Calculi of Generalised β-Reduction and Explicit Substitutions: The Type Free and Simply Typed Versions[*]

Fairouz Kamareddine and Alejandro Ríos

University of Glasgow
Department of Computing Science
17 Lilybank Gardens
Glasgow G12 8QQ, Scotland
FAX: +44 141 330 4913
{fairouz,rios}@dcs.gla.ac.uk
http://www.dcs.gla.ac.uk/research/beauty

J. B. Wells

Boston University
College of Arts and Sciences
Department of Computer Science
111 Cummington Street, Room 138
Boston, MA 02215, U.S.A.
FAX: +1 617 353 6457
jbw@cs.bu.edu
http://www.cs.bu.edu/~jbw

April 11, 1997

Abstract

Extending the λ-calculus with either explicit substitution or generalised reduction has been the subject of extensive research recently and still has many open problems. This paper is the first investigation into the properties of a calculus combining both generalised reduction and explicit substitutions. We present a calculus, λgs, that combines a calculus of explicit substitution, λs, and a calculus with generalized reduction, λg. We believe that λgs is a useful extension of the λ-calculus because it allows postponment of work in two different but complementary ways. Moreover, λgs (and also λs) satisfies desirable properties of calculi of explicit substitutions and generalised reductions. In particular, we show that λgs preserves strong normalisation, is a conservative extension of λg, and simulates β-reduction of λg and the classical λ-calculus. Furthermore, we study the simply typed versions of λs and λgs and show that well typed terms are strongly normalising and that other properties such as typing of subterms and subject reduction hold. Our proof of the preservation of strong normalisation (PSN) is based on the minimal derivation method. It is however much simpler because we prove the commutation of arbitrary internal and external reductions. Moreover, we use one proof to show both the preservation of λ-strong normalisation in λs and the preservation of λg-strong normalisation in λgs. We remark that the technique of these proofs is not suitable for calculi without explicit substitutions (e.g. the preservation of λ-strong normalisation in λg requires a different technique).

1 Introduction

1.1 The λ-calculus with generalized reduction

In the term $((\lambda_x.\lambda_y.N)P)Q$, the abstraction starting with λ_x and the argument P form the redex $(\lambda_x.\lambda_y.N)P$. When this redex is contracted, the abstraction starting with λ_y and Q will in turn form a redex. What is important is that the only argument the abstraction starting with λ_y (or some residual of this abstraction) can ever have is Q (or some residual of Q). This fact has been exploited by many researchers and reduction has been extended so that the implicit redex based on the matching λ_y and Q is given the same priority as the intervening redex.

An initial attempt to generalize the notion of redex might be to define a rule like the following:

$$(\lambda_x.\lambda_y.N)PQ \to (\lambda_x.N[y{:=}Q])P$$

[*]This work is supported by EPSRC grants GR/K25014 and GR/L36963 and by NSF grant CCR-9417382.

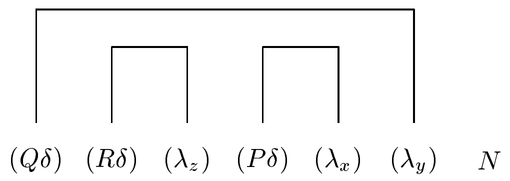
It quickly becomes evident that this is not sufficient. For example, the proposed rule does not allow directly reducing the binding of y to Q in the term $A \equiv (\lambda_z.(\lambda_x.\lambda_y.N)P)RQ$. We shall exploit the notion of a *well balanced segment* (sometimes known as a *β-chain*), which is the special case of one-hole contexts given by this grammar:

$$S ::= [\,\cdot\,] \mid (S[\lambda_x.[\,\cdot\,]])M \mid S[S]$$

Using balanced segments, *generalized reduction* is then given by this rule:

$$S[\lambda_x.M]N \to S[M[x{:=}N]]$$

We find the above definition of well-balanced segments and generalised reduction rather cumbersome and believe that a more elegant definition can be given. In order to do so, we change from the classical notation to the *item notation*. Instead of writing $\lambda_x.M$, we write $(\lambda_x)M$ and instead of MN we write $(N\delta)M$. Item notation has many advantages as shown in [21, 22]. Let us illustrate here with the term A given above which we write in item notation as in Figure 1. We see immediately that the redexes originate from the



$(Q\delta)$ $(R\delta)$ (λ_z) $(P\delta)$ (λ_x) (λ_y) N

Figure 1: Redexes in item notation in term A

couples $(Q\delta)(\lambda_y)$, $(R\delta)(\lambda_z)$ and $(P\delta)(\lambda_x)$. Moreover, $(Q\delta)(R\delta)(\lambda_z)(P\delta)(\lambda_x)(\lambda_y)$ is a well-balanced segment. This natural matching was not present in the classical notation. We call items of the form $(P\delta)$ and (λ_x), application and abstraction items respectively. With item notation, generalised reduction is written as: $(M\delta)\overline{s}(\lambda_x)N \to_{g\beta} \overline{s}\{N[x := M]\}$ for $\overline{s}$ well-balanced. (Here, { and } are used for grouping purposes so that no confusion arises.) For example,

$$(Q\delta)(R\delta)(\lambda_z)(P\delta)(\lambda_x)(\lambda_y)N \to_{g\beta} (R\delta)(\lambda_z)(P\delta)(\lambda_x)\{N[y := Q]\}$$

Surely this is clearer than writing $(\lambda_z.(\lambda_x.\lambda_y.N)P)RQ \to_{g\beta} (\lambda_z.(\lambda_x.N[y := Q]P)R$.

Generalized reduction was first introduced by Nederpelt in 1973 to aid in proving the strong normalization of AUTOMATH [39]. Kamareddine and Nederpelt have shown how generalised reduction makes more redexes visible, allowing flexibility in reducing a term [21]. Bloo, Kamareddine, and Nederpelt show that with generalised reduction one may indeed avoid size explosion without the cost of a longer reduction path and that simultaneously the λ-calculus can be elegantly extended with definitions which result in shorter type derivations [7]. Generalised reduction is strongly normalising [7] for all systems of the λ-cube [4] and preserves the strong normalisation of ordinary β-reduction [19]. In particular, generalized reduction allows the postponement of K-reductions (which discard their argument) after I-reductions (which use their argument in at least one place).

An alternative approach to generalized reduction which has been followed by many researchers is to use one of these two local transformations:

$$\begin{array}{ll} (\theta) & ((\lambda_x.N)P)Q \to (\lambda_x.NQ)P \\ (\gamma) & (\lambda_x.\lambda_y.N)P \to \lambda_y.(\lambda_x.N)P \end{array}$$

These rules transform terms to make more redexes visible to the ordinary notion of β-reduction. For example, the γ rule makes sure that λ_y and Q in the example A above can form a redex before the redex based on λ_x and P is contracted. Also, $((\lambda_x.\lambda_y.N)P)Q \to_\theta (\lambda_x.(\lambda_y.N)Q)P$ and hence both θ and γ put λ_y next to its matching argument. The θ rule moves the argument next to its matching λ whereas γ moves the λ next to its matching argument.

Obviously, θ and γ are related to generalised reduction. in fact, θ and γ transform terms in order to make more potential redexes visible and then conventional β-reduction can be used to contract those newly visible redexes. Generalised reduction on the other hand, performs reduction on the potential redexes without having to bother to make them into classical redexes. Now, we go back to the above example, where with generalised reduction we got: $(\lambda_z.(\lambda_x.\lambda_y.N)P)RQ \rightarrow_{g\beta} (\lambda_z.(\lambda_x.N[y := Q])P)R$. We illustrate how θ and γ work:

$$(\lambda_z.(\lambda_x.\lambda_y.N)P)RQ \rightarrow_\theta (\lambda_z.(\lambda_x.\lambda_y.N)PQ)R \rightarrow_\theta (\lambda_z.(\lambda_x.(\lambda_y.N)Q)P)R \rightarrow_\beta (\lambda_z.(\lambda_x.N[y := Q])P)R$$

$$(\lambda_z.(\lambda_x.\lambda_y.N)P)RQ \rightarrow_\gamma (\lambda_z.\lambda_y(\lambda_x.N)P)RQ \rightarrow_\gamma (\lambda_y(\lambda_z.(\lambda_x.N)P)R)Q \rightarrow_\beta (\lambda_z.(\lambda_x.N[y := Q])P)R$$

Finally, note that in item notation it is easier to describe θ and γ. We illustrate with θ and the above example:

We can reshuffle the term $(Q\delta)(R\delta)(\lambda_z)(P\delta)(\lambda_x)(\lambda_y)N$ to $(R\delta)(\lambda_z)(P\delta)(\lambda_x)(Q\delta)(\lambda_y)N$ in order to transform the bracketing structure {{ }{ }} into { }{ }{ }, where all the redexes correspond to adjacent '{' and '}'. In other words, Figure 1 can be redrawn using the θ-reduction twice in Figure 2.

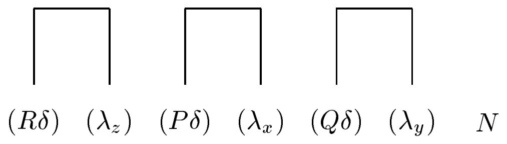


Figure 2: θ-normal forms in item notation for term A

The θ rule can be applied to both explicitly and implicitly typed systems. However, the transfer of γ to explicitly typed systems is not straightforward, since in these systems the type of y in the term A may be affected by the reducible pair of λ_x and P. For example, it is fine to write $((\lambda_{x:*}.\lambda_{y:x}.y)z)u \rightarrow_\theta (\lambda_{x:*}.(\lambda_{y:x}.y)u)z$ but not to write $((\lambda_{x:*}.\lambda_{y:x}.y)z)u \rightarrow_\gamma (\lambda_{y:x}.(\lambda_{x:*}.y)z)u$.[1]

Local transformations like γ and θ began to appear in the literature around 1989. (See [29] for a summary). Regnier [41] introduces the notion of a *premier redex* which is similar to the redex based on λ_y and Q above (which we call a *generalised redex*). Later, he uses θ and γ (and calls the combination σ) to show that the perpetual reduction strategy finds the longest reduction path when the term is Strongly Normalising (SN) [42]. Vidal also introduces similar reductions [46]. Kfoury, Tiuryn, and Urzyczyn use θ (and other reductions) to show that typability in ML is equivalent to acyclic semi-unification [27]. Sabry and Felleisen describe a relationship between a reduction similar to θ and a particular style of CPS [44]. De Groote [13] uses θ and Kfoury and Wells [30] use γ to reduce the problem of β-strong normalisation to the problem of weak normalisation (WN) for related reductions. Kfoury and Wells use θ and γ to reduce typability in the rank-2 restriction of system F to the problem of acyclic semi-unification [28]. Klop, Sørensen, and Xi [31, 47, 45] use related reductions to reduce SN to WN. Finally, [2] uses θ (called "let-C") as a part of an analysis of how to represent sharing in a call-by-need language implementation in a formal calculus.

1.2 The λ-calculus with explicit substitution

Most literature on the λ-calculus treats substitution as an atomic operation and leaves implicit the actual computational steps necessary to perform substitution. Substitution is usually defined with operators which do not belong to the language of the λ-calculus. In any real implementation, the substitution required by

[1]An alternative is to apply γ to the *type erasure* of the term, which may be quite complicated to express in terms of the type-annotated term.

β-reduction (and similar higher-order operations) must be implemented via smaller operations. Thus, there is a conceptual gap between the theory of the λ-calculus and its implementation in programming languages and proof assistants. Explicit substitution attempts to bridge this gap without abandoning the setting of the λ-calculus.

By representing substitutions in the structure of terms and by providing (first-order) reductions to propagate the substitutions, explicit substitution provides a number of benefits. A major benefit is that explicit substitution allows more flexibility in ordering work. Propagating substitutions through a particular subterm can wait until the subterm is the focus of computation. This allows all of these substitutions to be done at once, thus improving locality of reference. Obtaining more control over the ordering of work has become an important issue in functional programming language implementation (cf. [40]). The flexibility provided by explicit substitution also allows postponing unneeded work indefinitely (i.e., avoiding it completely). This can yield profits, since implicit substitution can be an inefficient, maybe even exploding, process by the many repetitions it causes. Another benefit is that explicit substitution allows formal modeling of the techniques used in real implementations, e.g., environments. Because explicit substitution is closer to real implementations, it has the potential to provide a more accurate cost model. (This possibility is particularly interesting in light of the difficulty encountered in formulating a useful cost model in terms of graph reduction [33, 40].)

Proof assistants may benefit from explicit substitution, due to the desire to perform substitutions locally and in a formal manner. Local substitutions are needed as follows. Given $xx[x{:=}y]$, one may not be interested in having yy as the result of $xx[x{:=}y]$ but rather only $yx[x{:=}y]$. In other words, one only substitutes one occurrence of x by y and continues the substitution later. Theorem provers like Nuprl [8] and HOL [14] implement substitution which allows the local replacement of some abbreviated term. This avoids a size explosion when it is necessary to replace a variable by a huge term only in specific places to prove a certain theorem.

Formalization helps in studying the termination and confluence properties of systems. Without formalisation, important properties such as the correctness of substitutions often remain unestablished, causing mistrust in the implementation. As the implementation of substitution in many theorem provers is not based on a formal system, it is not clear what properties their underlying substitution has, nor can their implementations be compared. Thus, it helps to have a choice of explicit substitution systems whose properties have already been established. This is witnessed by the recent theorem prover ALF, which is formally based on Martin-Löf's type theory with explicit substitution [34]. Another justification for explicit substitution in theorem proving is that some researchers believe "tactics" can be replaced by the notion of incomplete proofs, which are believed to need explicit substitutions [37, 34].

The last fifteen years have seen an increasing interest in formalising substitution explicitly; various calculi including new operators to denote substitution have been proposed. Amongst these calculi we mention $C\lambda\xi\phi$ [12]; the calculi of categorical combinators [9]; $\lambda\sigma$ [1], $\lambda\sigma_{\Uparrow}$ [10], $\lambda\sigma_{SP}$ [43], referred to as the $\lambda\sigma$-family; $\lambda\upsilon$ [5], a descendant of the $\lambda\sigma$-family; $\varphi\sigma BLT$ [20], $\lambda\mathtt{exp}$ [6], λs [23], λs_e [26], and $\lambda\zeta$ [36]. All these calculi (except $\lambda\mathtt{exp}$) are described in a de Bruijn setting where natural numbers play the role of variables.

In [23], we extended the λ-calculus with explicit substitutions by turning de Bruijn's meta-operators into object-operators, thus offering a style of explicit substitution that differs from that of $\lambda\sigma$. The resulting calculus λs remains intuitively as close to the λ-calculus as possible for a calculus of explicit substitution. An important motivation for introducing the λs-calculus [23] was to provide a calculus of explicit substitutions which would both preserve strong normalisation and have a confluent extension on open terms [26]. There are calculi of explicit substitutions which are confluent on open terms, e.g. $\lambda\sigma_{\Uparrow}$ [10] and $\lambda\zeta$ [36], but they also have important disadvantages. Melliès proved that $\lambda\sigma_{\Uparrow}$ (as well as both the rest of the $\lambda\sigma$-family and the categorical combinators) does not preserve strong normalisation [35]. There are also calculi which preserve strong normalization, e.g., the $\lambda\upsilon$-calculus [5], but this calculus is not confluent on open terms. Recently, the $\lambda\zeta$-calculus (cf. [36]) has been proposed as a calculus which preserves strong normalisation and is itself confluent on open terms. The $\lambda\zeta$-calculus works with two new applications that allow the passage of substitutions within classical applications only if these applications have a head variable. This is done to cut the branch of the critical pair which is responsible for the non-confluence of $\lambda\upsilon$ on open terms. Hence, $\lambda\zeta$ preserves strong normalisation and is itself confluent on open terms. Unfortunately, $\lambda\zeta$ is not able to simulate one step β-reduction as shown in [36]. Instead, it simulates only a "big step" β-reduction. On the other hand, λs has been extended to λs_e which is confluent on open terms (cf. [26]) and simulates one step β-reduction but the preservation of strong normalisation for the extension λs_e is still an open problem.

1.3 Combining generalised reduction and explicit substitution

We have already explained the separate usefulness of generalised reduction and explicit substitutions. The main benefits of these concepts are similar: both emphasize flexibility in the ordering of operations. In particular, both generalized reduction and explicit substitution allow the postponement of work, but in different, complementary ways. On one side, generalized reduction always allows unnecessary K-redexes to be bypassed. Explicit substitution will not in general allow this, since reducing the K-redex might be necessary to expose an essential I-redex. Similarly, on the other side, explicit substitution allows bypassing any work inside a subterm that will be discarded later. However, generalized reduction does not provide any means for performing only those parts of a substitution that will be used later. Thus, we can see that their benefits are complementary.

We claim that a system with the combination of generalized reduction and explicit substitution is more advantageous than a system with each concept separately. Obviously, if the benefits of both are desired simultaneously, it is important to study the combination, a task which this paper performs. Before the combination can be safely used, it must be checked that this combination is sound and safe exactly like it has been checked that each of explicit substitutions and generalised reductions separately are sound and safe. This paper shows that extending the λ-calculus with both concepts results in theories that are confluent, preserve termination, and simulate β-reduction.

Generalised reduction ($g\beta$), has never before been introduced in a de Bruijn setting. Explicit substitution has almost always been presented in a de Bruijn setting. Since explicit substitution calculi are usually written with de Bruijn indices, we combine $g\beta$-reduction and explicit substitution in a de Bruijn setting, giving the first calculus of generalised reduction à la de Bruijn[2]. As we need to describe generalised redexes in an elegant way, we use a notation for λ-terms suitable for this purpose, the *item notation* [22].

In Section 2 we introduce the calculus of generalised reduction, the λg-calculus, in item notation with de Bruijn indices and prove its confluence. In Section 3 we introduce the λs-calculus and extend it into the λgs-calculus by adding the necessary reductions to simulate $\to_{g\beta}$. We show that λgs is a conservative extension of λg, it simulates $g\beta$ and is confluent. In Section 4 we prove that the λgs-calculus preserves λg-strong normalisation (i.e., a is λg-SN $\Rightarrow$ a is λgs-SN) and that the λs-calculus preserves the λ-strong normalisation. We conclude that a is λ-SN $\Leftrightarrow$ a is λs-SN $\Leftrightarrow$ a is λg-SN $\Leftrightarrow$ a is λgs-SN. In Section 5 the simply typed versions of the λs- and λgs-calculi are presented and subject reduction, typing of subterms, strong normalisation of well typed terms, and other properties are proved.

2 The λg-calculus

We assume familiarity with the λ-calculus and its various notions like reduction, contexts, etc. Where not otherwise defined, we follow the conventions of Barendregt [3, 4]. Nevertheless, we present some basic needed definitions in what follows:

Definition 2.1. [Reduction Notations]
Let A be a set and r a binary relation on A. We denote the fact $(a,b) \in r$ by $a \to_r b$ or $a \to b$ when the context is clear enough. We denote:

1. r^ϵ or $\overset{\epsilon}{\to}_r$ or just $\overset{\epsilon}{\to}$, the reflexive closure r.

2. r^+ or $\to_r^+$ or just $\to^+$, the transitive closure of r.

3. r^* or $\twoheadrightarrow_r$ or just $\twoheadrightarrow$ the reflexive and transitive closure of r. When $a \twoheadrightarrow b$ we say there exists a *reduction sequence* from a to b.

4. $=_r$ the reflexive, symmetric and transitive closure of $\to_r$. That is, $=_r$ is the least equivalence relation containing $\to_r$.

5. $=$ for syntactic identity and write $a = b$ when a and b are syntactically identical.

[2]The main advantages of de Bruijn's notation is that it allows to get rid of Barendregt's variable convention (which insists that free variables be different from bound ones and that if λ_x and λ_y occur in a term, then x must be distinct than y), since α-congruent terms are syntactically identical.

Definition 2.2. **[Reduction Relations and Systems]** For a given set of rewrite rules r on a set A, we call r-reduction, the reduction relation of the r-calculus (i.e., the least compatible relation containing the rules of r). If R is a reduction relation on a set A, we say that (A, R) is a Reduction System.[3]

Definition 2.3. **[Conluence and Church Rosser]**
Let R be a reduction relation on A. For R, we define local confluence (or Weak Church Rosser WCR), confluence (or Church Rosser CR) and strong confluence (or Strong Church Rosser SCR) respectively as follows:

1. WCR: $\forall a, b, c \in A \ \exists d \in A \ : (a \to_R, b \wedge a \to_R c) \Rightarrow (b \twoheadrightarrow_R d \wedge c \twoheadrightarrow_R d)$.
2. CR: $\forall a, b, c \in A \ \exists d \in A \ : (a \twoheadrightarrow_R b \wedge a \twoheadrightarrow_R c) \Rightarrow (b \twoheadrightarrow_R d \wedge c \twoheadrightarrow_R d)$.
3. SCR: $\forall a, b, c \in A \ \exists d \in A \ : (a \to_R b \wedge a \to_R c) \Rightarrow (b \to_R d \wedge c \to_R d)$.

Definition 2.4. **[Normal Forms and Normalisation]**
Let R be a reduction relation on A.

- We say that $a \in A$ is an *R-normal form* (R-nf for short) if there exists no $b \in A$ such that $a \to_R b$.
- We say that b *has an R-normal form* if there exists an R-normal form a such that $b \twoheadrightarrow_R a$. In this case, we say b is R-normalising.
- We say that R is *weakly normalising* (WN) if every $a \in A$ has an R-normal form.
- We say that R is *strongly normalising* (SN) if there is no infinite sequence $(a_i)_{i \geq 0}$ in A such that $a_i \to_R a_{i+1}$ for all $i \geq 0$.
- We say that a term M is *strongly R-normalisaing* if there are no infinite R-reduction sequences starting at M.
- When no confusion arises, then R may be omitted and we speak simply of normal forms or normalisation.

Note that confluence of R guarantees unicity of R-normal forms. In that case, the R-normal form of a, if it exists, is denoted by $R(a)$. Strong normalisation implies weak normalisation and therefore the existence of normal forms. The following lemma is an important connection between strong normalisation and confluence (its proof can be found in [3], proposition 3.1.25):

Lemma 2.5 (Newman).
Every strongly normalising, locally confluent reduction relation is confluent.

We assume familiarity with de Bruijn notation. For example, $\lambda_x.\lambda_y.(x(\lambda_z.zx))y$ is written in ordinary de Bruijn notation as $\lambda(\lambda(2(\lambda(13))1))$ and $\lambda_x.\lambda_y.xy$ as $\lambda\lambda(21)$. To translate free variables, we assume a fixed ordered list of binders (written from left to right) $\cdots, \lambda_z, \lambda_y, \lambda_x$ and prefix it to the term to be translated. Hence, $\lambda_x.yz$ translates as $\lambda 34$ whereas $\lambda_x.zy$ translates as $\lambda 43$. Since generalized β-reduction is better described in item notation, we adopt the item syntax (see [21, 22] for the advantages of item notation) and write $a\,b$ as $(b\,\delta)a$ and λa as $(\lambda)a$. The δ symbol informs us that we are dealing with an application, just like λ informs us that there is an abstraction.

Definition 2.6. The *set of terms* Λ, is defined by the grammar $\Lambda ::= \mathbb{N} \mid (\Lambda\,\delta)\Lambda \mid (\lambda)\Lambda$. We let $a, b, \ldots$ range over Λ and $m, n, \ldots$ over $\mathbb{N}$ (positive natural numbers).[4] We write $a \lhd b$ when a is a subterm of b. A reduction $\to$ is *compatible on* Λ when for all a, b, $c \in \Lambda$, it holds that $a \to b$ implies $(a\,\delta)c \to (b\,\delta)c$, $(c\,\delta)a \to (c\,\delta)b$, and $(\lambda)a \to (\lambda)b$.

[3]Note that we depart from [3], Definition 3.1.1, where a reduction relation is not only compatible, but also reflexive and transitive. Our reason for doing so is that we want to keep the notation for the reduction system simpler.

[4]Our use of $\mathbb{N}$ as the set of *positive* natural numbers may be considered non-standard by computer scientists who insist on having the number 0 as an element of $\mathbb{N}$.

For example, $(\lambda_x\lambda_y.zxy)(\lambda_x.yx) \to_\beta \lambda_u.z(\lambda_x.yx)u$ which in de Bruijn notation is $(\lambda\lambda\mathtt{521})(\lambda\mathtt{31}) \to_\beta \lambda\mathtt{4}(\lambda\mathtt{41})\mathtt{1}$, is expressed in de Bruijn *item* notation as $((\lambda)(\mathtt{1}\delta)\mathtt{3}\delta)(\lambda)(\lambda)(\mathtt{1}\delta)(\mathtt{2}\delta)\mathtt{5} \to_\beta (\lambda)(\mathtt{1}\delta)((\lambda)(\mathtt{1}\delta)\mathtt{4}\delta)\mathtt{4}$. Note that we did not simply replace $\mathtt{2}$ in $(\lambda)(\mathtt{1}\delta)(\mathtt{2}\delta)\mathtt{5}$ by $(\lambda)(\mathtt{1}\delta)\mathtt{3}$. Instead, we decreased $\mathtt{5}$ as one λ disappeared, and incremented the free variables of $(\lambda)(\mathtt{1}\delta)\mathtt{3}$ as they occur within the scope of one more λ. For incrementing the free variables we need updating functions U_k^i, where k tests for free variables and $i-1$ is the value by which a variable, if free, must be incremented:

Definition 2.7. The *updating functions* $U_k^i : \Lambda \to \Lambda$ for $k \geq 0$ and $i \geq 1$ are defined inductively:

$$U_k^i((a\,\delta)b) = (U_k^i(a)\,\delta)U_k^i(b) \qquad U_k^i(\mathtt{n}) = \begin{cases} \mathtt{n+i-1} & \text{if } n > k, \\ \mathtt{n} & \text{if } n \leq k. \end{cases}$$
$$U_k^i((\lambda)a) \;\;= (\lambda)(U_{k+1}^i(a))$$

Now in the following we define meta-substitution. The last equality substitutes the intended variable (when $n = j$) by the updated term. If n is not the intended variable, it is decreased by 1 if it is free (case $n > j$) as one λ has disappeared and if it is bound (case $n < j$) it remains unaltered.

Definition 2.8. The *meta-substitutions at level* j, for $j \geq 1$, of a term $b \in \Lambda$ in a term $a \in \Lambda$, denoted $a\{\!\{\mathtt{j}\leftarrow b\}\!\}$, are defined inductively on a as follows:

$$((a_1\delta)a_2)\{\!\{\mathtt{j}\leftarrow b\}\!\} = ((a_1\{\!\{\mathtt{j}\leftarrow b\}\!\})\delta)(a_2\{\!\{\mathtt{j}\leftarrow b\}\!\}) \qquad \mathtt{n}\{\!\{\mathtt{j}\leftarrow b\}\!\} = \begin{cases} \mathtt{n-1} & \text{if } n > j, \\ U_0^j(b) & \text{if } n = j, \\ \mathtt{n} & \text{if } n < j. \end{cases}$$
$$((\lambda)c)\{\!\{\mathtt{j}\leftarrow b\}\!\} = (\lambda)(c\{\!\{\mathtt{j+1}\leftarrow b\}\!\})$$

The following lemma establishes the properties of meta-substitution and updating.

Lemma 2.9. *Let $a, b, c \in \Lambda$. The following properties hold:*

1. *For $k < n < k+i$:* $(U_k^i(a))\{\!\{\mathtt{n}\leftarrow b\}\!\} = U_k^{i-1}(a)$
2. *For $l \leq k < l+j$:* $U_k^i(U_l^j(a)) = U_l^{j+i-1}(a)$
3. *For $l+j \leq k+1$:* $U_k^i(U_l^j(a)) = U_l^j(U_{k+1-j}^i(a))$
4. *For $k+i \leq n$:* $(U_k^i(a))\{\!\{\mathtt{n}\leftarrow b\}\!\} = U_k^i(a\{\!\{\mathtt{n-i+1}\leftarrow b\}\!\})$
5. *For $n \leq k+1$:* $U_k^i(a\{\!\{\mathtt{n}\leftarrow b\}\!\}) = (U_{k+1}^i(a))\{\!\{\mathtt{n}\leftarrow U_{k-n+1}^i(b)\}\!\}$
6. *For $i \leq n$:* $a\{\!\{\mathtt{i}\leftarrow b\}\!\}\{\!\{\mathtt{n}\leftarrow c\}\!\} = a\{\!\{\mathtt{n+1}\leftarrow c\}\!\}\{\!\{\mathtt{i}\leftarrow b\{\!\{\mathtt{n-i+1}\leftarrow c\}\!\}\}\!\}$

Proof. The proof is by induction on a. The proof of 4 requires 2 with $l = 0$; the proof of 6 uses 1 and 4 both with $k = 0$; finally, 3 with $l = 0$ is needed to prove 5. □

In order to introduce generalised β-reduction we need some definitions (cf. [22]).

Definition 2.10. *Items*, *segments* and *well-balanced segments (w.b.)* are defined respectively by:

$$\mathcal{I} ::= (\Lambda\,\delta) \mid (\lambda) \quad \mathcal{S} ::= \phi \mid \mathcal{I}\,\mathcal{S} \quad \mathcal{W} ::= \phi \mid (\Lambda\delta)\mathcal{W}(\lambda) \mid \mathcal{W}\,\mathcal{W}$$

where ϕ is the empty segment. Hence, a segment is a sequence of items. $(a\,\delta)$ and (λ) are called a δ-*item* and a λ-*item*, respectively. We let I, J, ... range over $\mathcal{I}$; S, S', ... over $\mathcal{S}$; and W, U, ... over $\mathcal{W}$. For a segment S, $\operatorname{len} S$, is given by $\operatorname{len}\phi = 0$ and $\operatorname{len}(I\,S) = 1 + \operatorname{len} S$. The number of main λ-items in S, $\#_\lambda(S)$, is given by $\#_\lambda(\phi) = 0$ and $\#_\lambda((a\,\delta)S) = \#_\lambda(S)$ and $\#_\lambda((\lambda)S) = 1 + \#_\lambda(S)$.

Definition 2.11 (λ-Calculus). The λ-*calculus (à la de Bruijn)* is the reduction system $(\Lambda, \to_\beta)$, where $\to_\beta$ is the least compatible reduction on Λ generated by the β-rule: $(a\delta)(\lambda)b \to b\{\!\{\mathtt{1}\leftarrow a\}\!\}$.

Definition 2.12 (λg-Calculus). The λg-*calculus* is the reduction system $(\Lambda, \to_{g\beta})$, where $\to_{g\beta}$ denotes *generalized β-reduction*, the least compatible reduction on Λ generated by the $g\beta$-rule:

$$(a\delta)W(\lambda)b \to W(b\{\!\{\mathtt{1}\leftarrow U_0^{\#_\lambda(W)+1}(a)\}\!\}) \quad \text{where } W \text{ is well-balanced}$$

Remark 2.13. The β-rule is an instance of the $g\beta$-rule. (Take $W = \phi$ and check that $U_0^1(a) = a$.)

Now, let us briefly explain the relation between $\to_{g\beta}$ and $\to_\theta$ and $\to_\gamma$ given in the introduction. It would be helpful if we write $\to_\theta$ and $\to_\gamma$ in item notation:

$$(Q\delta)(P\delta)(\lambda_x)N \to_\theta (P\delta)(\lambda_x)(Q\delta)N \qquad (P\delta)(\lambda_x)(\lambda_y)N \to_\gamma (\lambda_y)(P\delta)(\lambda_x)N$$

Note how in $\to_\theta$, the start of a redex $(P\delta)(\lambda_x)$ is moved (or reshuffled) giving $(Q\delta)$ the chance to find its matching (λ) in N. In $\to_\gamma$ the same happens but now it is (λ_y) which is given the chance to look for its matching $(-\delta)$. Only once reshuffling has taken place, can the newly found redex be contracted. $\to_{g\beta}$ on the other hand avoids reshuffling and contracts the redex as soon as it sees the matching of δ and λ.

In the following, we extend the definitions of updating and meta-substitution to cover segments and prove some useful properties.

Definition 2.14. Let $S \in \mathcal{S}$, $a, b \in \Lambda$, $k \geq 0$ and $n, i \geq 1$. We define $U_k^i(S)$ and $S\{\!\{\mathtt{n}\leftarrow a\}\!\}$ by:

$$\begin{array}{llll} U_k^i(\phi) & = \phi & \phi\{\!\{\mathtt{n}\leftarrow a\}\!\} & = \phi \\ U_k^i((b\,\delta)S) & = (U_k^i(b)\,\delta)U_k^i(S) & ((b\,\delta)S)\{\!\{\mathtt{n}\leftarrow a\}\!\} & = (b\{\!\{\mathtt{n}\leftarrow a\}\!\}\,\delta)(S\{\!\{\mathtt{n}\leftarrow a\}\!\}) \\ U_k^i((\lambda)S) & = (\lambda)(U_{k+1}^i(S)) & ((\lambda)S)\{\!\{\mathtt{n}\leftarrow a\}\!\} & = (\lambda)(S\{\!\{\mathtt{n+1}\leftarrow a\}\!\}) \end{array}$$

Lemma 2.15. *Let S, T be segments and $a, b \in \Lambda$. The following hold:*

1. *$U_k^i(S\,T) = U_k^i(S)U_{k+\#_\lambda(S)}^i(T)$ and $U_k^i(S\,a) = U_k^i(S)U_{k+\#_\lambda(S)}^i(a)$.*
2. *$\mathrm{len}(S) = \mathrm{len}(U_k^i(S))$ and $\#_\lambda(S) = \#_\lambda(U_k^i(S))$ and if S is w.b. then $U_k^i(S)$ is w.b.*
3. *$(S\,\xi)\{\!\{\mathtt{n}\leftarrow a\}\!\} = S\{\!\{\mathtt{n}\leftarrow a\}\!\}\,\xi\{\!\{\mathtt{n} + \#_\lambda(S)\leftarrow a\}\!\}$ for ξ a segment or a term.*
4. *$\mathrm{len}(S) = \mathrm{len}(S\{\!\{\mathtt{n}\leftarrow a\}\!\})$ and $\#_\lambda(S) = \#_\lambda(S\{\!\{\mathtt{n}\leftarrow a\}\!\})$. If S is w.b. then $S\{\!\{\mathtt{n}\leftarrow a\}\!\}$ is w.b.*

Proof. 1. and 3. By induction on S. 2. and 4. By induction on S using 1. and 3. respectively. □

Lemma 2.16 (Preservation of β-Equivalence). *Let a, $b \in \Lambda$. If $a \twoheadrightarrow_{g\beta} b$ then $a =_\beta b$.*

Proof. It is sufficient to prove by induction on a that $a \to_{g\beta} b$ implies $a =_\beta b$. We will only prove the particular base case $a = (c\delta)W(\lambda)d \to_{g\beta} W(d\{\!\{\mathtt{1}\leftarrow U_0^{\#_\lambda(W)+1}(c)\}\!\}) = b$, with $W \neq \phi$, since the other cases are simpler. We prove this case by a nested induction on $\mathrm{len}\,W$. Observe that $W = (e\delta)W_1(\lambda)W_2$, where W_1 and W_2 are well balanced, because $W \neq \phi$. Let $w_1 = \#_\lambda(W_1)$ and $w_2 = \#_\lambda(W_2)$. We have the following equalities, where in the justifications "IH" means the induction hypothesis and the numbers are lemmas:

$$\begin{array}{lll} & & (c\delta)W(\lambda)d \\ & = & (c\delta)(e\delta)W_1(\lambda)W_2(\lambda)d \\ \text{(IH)} & =_\beta & (c\delta)W_1((W_2(\lambda)d)\{\!\{\mathtt{1}\leftarrow U_0^{w_1+1}(e)\}\!\}) \\ \text{(2.15.3)} & = & (c\delta)W_1(W_2\{\!\{\mathtt{1}\leftarrow U_0^{w_1+1}(e)\}\!\})(\lambda)(d\{\!\{\mathtt{2} + \mathtt{w_2}\leftarrow U_0^{w_1+1}(e)\}\!\}) \\ \text{(IH \& 2.15.4)} & =_\beta & W_1(W_2\{\!\{\mathtt{1}\leftarrow U_0^{w_1+1}(e)\}\!\})(d\{\!\{\mathtt{2} + \mathtt{w_2}\leftarrow U_0^{w_1+1}(e)\}\!\}\{\!\{\mathtt{1}\leftarrow U_0^{w_1+w_2+1}(c)\}\!\}) \\ \text{(2.9.1)} & = & W_1(W_2\{\!\{\mathtt{1}\leftarrow U_0^{w_1+1}(e)\}\!\}) \\ & & (d\{\!\{\mathtt{2} + \mathtt{w_2}\leftarrow U_0^{w_1+1}(e)\}\!\}\{\!\{\mathtt{1}\leftarrow U_0^{w_1+w_2+2}(c)\{\!\{\mathtt{1} + \mathtt{w_2}\leftarrow U_0^{w_1+1}(e)\}\!\}\}\!\}) \\ \text{(2.9.6)} & = & W_1(W_2\{\!\{\mathtt{1}\leftarrow U_0^{w_1+1}(e)\}\!\})(d\{\!\{\mathtt{1}\leftarrow U_0^{w_1+w_2+2}(c)\}\!\}\{\!\{\mathtt{1} + \mathtt{w_2}\leftarrow U_0^{w_1+1}(e)\}\!\}) \\ \text{(2.15.3 \& 2.15.4)} & = & W_1((W_2(d\{\!\{\mathtt{1}\leftarrow U_0^{w_1+w_2+2}(c)\}\!\}))\{\!\{\mathtt{1}\leftarrow U_0^{w_1+1}(e)\}\!\}) \\ \text{(IH)} & =_\beta & (e\delta)W_1(\lambda)W_2(d\{\!\{\mathtt{1}\leftarrow U_0^{w_1+w_2+2}(c)\}\!\}) \\ & = & W(d\{\!\{\mathtt{1}\leftarrow U_0^{\#_\lambda(W)+1}(c)\}\!\}) \end{array}$$

□

Theorem 2.17 (Confluence of λg). *The λg-calculus is confluent.*

Proof. This proof is the de Bruijn version of the proof given in [21]. Let $a \twoheadrightarrow_{g\beta} b$ and $a \twoheadrightarrow_{g\beta} c$. By Lemma 2.16, $a =_\beta b$ and $a =_\beta c$, hence $b =_\beta c$. By confluence of β, $\exists d \in \Lambda$ where $b \twoheadrightarrow_\beta d$ and $c \twoheadrightarrow_\beta d$. By Remark 2.13, $b \twoheadrightarrow_{g\beta} d$ and $c \twoheadrightarrow_{g\beta} d$.

There are, as we mentioned in the introduction, various notions of generalised reduction. For other proofs of confluence of some of these notions, we refer the reader to [2, 13, 19, 30, 31]. □

Finally, the following ensures the good passage of $g\beta$-reduction through $\{\!\{\leftarrow\ \}\!\}$ and U_k^i:

Lemma 2.18. *Let a, b, c, $d \in \Lambda$. The following hold:*

1. *If $c \to_{g\beta} d$ then $U^i_k(c) \to_{g\beta} U^i_k(d)$.*
2. *If $c \to_{g\beta} d$ then $a\{\!\{\mathtt{n} \leftarrow c\}\!\} \twoheadrightarrow_{g\beta} a\{\!\{\mathtt{n} \leftarrow d\}\!\}$.*
3. *If $a \to_{g\beta} b$ then $a\{\!\{\mathtt{n} \leftarrow c\}\!\} \to_{g\beta} b\{\!\{\mathtt{n} \leftarrow c\}\!\}$.*

Proof.
1. By induction on c. We only prove the base case where $c = (c_1\delta)W(\lambda)c_3$, W well balanced, and $d = W(c_3\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(c_1)\}\!\})$.

$$\begin{array}{rcl}
 & & U^i_k(c) \\
 & = & U^i_k((c_1\delta)W(\lambda)c_3) \\
(2.15.1) & = & (U^i_k(c_1)\delta)(U^i_k(W))(\lambda)U^i_{k+\#_\lambda(W)+1}(c_3) \\
(2.15.2) & \to_{g\beta} & (U^i_k(W))((U^i_{k+\#_\lambda(W)+1}(c_3))\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(U^i_k(c_1))\}\!\}) \\
(2.9.3) & = & (U^i_k(W))((U^i_{k+\#_\lambda(W)+1}(c_3))\{\!\{\mathtt{1} \leftarrow U^i_{k+\#_\lambda(W)}(U_0^{\#_\lambda(W)+1}(c_1))\}\!\}) \\
(2.9.5) & = & (U^i_k(W))U^i_{k+\#_\lambda(W)}(c_3\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(c_1)\}\!\}) \\
(2.15.1) & = & U^i_k(W(c_3\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(c_1)\}\!\})) \\
 & = & U^i_k(d)
\end{array}$$

2. By induction on a using 1.
3. By induction on a. We only prove the base case: $a = (a_1\delta)W(\lambda)a_2$ and $b = W(a_2\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(a_1)\}\!\})$.

$$\begin{array}{rcl}
 & & a\{\!\{\mathtt{i} \leftarrow c\}\!\} \\
 & = & ((a_1\delta)W(\lambda)a_2)\{\!\{\mathtt{i} \leftarrow c\}\!\} \\
(2.15.3) & = & (a_1\{\!\{\mathtt{i} \leftarrow c\}\!\}\delta)(W\{\!\{\mathtt{i} \leftarrow c\}\!\})(\lambda)(a_2\{\!\{\mathtt{i} + \#_\lambda(W) + \mathtt{1} \leftarrow c\}\!\}) \\
(2.15.4) & \to_{g\beta} & W\{\!\{\mathtt{i} \leftarrow c\}\!\}(a_2\{\!\{\mathtt{i} + \#_\lambda(W) + \mathtt{1} \leftarrow c\}\!\}\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(a_1\{\!\{\mathtt{i} \leftarrow c\}\!\})\}\!\}) \\
(2.9.4) & = & W\{\!\{\mathtt{i} \leftarrow c\}\!\}(a_2\{\!\{\mathtt{i} + \#_\lambda(W) + \mathtt{1} \leftarrow c\}\!\}\{\!\{\mathtt{1} \leftarrow (U_0^{\#_\lambda(W)+1}(a_1))\{\!\{\mathtt{i} + \#_\lambda(W) \leftarrow c\}\!\}\}\!\}) \\
(2.9.6) & = & W\{\!\{\mathtt{i} \leftarrow c\}\!\}(a_2\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(a_1)\}\!\}\{\!\{\mathtt{i} + \#_\lambda(W) \leftarrow c\}\!\}) \\
(2.15.3) & = & (W(a_2\{\!\{\mathtt{1} \leftarrow U_0^{\#_\lambda(W)+1}(a_1)\}\!\}))\{\!\{\mathtt{i} \leftarrow c\}\!\} \\
 & = & b\{\!\{\mathtt{i} \leftarrow c\}\!\}
\end{array}$$

□

3 The λs- and λgs-calculi

The $\lambda\sigma$-calculus (cf. [1]) reflects in its choice of operators and rules the calculus of categorical combinators (cf. [9]). The main innovation of the $\lambda\sigma$-calculus is the division of terms in two sorts: sort `term` and sort `substitution`. We depart from this style of explicit substitutions in two ways. First, we keep the classical and unique sort `term` of the λ-calculus. Second, we do not use some of the categorical operators, especially those which are not present in the classical λ-calculus. We introduce new operators which reflect the substitution and updating that are only present in the meta-language of the λ-calculus. By doing so, we believe that our calculi are closer to the λ-calculus from an intuitive point of view, rather than a categorical one.

A calculus accommodating explicit substitution via explicit rewrite rules in the λ-calculus was first presented in [20]. In that article, the intention was to introduce the philosophy in general and the calculus obtained did not possess neither confluence nor preservation of strong normalisation. In [23] the part of the calculus that was confluent and preserved strong normalisation was singled out. In this paper, we take that part (λs) and extend it with generalised reduction. We start this section by presenting the λs- and λgs-calculi and then by studying their properties.

The λs-calculus is obtained by internalising the meta-operators of Definitions 2.7 and 2.8 in order to handle substitutions explicitly. Therefore, the syntax of the λs-calculus is obtained by adding to Λ two families of operators:

1. Explicit substitution operators $\{\sigma^j\}_{j\geq 1}$ where $(b\,\sigma^j)a$ stands for a where all free occurrences of the variable representing the index j are to be substituted by the appropriately updated b.

$$\boxed{\begin{array}{lrcl}
(\sigma\textit{-generation}) & (b\,\delta)(\lambda)a & \longrightarrow & (b\,\sigma^1)a \\
(\sigma\text{-}\lambda\textit{-transition}) & (b\,\sigma^j)(\lambda)a & \longrightarrow & (\lambda)(b\,\sigma^{j+1})a \\
(\sigma\text{-}\delta\textit{-transition}) & (b\,\sigma^j)(a_1\delta)a_2 & \longrightarrow & ((b\,\sigma^j)a_1\delta)\,(b\,\sigma^j)a_2 \\
(\sigma\textit{-destruction}) & (b\,\sigma^j)\mathtt{n} & \longrightarrow & \begin{cases} \mathtt{n-1} & \text{if } n > j, \\ (\varphi_0^j)b & \text{if } n = j, \\ \mathtt{n} & \text{if } n < j. \end{cases} \\
(\varphi\text{-}\lambda\textit{-transition}) & (\varphi_k^i)(\lambda)a & \longrightarrow & (\lambda)(\varphi_{k+1}^i)a \\
(\varphi\text{-}\delta\textit{-transition}) & (\varphi_k^i)(a_1\delta)a_2 & \longrightarrow & ((\varphi_k^i)a_1\delta)(\varphi_k^i)a_2 \\
(\varphi\textit{-destruction}) & (\varphi_k^i)\mathtt{n} & \longrightarrow & \begin{cases} \mathtt{n+i-1} & \text{if } n > k, \\ \mathtt{n} & \text{if } n \leq k. \end{cases}
\end{array}}$$

Figure 3: The λs-calculus

2. Updating operators $\{\varphi_k^i\}_{\substack{k\geq 0 \\ i\geq 1}}$ necessary for working with de Bruijn indices.

Definition 3.1. The *set of terms of the λs-calculus*, denoted Λs, is given as follows:

$$\Lambda s ::= \mathbb{N} \mid (\Lambda s\,\delta)\Lambda s \mid (\lambda)\Lambda s \mid (\Lambda s\,\sigma^j)\Lambda s \mid (\varphi_k^i)\Lambda s \quad \text{where } j, i \geq 1 \text{ and } k \geq 0$$

We let a, b, and c range over Λs. A term $(a\,\sigma^j)b$ is called a *closure*. Furthermore, a term containing neither σ's nor φ's is called a *pure term*. The symbol Λ denotes the set of pure terms. The set $\mathcal{DL}$ of *$\delta\lambda$-segments* is the set whose main items are either δ-items or λ-items, i.e., $\mathcal{DL} ::= \phi \mid (\Lambda s\,\delta)\mathcal{DL} \mid (\lambda)\mathcal{DL}$. As usual, a reduction $\to$ on Λs is *compatible* if for all $a, b, c \in \Lambda s$, if $a \to b$ then $(a\,\delta)c \to (b\,\delta)c$, $(c\,\delta)a \to (c\,\delta)b$, $(\lambda)a \to (\lambda)b$, $(a\,\sigma^j)c \to (b\,\sigma^j)c$, $(c\,\sigma^j)a \to (c\,\sigma^j)b$ and $(\varphi_k^i)a \to (\varphi_k^i)b$.

Definition 3.2. *Items*, *segments* and *well-balanced segments* for Λs are defined as follows:

$$\begin{array}{lcl}
\mathcal{I}s & ::= & (\Lambda s\,\delta) \mid (\lambda) \mid (\Lambda s\,\sigma^j) \mid (\varphi_k^i) \\
\mathcal{S}s & ::= & \phi \mid \mathcal{I}s\,\mathcal{S}s \\
\mathcal{W}s & ::= & \phi \mid (\Lambda s\,\delta)\mathcal{W}s(\lambda) \mid \mathcal{W}s\,\mathcal{W}s
\end{array}$$

We let I, J, $\ldots$ range over $\mathcal{I}s$; S, S', $\ldots$ over $\mathcal{S}s$; and W, U, $\ldots$ over $\mathcal{W}s$. We call $(a\,\sigma^j)$ and (φ_k^i), a *σ-item* and a *φ-item* respectively. The notion len(S) is trivially extended to $S \in \mathcal{S}s$ in the obvious way and $\#_\lambda(S)$ is extended by declaring that $\#_\lambda((a\,\sigma^j)S) = \#_\lambda(S)$ and $\#_\lambda((\varphi_k^i)S) = \#_\lambda(S)$.

As the λs-calculus updates and substitutes explicitly, we include a set of rules which are the equations in Definitions 2.7 and 2.8 oriented from left to right.

Definition 3.3 (λs-Calculus). The *λs-calculus* is the reduction system $(\Lambda s, \to_{\lambda s})$, where $\to_{\lambda s}$ is the least compatible reduction on Λs generated by the rules given in Figure 3. We use λs to denote this set of rules.

Definition 3.4 (s-Calculus). The *calculus of substitutions associated with the λs-calculus* is the reduction system generated by the set of rules $s = \lambda s - \{(\sigma\textit{-generation})\}$ and we call it the *s-calculus*.

Definition 3.5 (Notation for s-Normal Forms). We use $s(a)$ to denote the s-normal form of a. Define $s(S)$ for a $\delta\lambda$-segment by $s(\phi) = \phi$ and $s((a\,\delta)S) = (s(a)\,\delta)s(S)$ and $s((\lambda)S) = (\lambda)s(S)$.

Definition 3.6 (λgs-Calculus). The *λgs-calculus* is the calculus whose set of rules consists of λgs where $\lambda gs = \lambda s \cup \{(g\sigma\textit{-generation})\}$ and:

$$(g\sigma\textit{-generation}): \qquad (b\,\delta)W(\lambda)a \longrightarrow W((\varphi_0^{\#_\lambda(W)+1})b\,\sigma^1)a \quad \text{where } W \text{ well balanced and } W \neq \phi$$

Note that in the λgs-calculus we do not merge (σ*-generation*) and ($g\sigma$*-generation*) into the following:

$$(\textit{new } g\sigma\textit{-generation}): \qquad (b\,\delta)W(\lambda)a \longrightarrow W((\varphi_0^{\#_\lambda(W)+1})b\,\sigma^1)a \quad \text{where } W \text{ well balanced}$$

The reason for this lies in the fact that (*new* $g\sigma$*-generation*) does not generalise (σ*-generation*) of the λs-calculus. That is, $(b\delta)(\lambda)a \rightarrow_{\sigma\text{-}gen} (b\sigma^1)a$ yet $(b\delta)(\lambda)a \rightarrow_{new\ g\sigma\text{-}gen} ((\varphi_0^1 b)\sigma^1)a$.

The (σ*-generation*) rule starts the simulation of a β-reduction by generating a substitution operator (σ^1). The (σ-δ*-transition*) and (σ-λ*-transition*) rules propagate copies of this operator throughout the term until they arrive at the variable occurrences. If a variable should be affected by the substitution, the (σ*-destruction*) rule (case $j = n$) carries out the substitution by the updated term, thus introducing the updating operators. Finally the φ*-rules* compute the updating.

We state now the following theorem of the λs-calculus.

Theorem 3.7. *The following holds:*

1. *The s-calculus is strongly normalising and confluent on* Λs.
2. *All s-normal forms are unique.*
3. *The set of s-normal forms is exactly* Λ.
4. *For every* $a, b \in \Lambda s$, *the following hold:*

$$\begin{array}{ll} 1.\ s((a\,\delta)b) = (s(a)\,\delta)s(b) & 2.\ s((\lambda)a) \quad = (\lambda)(s(a)) \\ 3.\ s((\varphi_k^i)a) = U_k^i(s(a)) & 4.\ s((b\,\sigma^j)a) = s(a)\{\!\{\mathtt{j} \leftarrow s(b)\}\!\} \end{array}$$

Proof.

1. We define recursively a weight function W:
   $$\begin{array}{lll} W(\mathtt{n}) = 1 & W((a\delta)b) = W(a) + W(b) + 1 & W((\lambda)a) = W(a) + 1 \\ W((\varphi_k^i)a) = 2W(a) & W((b\,\sigma^j)a) = 2W(a)(W(b)+1) & \end{array}$$
   It is easy to show by induction on a that $a \rightarrow_s b$ implies $W(a) > W(b)$, hence the s-calculus is strongly normalising.

   As for confluence, note first that the reduction $\rightarrow_s$ is locally confluent because there are no critical pairs and the theorem of Knuth-Bendix applies trivially. Finally, Newman's Lemma (see Lemma 2.5) guarantees confluence.
2. The existence and unicity of s-normal forms (s-nf) is guaranteed by 1.
3. Check first by induction on a that $(b\,\sigma^i)a$ and $(\varphi_k^i)a$ are not s-normal forms. Then check by induction on a that if a is an s-nf then $a \in \Lambda$. Conclude by observing that every term in Λ is in s-nf.
4. 4.1 and 4.2 hold because there are no s-rules whose left-hand side is an application or an abstraction. 4.3 is shown as follows: first show the equality for terms in s-nf, i.e. use an inductive argument on $c \in \Lambda$ to show $s((\varphi_k^i)c) = U_k^i(s(c))$. Let now $a \in \Lambda s$, $s((\varphi_k^i)a) = s((\varphi_k^i)s(a)) = U_k^i(s(s(a))) = U_k^i(s(a))$. 4.4 is shown similarly to (and using) 4.3.

□

Lemma 3.8. *Let* $a, b \in \Lambda s$. *Then both of these statements hold:*

1. *If* $a \rightarrow_{\sigma\text{-}gen} b$ *then* $s(a) \twoheadrightarrow_\beta s(b)$.
2. *If* $a \rightarrow_{g\sigma\text{-}gen} b$ *then* $s(a) \twoheadrightarrow_{g\beta} s(b)$.

Proof. The first claim is proved by induction on a using Lemma 2.18 and Theorem 3.7. For the second claim, we need the following additional argument. Observe that for any $\delta\lambda$-segment S it holds that $s(S\,a) = s(S)s(a)$. Then note that if W is well balanced then it is a $\delta\lambda$-segment and thus $s(W\,a) = s(W)s(a)$. □

Corollary 3.9. *Let $a, b \in \Lambda s$. Then both of these statements hold:*

1. *If $a \twoheadrightarrow_{\lambda s} b$ then $s(a) \twoheadrightarrow_{\beta} s(b)$.*
2. *If $a \twoheadrightarrow_{\lambda gs} b$ then $s(a) \twoheadrightarrow_{g\beta} s(b)$.*

Corollary 3.10 (Conservative Extension). *Let $a, b \in \Lambda$. Then both of these statements hold:*

1. *If $a \twoheadrightarrow_{\lambda s} b$ then $a \twoheadrightarrow_{\beta} b$.*
2. *If $a \twoheadrightarrow_{\lambda gs} b$ then $a \twoheadrightarrow_{g\beta} b$.*

This last corollary says that the $\lambda(g)s$-calculus is correct with respect to the $\lambda(g)$-calculus, i.e., if a $\lambda(g)s$-reduction sequence begins and ends with pure terms, there is a $\lambda(g)$-reduction sequence beginning and ending with the same terms.

Moreover, the $\lambda(g)s$-calculus is powerful enough to simulate $(g)\beta$-reduction.

Lemma 3.11 (Simulation of $(g)\beta$-Reduction). *Let $a, b \in \Lambda$. Then the following statements hold:*

1. *If $a \to_{\beta} b$ then $a \to^{+}_{\lambda s} b$.*
2. *If $a \to_{g\beta} b$ then $a \to^{+}_{\lambda gs} b$.*

Proof. 1 is by induction on a. As usual the interesting case is when $a = (\lambda c)d$ and $b = c\{\!\{1 \leftarrow d\}\!\}$. In this case: $(\lambda c)d \to_{\sigma-gen} c\sigma^1 d \twoheadrightarrow_s s(c\sigma^1 d) \overset{T3.7}{=} s(c)\{\!\{1 \leftarrow s(d)\}\!\} \overset{c,d\in\Lambda}{=} c\{\!\{1 \leftarrow d\}\!\}$.
2 is by induction on a using Theorem 3.7. $\square$

Corollary 3.12. *Let $a \in \Lambda$. The following hold:*

1. *If a is strongly normalising in the λs-calculus, then a is strongly normalising in the λ-calculus.*
2. *If a is strongly normalising in the λgs-calculus, then a is strongly normalising in the λg-calculus.*

We prove now the confluence of λs and λgs on ground terms. We remark that not even λs is confluent on open terms. As a matter of fact, in order to obtain confluence on open terms, certain rules must be added. The calculus thus obtained, λs_e has been shown confluent (cf. [26]). The combination of λs_e with generalised reduction has not yet been studied.

Theorem 3.13 (Confluence of λs and λgs). *The λs and λgs-calculi are confluent on Λs.*

Proof. We use the interpretation method (cf. [16, 10]). To prove confluence of the λs-calculus, remove all the (g)'s from the proof below. For the confluence of the λgs-calculus, leave the (g)'s but remove the ()'s that embrace the g's. The proof goes as follows:

We interpret the $\lambda(g)s$-calculus into the $\lambda(g)$-calculus via s-normalisation:

$$\begin{tikzcd}
 & b \arrow[rr, two heads, "s"] & & s(b) \arrow[dr, "(g)\beta"] \arrow[dr, bend left, "\lambda(g)s"] & \\
a \arrow[ur, "\lambda(g)s"] \arrow[rr, two heads, "s"] \arrow[dr, "\lambda(g)s"'] & & s(a) \arrow[ur, "(g)\beta"] \arrow[dr, "(g)\beta"'] \arrow[rr, phantom, "\text{Thm. 2.17}"] & & d \\
 & c \arrow[rr, two heads, "s"] & & s(c) \arrow[ur, "(g)\beta"] \arrow[ur, bend right, "\lambda(g)s"'] &
\end{tikzcd}$$

The existence of the arrows $s(a) \twoheadrightarrow_{(g)\beta} s(b)$ and $s(a) \twoheadrightarrow_{(g)\beta} s(c)$ is guaranteed by Corollary 3.9. We can close the diamond thanks to the confluence of the $\lambda(g)$-calculus. Finally, Lemma 3.11 ensures $s(b) \twoheadrightarrow_{\lambda(g)s} d$ and $s(b) \twoheadrightarrow_{\lambda(g)s} d$ proving thus the confluence for the $\lambda(g)s$-calculus. $\square$

4 Preservation of Strong Normalisation

We show in this section that the λs-calculus preserves the λ-calculus strong normalisation and that the λgs-calculus preserves the λg-calculus strong normalisation.

The technique used in this section to prove preservation of strong normalisation (PSN) is an adaptation of the minimal derivation method used in [5] to prove PSN for λv and in [23] to prove PSN for λs. Our proof includes the first proof of the commutation of arbitrary external and internal reduction. Moreover, we give an inductive and elegant definition of internal/external reduction instead of the one that depends on internal and external positions as in [5]. Finally, we introduce a syntactic notion of *skeletons* that will be very informative about internal and external reduction. The elegance of our presentation is reflected by the fact that one proof is enough to achieve both preservation results above.

Notation 4.1. We write $a \in \lambda$-SN resp. $a \in \lambda r$-SN when a is strongly normalising in the λ-calculus resp. in the λr-calculus for $r \in \{g, gs, s\}$. We write $a \xrightarrow[p]{} b$ to denote that p is the position of the redex which is contracted. Therefore $a \xrightarrow[\epsilon]{} b$ means that the reduction takes place at the root. We denote by $\prec$ the prefix order between positions in a term. Hence if p, q are positions in the term a such that $p \prec q$, and we write a_p (resp. a_q) for the subterm of a at position p (resp. q), then a_q is a subterm of a_p.

For example, if $a = ((4\delta)(\lambda)1\sigma^3)2$, we have $a_1 = 2$, $a_2 = (4\delta)(\lambda)1$, $a_{21} = (\lambda)1$, $a_{211} = 1$, $a_{22} = 4$. For example, since $2 \prec 21$, it must hold that a_{21} is a subterm of a_2.

The following three lemmas assert that all the σ's in the last term of a reduction sequence beginning with a λ-term must have been created at some previous step by a (generalized) ($(g)\sigma$*-generation*) and trace the history of these closures. The first lemma deals with one-step reduction where the redex is at the root; the second generalises the first; the third treats arbitrary reduction sequences.

Lemma 4.2. *Let* $\to \in \{\to_{\lambda s}, \to_{\lambda gs}\}$. *If* $a \xrightarrow[\epsilon]{} C[(e\,\sigma^i)d]$ *then one of the following must hold:*

1. $a = (e\,\delta)(\lambda)d$, $C = [\,\cdot\,]$, *and* $i = 1$.
2. $\to = \to_{\lambda gs}$, $a = (e'\,\delta)W(\lambda)d$, $W \neq \phi$, $C = W[\,\cdot\,]$, $e = (\varphi_0^{\#_\lambda(W)+1})e'$, *and* $i = 1$.
3. $a = C'[(e\,\sigma^j)d']$ *for some context* C', *some term* d' *and* $j \in \{i-1, i\}$.

Proof. Since the reduction is at the root, we must check for every rule $a \to a'$ that if $(e\,\sigma^i)d$ occurs in a' then one of the three possibilities follows. We supply proofs only for the interesting rules:

(σ*-generation*) **:** $a = (c\,\delta)(\lambda)b$ and $a' = (c\,\sigma^1)b$. If $(e\,\sigma^i)d$ matches $(c\,\sigma^1)d$ then 1. Else $(e\,\sigma^i)d$ must occur within b or c and hence 3. with $j = i$ and $d' = d$.

($g\sigma$*-generation*) **:** Occurs only if $\to = \to_{\lambda gs}$. $a = (c\,\delta)W(\lambda)b$, $W \neq \phi$ and $a' = W((\varphi_0^{\#_\lambda(W)+1})c\,\sigma^1)b$. If $(e\,\sigma^i)d$ is $((\varphi_0^{\#_\lambda(W)+1})c\,\sigma^1)d$ then 2. Else $(e\,\sigma^i)d$ occurs in b, c or W, hence 3. with $j = i$, $d' = d$.

(σ-λ*-transition*) **:** $a = (c\,\sigma^h)(\lambda)b$ and $a' = (\lambda)(c\,\sigma^{h+1})b$. If $(e\,\sigma^i)d$ matches $(c\,\sigma^{h+1})b$ then 3. with $j = i-1$, $d' = (\lambda)d$. Else $(e\,\sigma^i)d$ occurs in b or c, hence 3. with $j = i$, $d' = d$. □

Lemma 4.3. *Let* $\to \in \{\to_{\lambda s}, \to_{\lambda gs}\}$. *If* $a \to C[(e\,\sigma^i)d]$ *then one of the following must hold:*

1. $a = C[(e\,\delta)(\lambda)d]$ *and* $i = 1$.
2. $\to = \to_{\lambda gs}$, $a = C'[(e'\,\delta)W(\lambda)d]$, $C = C'[W[\,\cdot\,]]$, $e = (\varphi_0^{\#_\lambda(W)+1})e'$, *and* $i = 1$.
3. $a = C'[(e'\,\sigma^j)d']$ *where* $e' = e$ *or* $e' \to e$ *and* $j \in \{i-1, i\}$.

Proof. Induction on a, using lemma 4.2 for the reductions at the root. □

Lemma 4.4. *Let* $\to \in \{\to_{\lambda s}, \to_{\lambda gs}\}$. *Let* $a_1 \to \ldots \to a_n \to a_{n+1} = C[(e\,\sigma^i)d]$. *There exists* $e', d' \in \Lambda s$ *and a context* $C'[\,\cdot\,]$ *such that* $e' \twoheadrightarrow e$ *and one of the following holds:*

1. $a_k = C'[(e'\,\delta)(\lambda)d']$ *and* $a_{k+1} = C'[(e'\,\sigma^1)d']$ *for some* $k \leq n$.

2. *$\to = \to_{\lambda gs}$, $a_k = C'[(e'\,\delta)W(\lambda)d']$ and $a_{k+1} = C'[W(e''\,\sigma^1)d']$ and $e'' = (\varphi_0^{\#_\lambda(W)+1})e'$ for $k \le n$ and w.b. W.*

3. *$a_1 = C'[(e'\,\sigma^j)d']$ where $j \le i$.*

Proof. Induction on n and use the previous lemma. □

We define now internal and external reductions. An internal reduction takes place somewhere at the left of a σ^i operator. An external reduction is a non-internal one. Our definition is inductive rather than starting from the notion of internal and external position as in [5].

Definition 4.5 (Internal Reduction).
For any notion of reduction r, the reduction $\xrightarrow{\mathtt{int}}_r$ is defined by the following rules:

$$\frac{a \longrightarrow_r b}{(a\,\sigma^i)c \xrightarrow{\mathtt{int}}_r (b\,\sigma^i)c} \qquad \frac{a \xrightarrow{\mathtt{int}}_r b}{(a\,\delta)c \xrightarrow{\mathtt{int}}_r (b\,\delta)c} \qquad \frac{a \xrightarrow{\mathtt{int}}_r b}{(c\,\delta)a \xrightarrow{\mathtt{int}}_r (c\,\delta)b}$$

$$\frac{a \xrightarrow{\mathtt{int}}_r b}{(\lambda)a \xrightarrow{\mathtt{int}}_r (\lambda)b} \qquad \frac{a \xrightarrow{\mathtt{int}}_r b}{(c\,\sigma^i)a \xrightarrow{\mathtt{int}}_r (c\,\sigma^i)b} \qquad \frac{a \xrightarrow{\mathtt{int}}_r b}{(\varphi^i_k)a \xrightarrow{\mathtt{int}}_r (\varphi^i_k)b}$$

Therefore, $\xrightarrow{\mathtt{int}}_r$ is the least compatible relation closed under $\dfrac{a \longrightarrow_r b}{(a\,\sigma^i)c \xrightarrow{\mathtt{int}}_r (b\,\sigma^i)c}$.

Remark 4.6. By inspecting the inference rules one can check that:

1. If $a \xrightarrow{\mathtt{int}}_r (\lambda)b$ then $a = (\lambda)c$ and $c \xrightarrow{\mathtt{int}}_r b$.
2. If $a \xrightarrow{\mathtt{int}}_r (c\,\delta)b$ then $a = (e\,\delta)d$ and $((d \xrightarrow{\mathtt{int}}_r b$ and $e = c)$ or $(e \xrightarrow{\mathtt{int}}_r c$ and $d = b))$.
3. If $a \xrightarrow{\mathtt{int}}_r W(\lambda)b$ with W well-balanced, then one of the following holds:
   - $a = W(\lambda)b'$ with $b' \xrightarrow{\mathtt{int}}_r b$.
   - $W = W_1(b'_1\delta)W_2(\lambda)W_3$ where W_1, W_2, and W_3 are well-balanced, $a = W_1(b_1\delta)W_2(\lambda)W_3(\lambda)b$ and $b_1 \xrightarrow{\mathtt{int}}_r b'_1$.
4. $a \xrightarrow{\mathtt{int}}_r \mathtt{n}$ is impossible.

Definition 4.7 (External Reduction). For any notion of reduction r, the reduction $\xrightarrow{\mathtt{ext}}_r$ is defined by induction. The axioms are the rules of r and the inference rules are the following:

$$\frac{a \xrightarrow{\mathtt{ext}}_r b}{(a\,\delta)c \xrightarrow{\mathtt{ext}}_r (b\,\delta)c} \qquad \frac{a \xrightarrow{\mathtt{ext}}_r b}{(c\,\delta)a \xrightarrow{\mathtt{ext}}_r (c\,\delta)b}$$

$$\frac{a \xrightarrow{\mathtt{ext}}_r b}{(\lambda)a \xrightarrow{\mathtt{ext}}_r (\lambda)b} \qquad \frac{a \xrightarrow{\mathtt{ext}}_r b}{(c\,\sigma^i)a \xrightarrow{\mathtt{ext}}_r (c\,\sigma^i)b} \qquad \frac{a \xrightarrow{\mathtt{ext}}_r b}{(\varphi^i_k)a \xrightarrow{\mathtt{ext}}_r (\varphi^i_k)b}$$

Note that the potential rule $\dfrac{a \xrightarrow{\mathtt{ext}}_r b}{(a\,\sigma^i)c \xrightarrow{\mathtt{ext}}_r (b\,\sigma^i)c}$ is excluded from the definition of external reduction. Thus, as expected, external reductions will not occur at the left of a σ^i operator. This enables us to write $\to^+{}_\beta$ instead of $\twoheadrightarrow_\beta$ in the following proposition (compare with Lemma 3.8).

Proposition 4.8. *Let $a, b \in \Lambda s$. $a \xrightarrow{\mathtt{ext}}_{\sigma\text{-}gen} b \Rightarrow s(a) \to^+{}_\beta\, s(b)$ and $a \xrightarrow{\mathtt{ext}}_{g\sigma\text{-}gen} b \Rightarrow s(a) \to^+{}_{g\beta}\, s(b)$.*

Proof. By induction on a (as in Lemma 3.8). Note that when $a = (d\,\sigma^i)c$, the reduction cannot take place within d because it is external, and this is the only case that forced us to consider the reflexive-transitive closure because of Lemma 2.18.2. □

The following is needed in Lemma 4.11 and hence in the Preservation Theorem. Note that we depart from the traditional *minimal derivation method* (which we call here *minimal reduction sequence method*) which commutes internal λr-steps and external s-steps and assumes that $s(a)$ is λ-SN and that $s(a) = s(b)$. Instead, we commute arbitrary internal and external reduction and drop the extra assumptions concerning SN and the s-normal forms. Our generality enables us to simplify the proof of the commutation lemma (no need to always check (during the induction) that terms are λ-SN and evaluate the s-normal forms). Moreover, our commutation of arbitrary internal and external reduction simplifies Lemma 4.11 needed in the proof of PSN. In particular, Lemma 4.11 drops the condition that the term is strongly normalising and its proof is very simple.

Lemma 4.9 (Commutation of Internal/External Reduction).
Let $a, b \in \Lambda s$ and $r \in \{s, gs\}$. If $a \xrightarrow{\texttt{int}}_{\lambda r} \cdot \xrightarrow{\texttt{ext}}_{\lambda r} b$ then $a \xrightarrow{\texttt{ext}}{}^{+}_{\lambda r} \cdot \xrightarrow{\texttt{int}}\!\!\!\!\rightarrow_{\lambda r} b$.

Remark 4.10. The $\xrightarrow{\texttt{ext}}{}^{+}_{\lambda r}$ can represent 1 or 2 steps. The 2-step use is necessary when the internal step changes an external redex from $(a\,\sigma^n)\mathtt{n}$ to $(a'\sigma^n)\mathtt{n}$ and the external step uses (σ-*destruction*) to destroy this redex, producing $\varphi_0^n a'$. The $\xrightarrow{\texttt{int}}\!\!\!\!\rightarrow_{\lambda r}$ can represent 0, 1, or 2 steps. The 0-step case is necessary when the 2-step case of $\xrightarrow{\texttt{ext}}{}^{+}_{\lambda r}$ occurs (already mentioned) or when the internal step changes an external redex from $(a\,\sigma^i)\mathtt{n}$ to $(a'\sigma^i)\mathtt{n}$ and $i \neq n$ and the external step uses (σ-*destruction*) to destroy this redex, discarding the subterm a'. The 2-step case happens when the internal step changes an external redex from $(a\ \sigma^i)(b\,\delta)c$ to $(a'\,\sigma^i)(b\,\delta)c$ and the external step uses the (σ-δ-*transition*) rule to duplicate the σ-item producing $((a'\,\sigma^i)b\,\delta)(a'\,\sigma^i)c$.

Proof. By induction on a analysing the positions of the redexes. We give the proof for $r = gs$. The basic case which is $a = \mathtt{n}$ is trivial.

$a = (a_2\delta)a_1$ **:** Since we are dealing with an internal reduction there are only two possibilities: $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1'$ or $a_2 \xrightarrow{\texttt{int}}_{\lambda gs} a_2'$. Let us study, for instance, the first one. There are four cases:

- $a = (a_2\delta)a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2\delta)a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} (a_2\delta)a_1''$ and $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} a_1''$. Therefore, by induction hypothesis, $a_1 \xrightarrow{\texttt{ext}}{}^{+}_{\lambda gs} \cdot \xrightarrow{\texttt{int}}\!\!\!\!\rightarrow_{\lambda gs} a_1''$, and then $(a_2\delta)a_1 \xrightarrow{\texttt{ext}}{}^{+}_{\lambda gs} \cdot \xrightarrow{\texttt{int}}\!\!\!\!\rightarrow_{\lambda gs} (a_2\delta)a_1''$.
- $a = (a_2\delta)a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2\delta)a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} (a_2'\delta)a_1'$ with $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1'$ and $a_2 \xrightarrow{\texttt{ext}}_{\lambda gs} a_2'$. We can simply commute the reductions: $a = (a_2\delta)a_1 \xrightarrow{\texttt{ext}}_{\lambda gs} (a_2'\delta)a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2'\delta)a_1'$.
- $a = (a_2\delta)a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2\delta)(\lambda)a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} (a_2\sigma^1)a_1'$ with $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (\lambda)a_1'$. Hence, by remark 4.6, $a_1 = (\lambda)b_1$ with $b_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1'$. Now, $(a_2\delta)a_1 = (a_2\delta)(\lambda)b_1 \xrightarrow{\texttt{ext}}_{\lambda gs} (a_2\sigma^1)b_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2\sigma^1)a_1'$.
- $a = (a_2\delta)a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2\delta)W(\lambda)a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} W((\varphi^{\#_\lambda(W)+1})a_2\sigma^1)a_1'$ with W well-balanced and $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} W(\lambda)a_1'$. Hence, by remark 4.6.3, there are two cases:
  - Case $a_1 = W(\lambda)b_1$ where $b_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1'$. Then
  $$a = (a_2\delta)W(\lambda)b_1 \xrightarrow{\texttt{ext}}_{\lambda gs} W((\varphi^{\#_\lambda(W)+1})a_2\sigma^1)b_1 \xrightarrow{\texttt{int}}_{\lambda gs} W((\varphi^{\#_\lambda(W)+1})a_2\sigma^1)a_1'$$
  - Case $a_1 = W_1(b_1\delta)W_2(\lambda)W_3(\lambda)a_1'$ and $W = W_1(b_1'\delta)W_2(\lambda)W_3$ where W_1, W_2, W_3 well-balanced and $b_1 \xrightarrow{\texttt{int}}_{\lambda gs} b_1'$. Then,
  $$\begin{aligned} a = (a_2\delta)W_1(b_1\delta)W_2(\lambda)W_3(\lambda)a_1' &\xrightarrow{\texttt{ext}}_{\lambda gs} W_1(b_1\delta)W_2(\lambda)W_3((\varphi^{\#_\lambda(W)+1})a_2\sigma^1)a_1' \\ \xrightarrow{\texttt{int}}_{\lambda gs} W_1(b_1'\delta)W_2(\lambda)W_3((\varphi^{\#_\lambda(W)+1})a_2\sigma^1)a_1' &= W((\varphi^{\#_\lambda(W)+1}a_2)\sigma^1)a_1' \end{aligned}$$

$a = (\lambda)a_1$ **:** The reduction must take place within a_1 and we use the induction hypothesis.

$a = (a_2\,\sigma^i)a_1$ **:** Again, as we are analysing an internal reduction, two cases arise:

$a_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1'$ **:** The external reduction can only take place within a_1' or at the root:

- $a = (a_2\,\sigma^i)a_1 \xrightarrow{\texttt{int}}_{\lambda gs} (a_2\,\sigma^i)a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} (a_2\,\sigma^i)a_1''$ and $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} a_1''$. We can now apply the induction hypothesis to $a_1 \xrightarrow{\texttt{int}}_{\lambda gs} a_1' \xrightarrow{\texttt{ext}}_{\lambda gs} a_1''$ to obtain $a_1 \xrightarrow{\texttt{ext}}{}^{+}_{\lambda gs} \cdot \xrightarrow{\texttt{int}}\!\!\!\!\rightarrow_{\lambda gs} a_1''$, and hence $(a_2\,\sigma^i)a_1 \xrightarrow{\texttt{ext}}{}^{+}_{\lambda gs} \cdot \xrightarrow{\texttt{int}}\!\!\!\!\rightarrow_{\lambda gs} (a_2\,\sigma^i)a_1''$.

- $a = (a_2\,\sigma^i)a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} (a_2\,\sigma^i)a_1' \xrightarrow{\mathtt{ext}}_{\lambda gs} b$ and $a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} a_1'$, and the external reduction takes place at the root. We study the three possible rules:
  - (σ-λ-*transition*) : We have $a_1' = (\lambda)c'$ and $b = (\lambda)(a_2\sigma^{i+1})c'$. Remark 4.6.1 ensures that $a_1 = (\lambda)c$ and $c \xrightarrow{\mathtt{int}}_{\lambda gs} c'$. We can then commute:
  $$a = (a_2\,\sigma^i)a_1 = (a_2\,\sigma^i)(\lambda)c \xrightarrow{\mathtt{ext}}_{\lambda gs} (\lambda)(a_2\sigma^{i+1})c \xrightarrow{\mathtt{int}}_{\lambda gs} (\lambda)(a_2\sigma^{i+1})c' = b$$
  - (σ-δ-*transition*) : We have $a_1' = (c'\delta)d'$ and $b = ((a_2\,\sigma^i)c'\delta)(a_2\,\sigma^i)d'$. Remark 4.6.2 ensures that $a_1 = (c\delta)d$ and, either $c \xrightarrow{\mathtt{int}}_{\lambda gs} c'$ and $d = d'$, or $d \xrightarrow{\mathtt{int}}_{\lambda gs} d'$ and $c = c'$. In both cases we can commute as in the previous case.
  - (σ-*destruction*) : We have $a_1' = \mathtt{n}$ and this is impossible by Remark 4.6.4.

$a_2 \to_{\lambda gs} a_2'$ **:** As in the previous case, the external reduction can take place within a_1 or at the root:

- $a = (a_2\,\sigma^i)a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} (a_2'\,\sigma^i)a_1 \xrightarrow{\mathtt{ext}}_{\lambda gs} (a_2'\,\sigma^i)a_1'$ and and $a_1 \xrightarrow{\mathtt{ext}}_{\lambda gs} a_1'$. We can commute to obtain: $a = (a_2\,\sigma^i)a_1 \xrightarrow{\mathtt{ext}}_{\lambda gs} (a_2\,\sigma^i)a_1' \xrightarrow{\mathtt{int}}_{\lambda gs} (a_2'\,\sigma^i)a_1'$.
- $a = (a_2\,\sigma^i)a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} (a_2'\,\sigma^i)a_1 \xrightarrow{\mathtt{ext}}_{\lambda gs} b$ and the external reduction takes place at the root. We study the three possible rules:
  - (σ-λ-*transition*) : We have $a_1 = (\lambda)c$ and $b = (\lambda)(a_2'\sigma^{i+1})c$. We can commute:
  $$a = (a_2\,\sigma^i)a_1 = (a_2\,\sigma^i)(\lambda)c \xrightarrow{\mathtt{ext}}_{\lambda gs} (\lambda)(a_2\sigma^{i+1})c \xrightarrow{\mathtt{int}}_{\lambda gs} (\lambda)(a_2'\sigma^{i+1})c = b$$
  - (σ-δ-*trans*) : We have $a_1 = (c\delta)d$ and $b = ((a_2'\,\sigma^i)c\delta)(a_2'\,\sigma^i)d$. We can commute generating two internal steps:
  $$\begin{aligned} a = (a_2\,\sigma^i)a_1 = (a_2\,\sigma^i)(c\delta)d &\xrightarrow{\mathtt{ext}}_{\lambda gs} ((a_2\,\sigma^i)c\delta)(a_2\,\sigma^i)d \\ \xrightarrow{\mathtt{int}}_{\lambda gs} ((a_2'\,\sigma^i)c\delta)(a_2\,\sigma^i)d &\xrightarrow{\mathtt{int}}_{\lambda gs} ((a_2'\,\sigma^i)c)(a_2'\,\sigma^i)d = b \end{aligned}$$
  - (σ-*destruction*) : We have $a_1 = \mathtt{n}$. If $n > i$ then $b = \mathtt{n}-\mathtt{1}$. But $(a_2\,\sigma^i)\mathtt{n} \xrightarrow{\mathtt{ext}}_{\lambda gs} \mathtt{n}-\mathtt{1}$. If $n < i$ then $b = \mathtt{n}$. But $(a_2\,\sigma^i)\mathtt{n} \xrightarrow{\mathtt{ext}}_{\lambda gs} \mathtt{n}$. If $n = i$ then $b = (\varphi_0^i)a_2'$. We must now consider whether $a_2 \to_{\lambda s} a_2'$ is external or internal. If it is internal we can commute to obtain:
  $$a = (a_2\,\sigma^i)a_1 = (a_2\,\sigma^i)\mathtt{n} \xrightarrow{\mathtt{ext}}_{\lambda gs} (\varphi_0^i)a_2 \xrightarrow{\mathtt{int}}_{\lambda gs} (\varphi_0^i)a_2' = b$$
  If it is external, we get:
  $$a = (a_2\,\sigma^i)a_1 = (a_2\,\sigma^i)\mathtt{n} \xrightarrow{\mathtt{ext}}_{\lambda gs} (\varphi_0^i)a_2 \xrightarrow{\mathtt{ext}}_{\lambda gs} (\varphi_0^i)a_2' = b$$
  giving us $a \xrightarrow{\mathtt{ext}}{}^{+}_{\lambda gs} b \xrightarrow{\mathtt{int}}\!\!\!\!\to_{\lambda gs} b$.

$a = (\varphi_k^i)a_1$ **:** Two possibilities according to the position of the external reduction.

- $(\varphi_k^i)a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} (\varphi_k^i)a_1' \xrightarrow{\mathtt{ext}}_{\lambda gs} (\varphi_k^i)a_1''$ and $a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} a_1' \xrightarrow{\mathtt{ext}}_{\lambda gs} a_1''$. Use induction hypothesis.
- $(\varphi_k^i)a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} (\varphi_k^i)a_1' \xrightarrow{\mathtt{ext}}_{\lambda gs} b$ and $a_1 \xrightarrow{\mathtt{int}}_{\lambda gs} a_1'$ and the external reduction takes place at the root. Three rules are possible:
  - *(φ-λ-trans)* : We have $a_1' = (\lambda)c'$ and $b = (\lambda)(\varphi_{k+1}^i)c'$. Remark 4.6.1 ensures that $a_1 = (\lambda)c$ and $c \xrightarrow{\mathtt{int}}_{\lambda gs} c'$. We can then commute:
  $$a = (\varphi_k^i)a_1 = (\varphi_k^i)(\lambda)c \xrightarrow{\mathtt{ext}}_{\lambda gs} (\lambda)(\varphi_{k+1}^i)c \xrightarrow{\mathtt{int}}_{\lambda gs} (\lambda)(\varphi_{k+1}^i)c' = b$$
  - *(φ-δ-trans)* : We have $a_1' = (c'\delta)d'$ and $b = ((\varphi_k^i c')\delta)(\varphi_k^i)d'$. Remark 4.6.2 ensures that $a_1 = (c\delta)d$ and, either $c \xrightarrow{\mathtt{int}}_{\lambda gs} c'$ and $d = d'$, or $d \xrightarrow{\mathtt{int}}_{\lambda gs} d'$ and $c = c'$. In both cases we can commute as in the previous case.
  - *(φ-dest)* : We have $a_1' = \mathtt{n}$ and this is impossible by Remark 4.6.4. □

Lemma 4.11. *Let $a \in \Lambda s$ and $r \in \{s, gs\}$. For every infinite λr-reduction sequence $a \to_{\lambda r} b_1 \to_{\lambda r} \cdots \to_{\lambda r} b_n \to_{\lambda r} \cdots$, one of these two possibilities holds:*

1. *There exists N such that for $i \geq N$ it holds that $b_i \xrightarrow{\texttt{int}}_{\lambda r} b_{i+1}$, i.e., all the reductions beyond the Nth step are internal.*
2. *There exists an infinite external λr-reduction sequence:*

$$a \xrightarrow{\texttt{ext}}_{\lambda r} c_1 \xrightarrow{\texttt{ext}}_{\lambda r} \cdots \xrightarrow{\texttt{ext}}_{\lambda r} c_n \xrightarrow{\texttt{ext}}_{\lambda r} \cdots$$

Proof. Suppose there are an infinite number of external steps in the given reduction sequence. Then by repeated use of the Commutation Lemma (4.9), we construct an infinite external reduction sequence starting from a. Otherwise, there is some N such that all steps past the Nth are internal. $\square$

In order to prove the Preservation Theorem (Theorem 4.15) we need two definitions.

Definition 4.12. Let $r \in \{s, gs\}$. An infinite λr-reduction sequence $a_1 \to \cdots \to a_n \to \cdots$ is *minimal* if for every step $a_i \underset{p}{\to}_{\lambda r} a_{i+1}$, every other reduction sequence beginning with $a_i \underset{q}{\to}_{\lambda r} a'_{i+1}$ where $p \prec q$, is finite.

The idea of a minimal reduction sequence is that at every step, it picks a redex as deeply nested as possible without preventing an infinite reduction. If one changes any one of its steps to rewrite a redex within a subterm of the original redex, then an infinite reduction sequence is impossible.

Definition 4.13. The syntax of *skeletons* and the *skeleton of a term* are defined as follows:

$$\textbf{Skeletons} \qquad K ::= \mathbb{N} \mid (K\,\delta)K \mid (\lambda)K \mid ([\,\cdot\,]\,\sigma^j)K \mid (\varphi^i_k)K$$

$$\begin{array}{lll} Sk(\texttt{n}) = \texttt{n} & Sk((a\,\delta)b) = (Sk(a)\,\delta)Sk(b) & Sk((b\,\sigma^i)a) = ([\,\cdot\,]\,\sigma^i)Sk(a) \\ & Sk((\lambda)a) \ = (\lambda)Sk(a) & Sk((\varphi^i_k)a) \ = (\varphi^i_k)Sk(a) \end{array}$$

Remark 4.14. A definition of internal and external reduction equivalent to definitions 4.5 and 4.7 is the following. Let $a, b \in \Lambda s$.

$$a \xrightarrow{\texttt{int}}_r b \Leftrightarrow \ (a \to_r b \text{ and } Sk(a) = Sk(b))$$
$$a \xrightarrow{\texttt{ext}}_r b \Leftrightarrow \ (a \to_r b \text{ and } Sk(a) \neq Sk(b))$$

In other words, skeletons provide a syntax that is informative about what kind of r-reduction takes place. In particular, the following two properties hold:

1. Each occurrence of $[\,\cdot\,]$ in $Sk(a)$ corresponds to an external closure of a (i.e. a closure that is not at the left of any other closure), and this correspondence is a bijection.
2. Internal closures (those which are at the left of another closure) vanish in the skeleton.

Theorem 4.15 (Preservation of Strong Normalisation). *Let $a \in \Lambda$. The following hold:*

1. *If a is strongly normalising in the λ-calculus then a is strongly normalising in the λs-calculus.*
2. *If a is strongly normalising in the λg-calculus then a is strongly normalising in the λgs-calculus.*

Proof. The proof of 1 is obtained by replacing in the proof below, λg by λ, λgs by λs and by dropping the second case that lemma 4.4 gives. We prove 2.

Assume $a \in \lambda g$-SN, $a \notin \lambda gs$-SN and take a minimal infinite λgs-reduction sequence

$$\mathcal{D} : a \to_{\lambda gs} a_1 \to_{\lambda gs} \cdots \to a_n \to_{\lambda gs} \cdots$$

Lemma 4.11 gives N such that for $i \geq N$, $a_i \to_{\lambda gs} a_{i+1}$ is internal. (Note that case 2 of Lemma 4.11 cannot hold. Otherwise, by Proposition 4.8, there would be an infinite λg-reduction sequence starting at a and hence, $a \notin \lambda g$-SN. Contradiction.) By Remark 4.14, $Sk(a_i) = Sk(a_{i+1})$ for $i \geq N$. As there are only a finite number of closures in $Sk(a_N)$ and as the reductions within these closures are independent, an infinite reduction sequence $\mathcal{D}'$ can be formed by taking steps from $\mathcal{D}$ such that all steps take place within a single closure in $Sk(a_N)$ and $\mathcal{D}'$ is also minimal. Let C be the context such that $a_N = C[(d\,\sigma^i)c]$ and $(d\,\sigma^i)c$ is the closure where $\mathcal{D}'$ takes place:

$$\mathcal{D}' : a_N = C[(d\,\sigma^i)c] \xrightarrow{\texttt{int}}_{\lambda gs} C[(d_1\,\sigma^i)c] \xrightarrow{\texttt{int}}_{\lambda gs} \cdots \xrightarrow{\texttt{int}}_{\lambda gs} C[(d_n\,\sigma^i)c] \xrightarrow{\texttt{int}}_{\lambda gs} \cdots$$

Since a is a pure term, Lemma 4.4 ensures the existence of $I \leq N$ such that one of the following holds:

1. $a_I = C'[(d'\,\delta)(\lambda)c'] \to_{\lambda gs} a_{I+1} = C'[(d'\sigma^1)c']$ and $d' \twoheadrightarrow_{\lambda gs} d$
2. $a_I = C'[(d'\,\delta)W(\lambda)c'] \to_{\lambda gs} a_{I+1} = C'[W((\varphi_0^{\#_\lambda(W)+1})d'\sigma^1)c']$ and $d' \twoheadrightarrow_{\lambda gs} d$

Let us consider in the first and second cases respectively, the following infinite λgs-reduction sequences:

$$\begin{array}{ll}\mathcal{D}'': & a \twoheadrightarrow_{\lambda gs} a_I \twoheadrightarrow_{\lambda gs} C'[(d\delta)(\lambda)c'] \to_{\lambda gs} C'[(d_1\delta)(\lambda)c'] \to_{\lambda gs} \cdots \to_{\lambda gs} C'[(d_n\delta)(\lambda)c'] \to_{\lambda gs} \cdots \\ \mathcal{D}''': & a \twoheadrightarrow_{\lambda gs} a_I \twoheadrightarrow_{\lambda gs} C'[(d\delta)W(\lambda)c'] \to_{\lambda gs} C'[(d_1\delta)W(\lambda)c'] \to_{\lambda gs} \cdots \to_{\lambda gs} C'[(d_n\delta)W(\lambda)c'] \to_{\lambda gs} \cdots\end{array}$$

In $\mathcal{D}''$ and $\mathcal{D}'''$, the redex in a_I is within d' which is a proper subterm of $(d'\,\delta)(\lambda)c'$ (of $(d'\,\delta)W(\lambda)c'$ in the second case), whereas in $\mathcal{D}'$ the redex in a_I is $(d'\,\delta)(\lambda)c'$ (in the second case $(d'\,\delta)W(\lambda)c'$) and this contradicts the minimality of $\mathcal{D}'$. □

Theorem 4.16. *For every $a \in \Lambda$, the following equivalences hold:*

$$a \in \lambda\text{-}SN \Leftrightarrow a \in \lambda s\text{-}SN \qquad \textit{and} \qquad a \in \lambda g\text{-}SN \Leftrightarrow a \in \lambda gs\text{-}SN$$

Proof. By Lemma 3.12 and Theorem 4.15. □

In order to complete the picture, we need to use a result of [19]:

Theorem 4.17. *Let $a \in \Lambda$. It holds that: $a \in \lambda\text{-}SN \Leftrightarrow a \in \lambda g\text{-}SN$*

Corollary 4.18. *For every $a \in \Lambda$, the following equivalences hold:*

$$a \in \lambda g\text{-}SN \Leftrightarrow a \in \lambda\text{-}SN \Leftrightarrow a \in \lambda s\text{-}SN \Leftrightarrow a \in \lambda gs\text{-}SN$$

Note that the main preservation results that we show in this paper (Theorem 4.15) are concerned with substitution calculi. That is, we show that if $a \in \lambda r$-SN then $a \in \lambda rs$-SN for $\lambda r \in \{\lambda, \lambda g\}$. What we do not show in this paper is the preservation result concerned with generalised reduction. That is, we do not prove $a \in \lambda$-SN $\Rightarrow a \in \lambda g$-SN. Rather, we take the result of [19]. The reason for this is that the minimal derivation method and even our adaptation of it are not suited to prove PSN for calculi that do not explicit substitutions. In fact, the whole idea of internal and external reduction and of skeletons is based around substitutions. It is also fair to say that generalised reduction did not play any role in the proof of PSN (despite its role in proofs of SN as shown in [30, 31, 39]).

5 The typed λs- and λgs-calculi

Our calculi of explicit substitutions λs and λgs possess a very nice property that other calculi of explicit substitutions do not possess. Namely, the simply typed versions of λs and λgs are strongly normalising. The $\lambda\sigma$-calculus of [15] does not possess this property as is shown by Melliès in [35] and only very recently its weak normalisation on open terms has been shown to hold in [15]. The simply typed $\lambda\upsilon$ of [5] is strongly normalising however, it is not confluent on open terms. In fact, our calculi λs and λgs are the first calculi of explicit substitutions whose simply typed versions are strongly normalising (cf. [24, 25]) and which possess a confluent extension on open terms (we have shown the confluence of the extension of λs on open terms; although the extension for λgs on open terms has not yet been investigated, we believe that the details are similar to those for λs).

In this section, we present the simply typed versions of λs and λgs and prove the strong normalisation of the well typed terms using the technique developed in [24] to prove λs-SN and suggested to us by P.-A. Melliès as a successful technique to prove $\lambda\upsilon$-SN (personal communication).

We recall the syntax and typing rules for the simply typed λ-calculus in de Bruijn notation. The types are generated from a set of basic types T with the binary type operator $\to$. Environments are lists of types. Typed terms differ from the untyped ones only in the abstractions which are now marked with the type of the abstracted variable.

Definition 5.1 (Λ_t and L1). The syntax for the simply typed λ-terms is given as follows:

$$\begin{array}{ll} \textbf{Types} & \mathcal{T} ::= T \mid \mathcal{T} \to \mathcal{T} \\ \textbf{Environments} & \mathcal{E} ::= nil \mid \mathcal{T}, \mathcal{E} \\ \textbf{Terms} & \Lambda_t ::= \mathbb{N} \mid (\Lambda_t\, \delta)\Lambda_t \mid (\mathcal{T}\, \lambda)\Lambda_t \end{array}$$

We let A, B, ... range over $\mathcal{T}$; E, E_1, ... over $\mathcal{E}$; and a, b, ... over Λ_t. The typing rules are given by the typing system **L1** as follows:

$$(\mathbf{L1}\text{-}var) \quad A, E \vdash \mathtt{1} : A \qquad\qquad (\mathbf{L1}\text{-}abs) \quad \frac{A, E \vdash b : B}{E \vdash (A\, \lambda)b : A \to B}$$

$$(\mathbf{L1}\text{-}varn) \quad \frac{E \vdash \mathtt{n} : B}{A, E \vdash \mathtt{n+1} : B} \qquad\qquad (\mathbf{L1}\text{-}app) \quad \frac{E \vdash b : A \to B, \quad E \vdash a : A}{E \vdash (a\, \delta)b : B}$$

Before presenting the simply typed λs-calculus and λsg-calculus we introduce the following notation concerning environments. If E is the environment $A_1, A_2, \ldots, A_n$, we shall use the notation $E_{\geq i}$ for the environment $A_i, A_{i+1}, \ldots, A_n$. Analogously, $E_{\leq i}$ stands for $A_1, \ldots, A_i$. The notations $E_{<i}$ and $E_{>i}$ are defined similarly.

Definition 5.2 (Λs_t and Ls1). The syntax for the *simply typed λs-terms* is given as follows:

$$\Lambda s_t ::= \mathbb{N} \mid (\Lambda s_t\, \delta)\Lambda s_t \mid (\mathcal{T}\, \lambda)\Lambda s_t \mid (\Lambda s_t\, \sigma^i)\Lambda s_t \mid (\varphi^i_k)\Lambda s_t \quad \text{where } i \geq 1,\ k \geq 0$$

Types and environments are as above. The typing rules of the *system* **Ls1** are as follows. The rules **Ls1**-*var*, **Ls1**-*varn*, **Ls1**-abs and **Ls1**-*app* are exactly the same as **L1**-*var*, **L1**-*varn*, **L1**-abs and **L1**-*app*, respectively. The new rules are:

$$(\mathbf{Ls1}\text{-}\sigma) \quad \frac{E_{\geq i} \vdash b : B, \quad E_{<i}, B, E_{\geq i} \vdash a : A}{E \vdash (b\, \sigma^i)a : A} \qquad\qquad (\mathbf{Ls1}\text{-}\varphi) \quad \frac{E_{\leq k}, E_{\geq k+i} \vdash a : A}{E \vdash (\varphi^i_k)a : A}$$

The reduction rules of the *simply typed* λs- and λsg-*calculi* are given by the same rules of the corresponding untyped versions, except that abstractions in the typed versions are marked with types.

We say that $a : A$ is derivable in some type system $\mathbf{X} \in \{\mathbf{L1}, \mathbf{Ls1}\}$ from an environment E, notation $E \vdash_{\mathbf{X}} a : A$ if and only if $E \vdash a : A$ can be produced by the typing rules of the system $\mathbf{X}$. We say that $a \in \Lambda s_t$ is *well typed* if there exists an environment E and a type A such that $E \vdash_{\mathbf{Ls1}} a : A$. The symbol Λs_{wt} denotes the set of well typed terms.

The aim of this section is to prove that every well typed λs-term a is λgs-SN (and hence λs-SN). To do so, we show $\Lambda s_{wt} \subseteq \Xi \subseteq \lambda gs$-SN, where

$$\Xi = \{\, a \in \Lambda s_t \mid \text{for every subterm } b \text{ of } a,\ s(b) \in \lambda g\text{-SN} \,\}$$

To prove $\Lambda s_{wt} \subseteq \Xi$ (Proposition 5.10) we need to establish some useful results such as subject reduction, soundness of typing (i.e., **Ls1** is a conservative extension of **L1**), and typing of subterms:

Lemma 5.3. *Let E a type environment, A, B types and $a, b, c \in \Lambda s_t$. The following holds:*

1. $E \vdash ((\varphi^i_0)a\, \delta)(c\, \delta)(B\, \lambda)b : A$ *iff* $E \vdash (c\, \delta)(B\, \lambda)((\varphi^{i+1}_0)a\, \delta)b : A$
2. $E \vdash (\varphi^i_0)(\varphi^j_0)a : A$ *iff* $E \vdash (\varphi^{i+j-1}_0)a : A$
3. $E \vdash (a\, \delta)(B\, \lambda)b : A$ *iff* $E \vdash (a\, \sigma^1)b : A$

Proof. We supply only the proof of the first item, since the others are similar.

$$\begin{array}{llll}
 & E \vdash ((\varphi_0^i)a\,\delta)(c\,\delta)(B\,\lambda)b : A & & \\
\text{iff} & \exists \mathrm{C}.\ (E \vdash (c\,\delta)(B\,\lambda)b : C \to A & & \text{and}\ \ E \vdash (\varphi_0^i)a : C) \\
\text{iff} & \exists \mathrm{C}.\ (E \vdash (B\,\lambda)b : B \to (C \to A) & \text{and}\ \ E \vdash c : B & \text{and}\ \ E_{\geq i} \vdash a : C) \\
\text{iff} & \exists \mathrm{C}.\ (B, E \vdash b : C \to A & \text{and}\ \ E \vdash c : B & \text{and}\ \ (B,\, E)_{\geq i+1} \vdash a : C) \\
\text{iff} & \exists \mathrm{C}.\ (B, E \vdash b : C \to A & \text{and}\ \ E \vdash c : B & \text{and}\ \ B, E \vdash (\varphi_0^{i+1})a : C) \\
\text{iff} & \qquad (B, E \vdash ((\varphi_0^{i+1})a\,\delta)b : A & \text{and}\ \ E \vdash c : B) & \\
\text{iff} & \qquad (E \vdash (B\,\lambda)((\varphi_0^{i+1})a\,\delta)b : B \to A & \text{and}\ \ E \vdash c : B) & \\
\text{iff} & \qquad\ E \vdash (c\,\delta)(B\,\lambda)((\varphi_0^{i+1})a\,\delta)b : A & & \qquad\qquad \square
\end{array}$$

Lemma 5.4. *Let C be a context and $a, b \in \Lambda s_t$. E will range over type environments and A over types. The following holds:*

$$(\forall E, A.\ (E \vdash a : A \Leftrightarrow E \vdash b : A)) \Rightarrow (\forall E, A.\ (E \vdash C[a] : A \Leftrightarrow E \vdash C[b] : A))$$

Proof. By induction on C. □

The following lemma is necessary in the case of generalised reduction.

Lemma 5.5 (Shuffle Lemma). *Let S be an arbitrary segment, W a well balanced segment and $a, b \in \Lambda s_t$, then $E \vdash S(a\,\delta)W\,b : A$ iff $E \vdash SW((\varphi_0^{\#_\lambda(W)+1})a\,\delta)\,b : A$.*

Proof. By induction on W using Lemmas 5.3 and 5.4. If $W = \phi$, it is immediate since $E' \vdash d : D$ iff $E' \vdash (\varphi_0^1)d : D$. Let us assume $W = (c\,\delta)U(B\,\lambda)V$, with U, V well balanced. The following statements are equivalent:

$$\begin{array}{lll}
 & & E \vdash S(a\,\delta)(c\,\delta)U(B\,\lambda)V\,b : A \\
\text{(IH)} & \text{iff} & E \vdash S(a\,\delta)U((\varphi_0^{\#_\lambda(U)+1})c\,\delta)(B\,\lambda)V\,b : A \\
\text{(IH)} & \text{iff} & E \vdash SU((\varphi_0^{\#_\lambda(U)+1})a\,\delta)((\varphi_0^{\#_\lambda(U)+1})c\,\delta)(B\,\lambda)V\,b : A \\
\text{(5.3.1, 5.4)} & \text{iff} & E \vdash SU((\varphi_0^{\#_\lambda(U)+1})c\,\delta)(B\,\lambda)((\varphi_0^{\#_\lambda(U)+2})a\,\delta)V\,b : A \\
\text{(IH, twice)} & \text{iff} & E \vdash S(c\,\delta)U(B\,\lambda)V((\varphi_0^{\#_\lambda(V)+1})(\varphi_0^{\#_\lambda(U)+2})a\,\delta)b : A \\
\text{(5.3.2, 5.4)} & \text{iff} & E \vdash S(c\,\delta)U(B\,\lambda)V((\varphi_0^{\#_\lambda(V)+\#_\lambda(U)+2})a\,\delta)b : A \qquad \square
\end{array}$$

Lemma 5.6 (Subject Reduction). *Let $r \in \{s, gs\}$. If $E \vdash_{\mathbf{Ls1}} a : A$ and $a \to_{\lambda r} b$ then $E \vdash_{\mathbf{Ls1}} b : A$.*

Proof. By induction on a. If the reduction is not at the root, use IH. If it is, check that for each rule $a \to b$ we have $E \vdash_{\mathbf{Ls1}} a : A$ implies $E \vdash_{\mathbf{Ls1}} b : A$. For the case of ($\sigma$*-generation*), use Lemma 5.3.3. For the case of ($g\sigma$*-generation*) and $r = gs$, if $E \vdash (a\,\delta)W(B\,\lambda)b : A$ then, by Lemma 5.5, we have $E \vdash W((\varphi_0^{\#_\lambda(W)+1})a\,\delta)(B\,\lambda)b : A$ and, by Lemma 5.3.3, we conclude $E \vdash W((\varphi_0^{\#_\lambda(W)+1})a\,\sigma^1)b : A$. The other rules are proven by similar reasoning. □

Corollary 5.7. *Let $r \in \{s, gs\}$ and $E \vdash_{\mathbf{Ls1}} a : A$. If $a \twoheadrightarrow_{\lambda r} b$ then $E \vdash_{\mathbf{Ls1}} b : A$.*

Lemma 5.8 (Typing of Subterms). *If $a \in \Lambda s_{wt}$ and $b \triangleleft a$ then $b \in \Lambda s_{wt}$.*

Proof. By induction on a. If b is not an immediate subterm of a, use the induction hypothesis. Otherwise, the last rule used to type a must contain a premise in which b is typed. □

Lemma 5.9 (Conservative Extension of Typing). *If $a \in \Lambda_t$ and $E \vdash_{\mathbf{Ls1}} a : A$ then $E \vdash_{\mathbf{L1}} a : A$.*

Proof. Easy induction on a. □

Proposition 5.10. $\Lambda s_{wt} \subseteq \Xi$.

Proof. Let $a \in \Lambda s_{wt}$ and b a subterm of a. By Lemma 5.8, $b \in \Lambda s_{wt}$ and by Corollary 5.7, $s(b) \in \Lambda s_{wt}$. Since $s(b) \in \Lambda$ (Thm. 3.7), Lemma 5.9 yields that $s(b)$ is **L1**-typable, and it is well known that classical typable λ-terms are strongly normalising in the λ-calculus. Hence, $s(b) \in \lambda$-SN and, by preservation (Corollary 4.18), $s(b) \in \lambda g$-SN. Therefore $a \in \Xi$. □

Proposition 5.11. $\Xi \subseteq \lambda gs$*-SN.*

Proof. Suppose there exists $a' \in \Xi$ and $a' \notin \lambda gs$-SN, then there must exist a term a of minimal size such that $a \in \Xi$ and $a \notin \lambda gs$-SN. Let us consider a minimal infinite λgs-derivation $\mathcal{D} : a \to a_1 \to \cdots \to a_n \to \cdots$ and follow the proof of Theorem 4.15 to obtain:

$$\mathcal{D}' : a_N = C[(d\,\sigma^i)c] \xrightarrow{\texttt{int}}_{\lambda gs} C[(d_1\,\sigma^i)c] \xrightarrow{\texttt{int}}_{\lambda gs} \cdots \xrightarrow{\texttt{int}}_{\lambda gs} C[(d_n\,\sigma^i)c] \xrightarrow{\texttt{int}}_{\lambda gs} \cdots$$

(Again, as we argued in Theorem 4.15, case 2 of Lemma 4.11 is discarded. This is because, $a \in \Xi \Rightarrow s(a) \in \lambda g$-SN and Proposition 4.8 would yield a contradiction if case 2 of lemma 4.11 holds.) Now three possibilities arise from Lemma 4.4. Two of them have been considered in the proof of Theorem 4.15 and contradicted the minimality of $\mathcal{D}$. Take the third one, that $a = C'[(d'\,\sigma^i)c']$ where $d' \twoheadrightarrow d$. Now we have $d' \twoheadrightarrow d \to d_1 \to \cdots \to d_n \to \cdots$. Since d' is a subterm of a, it must be the case that $d' \in \Xi$, contradicting our choice of a with minimal size. □

Therefore we conclude, using Propositions 5.10 and 5.11 and Corollary 4.18:

Theorem 5.12. *Every well typed* λs*-term is strongly normalising in the* λgs*-calculus.*

Corollary 5.13. *Every well typed* λs*-term is strongly normalising in the* λs*-calculus.*

6 Conclusion

In this paper, we first explained the relevance and benefits of both generalized reduction and explicit substitution. We discussed alternatives and presented the research history behind the two concepts. We explained that they have never been combined together and we commented that the combination might indeed join both benefits and hence a λ-calculus extended with both needed to be studied.

Then we introduced λg, the first system of generalized reduction using de Bruijn indices. We proved λg confluent and sound (i.e., preserves β-equivalence) and complete (properly contains β-reduction) with respect to the ordinary λ-calculus with de Bruijn indices.

Building on this success, we then introduced λgs, the first system combining generalized reduction with explicit substitution. For this combination, we relied on the proven explicit substitution technology of the λs-calculus. We proved λgs sound (a conservative extension) and complete (simulating $g\beta$-reduction) with respect to λg. We proved that λgs preserves the strong normalization (PSN) of terms which are terminating in λg, λs, and the ordinary λ-calculus (all with de Bruijn indices). Our proof of PSN included the first proof of the commutation of arbitrary external and internal reductions and is simpler than the standard proof of PSN using the traditional method of minimal derivations.

We proceeded further and added simple types to both λs and λgs (simultaneously) in the form of the type system **Ls1**. By proving the strong normalization (SN) of well typed terms under λgs-reduction (and therefore also under λs-reduction), we have provided the first typed systems of explicit substitution and generalised reduction with the SN property. We proved λgs (and therefore also λs) has the essential property of subject reduction with respect to **Ls1** typing. We also proved typing for **Ls1** is sound (a conservative extension) and complete (a proper extension) with respect to the simply typed λ-calculus and has typing of subterms.

Now that a calculus combining both concepts have been shown to be theoretically correct, it would be interesting to extend our calculus λgs to one that is confluent on open terms as is the tradition with calculi of explicit substitution. It would be also interesting to study the polymorphically (rather than the simply) typed version of λgs. These are issues we are investigating at the moment. We are also investigating the correspondence of our calculus to methods that implement sharing to test if the analysis of sharing given in [2] can be recast in an elegant fashion in our calculus.

References

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

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Conf. Rec. 22nd Ann. ACM Symp. Principles Programming Languages*, 1995.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.

[4] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds., *Handbook of Logic in Computer Science*, vol. 2, chapter 2, pp. 117–309. Oxford University Press, 1992.

[5] Z.-E.-A. Benaissa, D. Briaud, P. Lescanne, and J. Rouyer-Degli. λv, a calculus of explicit substitutions which preserves strong normalisation. *J. Funct. Prog.*, 6(5), 1996.

[6] R. Bloo. Preservation of strong normalisation for explicit substitution. Technical Report 95-08, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1995.

[7] R. Bloo, F. Kamareddine, and R. Nederpelt. The Barendregt cube with definitions and generalised reduction. *Inf. & Comput.*, 126(2):123–143, May 1996.

[8] R. Constable et al. *Implementing Mathematics with the NUPRL Development System.* Prentice-Hall, 1986.

[9] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, 1986. Revised edition published by Birkhäuser in 1993.

[10] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, Mar. 1996.

[11] P.-L. Curien and T. Hardin and A. Ríos. Strong Normalisation of Substitutions. Proceedings of MFCS'92, Lecture Notes in Computer Science 629, pages 209-217, ed I.M. Havel and V. Koubek, Springer-Verlag, 1992.

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

[13] P. de Groote. The conservation theorem revisited. In *Int'l Conf. Typed Lambda Calculi and Applications*, vol. 664 of *LNCS*, pp. 163–178. Springer-Verlag, Mar. 1993.

[14] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[15] Jean Goubault-Larrecq. A Proof of Weak Termination of the Simply Typed $\lambda\sigma$-Calculus. Technical Report No 3090, INRIA, January 1997.

[16] T. Hardin. Confluence results for the pure strong categorical logic CCL: λ-calculi as subsystems of CCL. *Theor. Comp. Sc.*, 65(2):291–342, 1989.

[17] T. Hardin and A. Laville. Proof of Termination of the Rewriting System SUBST on CCL. *Theoretical Computer Science* 46:305–312, 1986.

[18] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting System. *Journal of the Association for Computing Machinery* 27:797–821, 1980.

[19] F. Kamareddine. A reduction relation for which postponement of k-contractions, conservation and preservation of strong normalisation hold. Technical Report TR-1996-11, Univ. of Glasgow, Glasgow G12 8QQ, Scotland, Mar. 1996.

[20] F. Kamareddine and R. Nederpelt. On stepwise explicit substitution. *International Journal of Foundations of Computer Science*, 4(3):197–240, 1993.

[21] F. Kamareddine and R. Nederpelt. Generalising reduction in the λ-calculus. *J. Funct. Prog.*, 5(4):637–651, 1995.

[22] F. Kamareddine and R. Nederpelt. A useful λ-notation. *Theoretical Computer Science*, 155:85–109, 1996.

[23] F. Kamareddine and A. Ríos. A λ-calculus à la de Bruijn with explicit substitution. In *7th International Symppsium of Programming Languages: Implementation, Logics & Programs, PLILP '95*, vol. 982 of *LNCS*, pp. 45–62. Springer-Verlag, 1995.

[24] F. Kamareddine and A. Ríos. The λs-calculus: Its typed and its extended versions. Technical Report TR-95-13, Department of Computing Science, University of Glasgow, 1995.

[25] F. Kamareddine and A. Ríos. A Generalised β-reduction and Explicit Substitutions. In *8th International Symppsium of Programming Languages: Implementation, Logics & Programs, PLILP '96*, vol. 1140 of *LNCS*, pp. 378–392. Springer-Verlag, 1996.

[26] F. Kamareddine and A. Ríos. Extending a λ-calculus with explicit substitution which preserves strong normalisation into a confluent calculus on open terms. *Journal of Functional Programming*, 7(4), 1997. To appear.

[27] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of ACM*, 41(2):368–398, March 1994.

[28] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proceedings 1994 ACM Conference LISP Functional Programmming*, 1994.

[29] A. J. Kfoury and J. B. Wells. Addendum to "New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi". Tech. Rep. 95-007, Comp. Sci. Dept., Boston Univ., 1995.

[30] A. J. Kfoury and J. B. Wells. New notions of reduction and non-semantic proofs of β-strong normalization in typed λ-calculi. In *Proc. 10th Ann. IEEE Symp. Logic in Computer Sci.*, pp. 311–321, 1995.

[31] J. W. Klop. *Combinatory Reduction Systems.* Number 127 in Mathematical Centre Tracts. Mathematisch Centrum, Amsterdam, 1980. Ph.D. Thesis.

[32] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. *Computational Problems in Abstract Algebra*, ed J. Leech, 263-297, Pergamon Press, 1970.

[33] J. L. Lawall and H. Mairson. Optimality and inefficiency: What isn't a cost model of the lambda calculus? In *Proc. 1996 ACM SIGPLAN Int'l Conf. Functional Programming*, pp. 92–101, 1996.

[34] L. Magnusson. *The Implementation of ALF – A proof editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Goteborg University, Jan. 1995.

[35] P.-A. Melliès. Typed λ-calculi with explicit substitutions may not terminate. In *Second Int'l Conf. Typed Lambda Calculi and Applications.* Springer-Verlag, Apr. 1995.

[36] C. A. Muñoz Hurtado. Confluence and preservation of strong normalisation in an explicit substitutions calculus. In *Proc. 11th Ann. IEEE Symp. Logic in Computer Sci.*, pp. 440–447, 1996.

[37] C. A. Muñoz Hurtado. Proof representation in type theory: State of the art. In *Proceedings, XXII Latin-American Conference of Informatics CLEI Panel '96*, Santafé de Bogotá, Colombia, June 1996.

[38] R. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected Papers on Automath.* North-Holland, Amsterdam, 1994.

[39] R. P. Nederpelt. *Strong Normalization for a Typed Lambda Calculus with Lambda Structured Types.* PhD thesis, Technische Hogeschool Eindhoven, 1973. Appears as a chapter in [38].

[40] S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[41] L. Regnier. *Lambda Calcul et Réseaux.* PhD thesis, Université de Paris VII, 1992.

[42] L. Regnier. Une équivalence sur les lambda-termes. *Theor. Comp. Sc.*, 126:281–292, 1994. In French.

[43] A. Ríos. *Contribution à l'Étude des λ-Calculs avec Substitutions Explicites.* PhD thesis, Université de Paris VII, 1993.

[44] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3/4):289, Nov. 1993.

[45] M. H. Sørensen. Strong normalization from weak normalization in typed λ-calculi. *Journal of Information and Comuptation.* To appear.

[46] D. Vidal. *Nouvelles Notions de Réduction en Lambda-Calcul.* Thèse de Doctorat, Université de Nancy 1, Feb. 1989.

[47] H. Xi. On weak and strong normalizations. Technical Report 96-187, Carnegie Mellon University, 1996.

[48] H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation* 17(1): 23–50, 1994.