge x'.x \twoheadrightarrow_{\omega} \lambda x'.x$.*

Example 3.4.4 *It is incorrect to forget end-of-scopes in the jump calculus, as witnessed by the $\curlywedge$-term $\lambda x.\lambda y.\curlywedge x.y$. The variable y is free in this term since $\curlywedge x$ implicitly closes the scope of y. However, forgetting this end-of-scope would yield the λ-term $\lambda x.\lambda y.y$ where y is bound. The easiest way to proceed seems to be to insert as many $\curlywedge$s as are needed to make the $\curlywedge$-term scope-balanced:*

$$\begin{array}{rcl}
\mathsf{scb}(x, X) & = & x \\
\mathsf{scb}(\lambda x.M, X) & = & \lambda x.\mathsf{scb}(M, xX) \\
\mathsf{scb}(\curlywedge x.M, \Box) & = & \mathsf{scb}(M, \Box) \\
\mathsf{scb}(\curlywedge x.M, yX) & = & \curlywedge x.\mathsf{scb}(M, X) \text{ if } x = y \\
\mathsf{scb}(\curlywedge x.M, yX) & = & \curlywedge y.\mathsf{scb}(\curlywedge x.M, X) \text{ if } x \neq y \\
\mathsf{scb}(M_1M_2, X) & = & \mathsf{scb}(M_1, X)\mathsf{scb}(M_2, X)
\end{array}$$

Indeed, for all terms M and stacks X, $\mathsf{scb}(M, X)$ is scope-balanced under X. Applied to the example, we first obtain $\mathsf{scb}(\lambda x.\lambda y.\curlywedge x.y, \Box) = \lambda x.\lambda y.\curlywedge y.\curlywedge x.y$. Now we see that in order to omit the $\curlywedge y$, we have to rename it first, say to z yielding $\lambda x.\lambda z.\curlywedge z.\curlywedge x.y$. Forgetting end-of-scopes now yields the (correct) λ-term $\lambda x.\lambda z.y$.

Remark 3.4.7 *In $\lambda\beta$-reduction renamings are performed, as soon as there is 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 $\curlywedge\beta$-reduction:

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

Observe that despite the final term of this $\curlywedge\beta$-reduction being an ordinary λ-term, α-conversion is *needed to project it (see Lemma 3.4.28).*

The relation $\twoheadrightarrow_{\omega}$ preserves α-equivalence:

Lemma 3.4.21 *$N \ {}_{\omega}\!\!\twoheadleftarrow \langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle X'\rangle M' \twoheadrightarrow_{\omega} N'$ implies $\langle X\rangle N =^{\mathsf{k}}_{\alpha} \langle X'\rangle N'$.*

Proof. By induction on the proposition $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle X'\rangle M'$. We show the case $\langle xX\rangle\curlywedge x.M =^{\mathsf{k}}_{\alpha} \langle yX'\rangle\curlywedge y.M'$. Then, $\langle X\rangle M \twoheadrightarrow_{\omega} N$, $x \notin \mathsf{FV}(M)$, $\langle X'\rangle M' \twoheadrightarrow_{\omega} N'$, and $y \notin \mathsf{FV}(M')$. Inversion gives $\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle X'\rangle M'$. By the induction hypothesis we have $\langle X\rangle N =^{\mathsf{k}}_{\alpha} \langle X'\rangle N'$, which, by the following lemma, implies the goal, $\langle xX\rangle N =^{\mathsf{k}}_{\alpha} \langle yX'\rangle N'$, because N, N' are free of $\curlywedge$s.

Corollary 3.4.1 *Schema D in Figure 3.4, stating that $Q' =_{\alpha} Q'_1 \twoheadrightarrow_{\omega} Q_1$ and $Q' =_{\alpha} Q'_2 \twoheadrightarrow_{\omega} Q_2$ imply $Q_1 =_{\alpha} Q_2$, now easily follows: first show that $Q'_1 =_{\alpha} Q'_2$ (by symmetry and transitivity of $=_{\alpha}$) and then apply the previous lemma.*

Lemma 3.4.22 *If M is a λ-term, that is, M contains no $\lambda\!\!\!\lambda$s, and $x \notin \mathsf{FV}(M)$, then*

$$\langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N \text{ implies } \langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \text{ and } y \notin \mathsf{FV}(N)$$

Conversely, if $x \notin \mathsf{FV}(M)$ and $y \notin \mathsf{FV}(N)$, then

$$\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \text{ implies } \langle xX\rangle M =^{\mathsf{k}}_{\alpha} \langle yY\rangle N$$

Given a sequence of $-\!\!\bullet\!\!\rightarrow_\omega$-steps and α-steps, the $-\!\!\bullet\!\!\rightarrow_\omega$-steps can always be postponed until the α-steps are performed:

Lemma 3.4.23 $\langle X\rangle M -\!\!\bullet\!\!\rightarrow_\omega P =^{\mathsf{k}}_{\alpha} \langle Y\rangle N \Rightarrow \exists Q.\langle X\rangle M =^{\mathsf{k}}_{\alpha} \langle Y\rangle Q -\!\!\bullet\!\!\rightarrow_\omega N.$

Proof. By induction on the definition of $-\!\!\bullet\!\!\rightarrow_\omega$. Consider case $\langle xX\rangle \lambda\!\!\!\lambda x.M -\!\!\bullet\!\!\rightarrow_\omega P =^{\mathsf{k}}_{\alpha} \langle yY\rangle N$. Then $x \notin \mathsf{FV}(M)$ and $\langle X\rangle M -\!\!\bullet\!\!\rightarrow_\omega P$. Because $-\!\!\bullet\!\!\rightarrow_\omega$ doesn't change the set of free variables, we have that $x \notin \mathsf{FV}(P)$. By Lemma 3.4.22, we get $\langle X\rangle P =^{\mathsf{k}}_{\alpha} \langle Y\rangle N$ and $y \notin \mathsf{FV}(N)$. By the induction hypothesis, we have Q such that $\langle M\rangle X =^{\mathsf{k}}_{\alpha} \langle Y\rangle Q -\!\!\bullet\!\!\rightarrow_\omega N$. $y \notin \mathsf{FV}(Q)$ follows and we obtain $\langle xX\rangle \lambda\!\!\!\lambda x.M =^{\mathsf{k}}_{\alpha} \langle yY\rangle \lambda\!\!\!\lambda y.Q -\!\!\bullet\!\!\rightarrow_\omega N$.

Both projection and lifting of reductions are performed stepwise. That is, a single $\lambda\beta$-step lifts to a single $\lambda\!\!\!\lambda\beta$-step and vice versa (not to reduction sequences, as in calculi with explicit substitutions). Lifting of $\rightarrow_{\lambda\beta}$ to $\rightarrow_{\lambda\!\!\!\lambda\beta}$ (Schema A' in Figure 3.4) is stated as follows.

Lemma 3.4.24 *If $M \rightarrow_{\lambda\beta} N$ and $\langle X\rangle M' -\!\!\bullet\!\!\rightarrow_\omega M$, then there are N_1, N_2 such that:*

$$\langle X\rangle N_1 =^{\mathsf{k}}_{\alpha} \langle X\rangle N_2 -\!\!\bullet\!\!\rightarrow_\omega N \text{ and } M' \rightarrow_{\lambda\!\!\!\lambda\beta} N_1$$

Proof. As an illustration, consider the case $M = (\lambda x.M_1)M_2$, and $M' = L_1L_2$. We have $\langle X\rangle L_1 -\!\!\bullet\!\!\rightarrow_\omega \lambda x.M_1$ and $\langle X\rangle L_2 -\!\!\bullet\!\!\rightarrow_\omega M_2$. By inversion, we obtain $L_1 = \lambda\!\!\!\lambda X_1.\lambda x.L_1'$, $X = X_1X_2$, $X_1 \cap \mathsf{FV}(\lambda x.L_1') = \emptyset$ and $\langle xX_2\rangle L_1' -\!\!\bullet\!\!\rightarrow_\omega M_1$. The proof obligation is

$$\exists N_1, N_2.\langle X\rangle N_1 =^{\mathsf{k}}_{\alpha} \langle X\rangle N_2 -\!\!\bullet\!\!\rightarrow_\omega M_1[x{:=}M_2] \wedge (\lambda\!\!\!\lambda X_1.\lambda x.L_1')L_2 \rightarrow_{\lambda\!\!\!\lambda\beta} N_1$$

Take $N_1 = L_1'[X_1, x{:=}L_2, \Box]$. The following lemma (Lemma 3.4.25) guarantees the existence of an $\lambda\!\!\!\lambda$-term P such that

$$\langle X\rangle L_1'[X_1, x{:=}L_2, \Box] =^{\mathsf{k}}_{\alpha} \langle X\rangle P[X_1, x{:=}L_2, \Box] -\!\!\bullet\!\!\rightarrow_\omega M_1[x{:=}M_2]$$

The witnessing $N_2 = P[X_1, x{:=}L_2, \Box]$ solves our goal.

The following lemma states projection of $\lambda\!\!\!\lambda$-substitution to λ-substitution.

Lemma 3.4.25

$$\begin{array}{lll} \langle xZ\rangle M_1 -\!\!\bullet\!\!\rightarrow_\omega M & & \\ \quad \wedge & \langle XZ\rangle N' -\!\!\bullet\!\!\rightarrow_\omega N & \\ \quad \wedge & X \cap \mathsf{FV}(\lambda x.M_1) = \emptyset & \\ \quad \Rightarrow & \exists P{:}\Lambda\!\!\!\lambda.\langle XZ\rangle M_1[X, x{:=}N', \Box] =^{\mathsf{k}}_{\alpha} \langle XZ\rangle P[X, x{:=}N', \Box] -\!\!\bullet\!\!\rightarrow_\omega M[x{:=}N] \end{array}$$

Proof. The difficult part was to find the right induction loading:

$$\begin{array}{lll}
\langle Y_1 x Z\rangle M_1 =^{\mathsf{k}}_{\alpha} \langle Y_2 x Z\rangle M_2 \twoheadrightarrow_{\omega} M & & \\
\quad \wedge & \langle X Z\rangle N' \twoheadrightarrow_{\omega} N & \\
\quad \wedge & X \cap \mathsf{FV}(M_1, xY_1) = \emptyset & \\
\quad \wedge & Y_2 \cap (\{x\} \cup \mathsf{FV}(N')) = \emptyset & \\
\quad \Rightarrow & \exists P : \Lambda. \langle Y_1 X Z\rangle M_1[X, x{:=}N', Y_1] =^{\mathsf{k}}_{\alpha} \langle Y_2 X Z\rangle P[X, x{:=}N', Y_2] & \\
& \qquad \twoheadrightarrow_{\omega} M[x{:=}N] &
\end{array}$$

Once appropriately loaded, the proof is a straightforward induction over M_1, the only interesting lemma used being Lemma 3.4.23.

Projecting $\rightarrow_{\lambda\!\!\!/\beta}$ to $\rightarrow_{\lambda\beta}$ (Schema C' in Figure 3.4) is stated as follows.

Lemma 3.4.26 *If $M \rightarrow_{\lambda\!\!\!/\beta} N$ and $\langle X\rangle M \twoheadrightarrow_{\omega} M'$, then there are N_1, N_2 such that:*

$$\langle X\rangle N =^{\mathsf{k}}_{\alpha} \langle X\rangle N_1 \twoheadrightarrow_{\omega} N_2 \text{ and } M' \rightarrow_{\lambda\beta} N_2$$

Proof. The β-rule case calls Lemma 3.4.25 again.

Lifting of $\lambda\beta$-reduction sequences to $\lambda\!\!\!/\beta$-reduction sequences (Schema A^* in Figure 3.4) is stated by the following lemma.

Lemma 3.4.27

$$\begin{array}{l}
M_1 \rightarrow^*_{\lambda\beta} M_2 \wedge P_1 =^{\mathsf{k}}_{\alpha} Q_1 \twoheadrightarrow_{\omega} M_1 \\
\quad \Rightarrow \quad \exists P_2, Q_2 : \Lambda. P_2 =^{\mathsf{k}}_{\alpha} Q_2 \twoheadrightarrow_{\omega} M_2 \wedge P_1 \rightarrow^*_{\lambda\!\!\!/\beta} P_2
\end{array}$$

Proof. By Lemmas 3.4.24 and 3.4.20 single $\lambda\beta$-steps can be lifted to single $\lambda\!\!\!/\beta$-steps (Schema A in Figure 3.4). The result for sequences follows by reflexively, transitively closing the single step case.

Projection of $\lambda\!\!\!/\beta$-reduction sequences to $\lambda\beta$-reduction sequences (Schema C^* in Figure 3.4) is stated by the following lemma.

Lemma 3.4.28

$$\begin{array}{l}
P_1 \rightarrow^*_{\lambda\!\!\!/\beta} P_2 \wedge P_1 =^{\mathsf{k}}_{\alpha} Q_1 \twoheadrightarrow_{\omega} M_1 \\
\quad \Rightarrow \quad \exists M_2 : \Lambda. \exists Q_2 : \Lambda. P_2 =^{\mathsf{k}}_{\alpha} Q_2 \twoheadrightarrow_{\omega} M_2 \wedge M_1 \rightarrow^*_{\lambda\beta} M_2
\end{array}$$

Proof. By Lemmas 3.4.26 and 3.4.20 single $\lambda\!\!\!/\beta$-steps can be lifted to single $\lambda\beta$-steps (Schema C in Figure 3.4). The result for sequences follows by reflexively, transitively closing the single step case.

3.5 Applications

We think that the $\lambda\!\!\!/$-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.

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:

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

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. [28], our $\mathcal{K}$-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:

$$\lambda x.(\mathcal{K}x.M)x \to M,$$

and the β-redex becomes $(\lambda x.\mathcal{K}x.M)N$, which indeed executes more efficiently. In [25] some statistical evidence is presented that this is a frequently occurring situation, i.e. that it is worthwhile to retain scoping information when evaluating ordinary λ-terms.

In the next section we present some further evidence to the usefulness of the $\mathcal{K}$-calculus.

3.6 Conclusion and Discussion

The reification of scopes provides a fundamental understanding of scoping mechanisms of *named*[9] terms. Our results can be summarised as follows.

- Confluence of the $\mathcal{K}$-calculus *without* α-conversion.
- Scope information is retained, possibly *speeding up* β-reduction.
- Free variable conditions are expressable in the object language.
- Unintended capture of free variables in the substituens by binders in the substitution body is avoided, not by renaming the binders, but by prefixing the substituens by $\mathcal{K}$s ending the scope of those binders. Because the $\mathcal{K}$s are not pushed to the variables (we don't perform *maximal* scope extrusion, that is), the transformation of arguments during substitution is avoided; thus, they can still be subject to sharing.
- α-conversion is a decidable congruent equivalence.

Restricting to a single name, the $\mathcal{K}$-calculus corresponds to the λ-calculus formalised using a generalisation of De Bruijn indices, with the *shift substitution* $[\uparrow]$ (cf. $\mathcal{K}$) as explicit term constructor (see the paragraph on related work in Section 3.1).

[9] Names are more pleasant for human beings (e.g. for debugging purposes), and we want our pen-and-paper proofs to be formalisable in a direct way. Moreover, to be user-friendly, implementations must use names, either internally or just for parsing and printing.

By lifting β-reduction of the λ-calculus to β-reduction of the $\curlywedge$-calculus, and projecting back, we can analyse more precisely renamings performed 'on the fly' by β-reduction in the λ-calculus. Confluence (up to α) of the λ-calculus is obtained as a derived result.

We conclude this chapter by discussing two potential applications of the $\curlywedge$-calculus we are currently investigating. In fact, it were these two applications which have led us to the discovery of the calculus.

Explicit substitution calculi The first part of this work arose from trying to understand Chapter 4 of [17] on perpetuality in David and Guillaume's calculus with explicit substitutions λ_{ws}, in a named setting, cf. [26], and in an atomic way. David and Guillaume introduce the λw-calculus as:

> We avoid the counter-example to the PSN property of the λs_e-calculus by adding to the usual syntax a new constructor that we call a *label* and which represents an updating information. The term t with label k (denoted by $\langle k \rangle t$) corresponds to the term t where all free indices have been increased by k (i.e. $\phi_0^k(t)$ in λs_e).
>
> In the terms we are finally interested in, two successive labels are not allowed. We first define preterms without this restriction.
>
> **Definition 3.1.** We define the set of λ_w-preterms by the following grammar:
>
> $$t ::= \underline{n} \mid \lambda t \mid (t\ t) \mid \langle k \rangle t \quad \text{with} \quad n, k \in \mathbb{N}$$

Observing that labels can be seen as repetitions of successors should make the relationship to the $\curlywedge$-calculus clear.[10] In [26] a named version of λ_w is presented having sets of names as labels. Our approach is more elementary as their labels are added only *after* α-conversion (not in their abstract syntax, but in their calculus), so they cannot get confluence of β-reduction without it.

Application: Preservation of Strong Normalisation The λ_{ws} calculus was introduced as a calculus having, among other desirable properties, the preservation of strong normalisation (PSN) property. From [25] 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 $\curlywedge$-calculus offers an easy insight as to *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 page 68). 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

[10] The relationship is not entirely trivial since in their λ_w calculus, David and Guilaume make use of commutativity and associativity of addition of natural numbers, whereas we are only allowed to manipulate *stacks* of names. However, one may reformulate their rules such that commutativity and associativity are not needed.

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):

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

Now we *recognise* that the two underlined xs in the right-hand side 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 (roughly speaking the maximal length of a reduction can be bounded by the length of a standard reduction, in the spirit of [44]). Applying this idea to the closed and open substitution lemmas (Lemmas 3.4.6 and 3.4.8) should give rise to named versions of the following two λ_{ws} rules, see e.g. page 65 of [17]:

$$\begin{array}{lll} M[k/N,l][\![i/P,j]\!] & \to_{lc1} & M[k/N[\![i-k/P,j]\!],j+l-1] \qquad k \le i < k+l \\ M[k/N,l][\![i/P,j]\!] & \to_{lc2} & M[\![i-l+1/P,j]\!][k/N[\![i-k/P,j]\!],l] \qquad k+l \le i \end{array}$$

which should yield a named version of λ_{ws} having PSN.

Localising scope extrusion The second part of this work arose from trying to understand Coppola's PhD thesis [24] on the (complexity of) an optimal implementation of the λ-calculus. The idea is to view the boxes featuring in that work as our boxes, i.e. as representing scoping information, and to view their optimal implementation as a *local* implementation of our (minimal) scope extrusion.

Application: optimal reduction Lamping provided in [47] the first implementation of the λ-calculus which was optimal in the sense of Lévy [48]. 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 Chapter 9 of [3].

We claim that the oracle can be understood to arise from making β-reduction in the ƛ-calculus *local* in the sense of [45]. That is scope extrusion and x-box removal as in Figure 3.3 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 3.5. In fact, a key insight (cf. the second step of Figure 3.5) 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 3.5 for

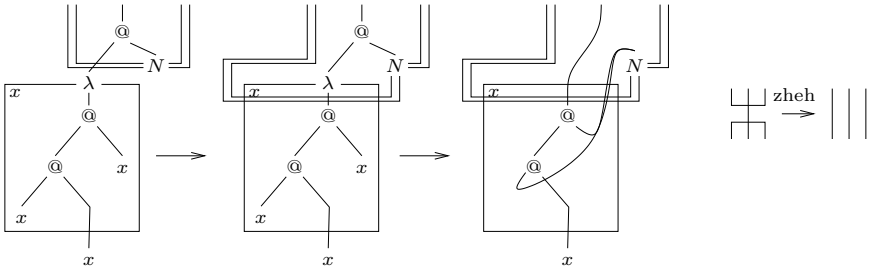

Figure 3.5: Left: β-reduction: local scope extrusion and rewiring. Right: scope fusion.

fusing two adjacent scopes. The implementation performs well on the examples in [3], 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 Figure 9.23 of [3], takes us roughly 5 times as many interactions (compared to BOHM 1.1).[11]

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