Characterizing lambda-terms with equal reduction behavior

Citation for published version (APA):
Characterizing $\lambda$-terms with equal reduction behavior

by

Fairouz Kamareddine and Roel Bloo and Rob Nederpelt

00/16

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
        prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/cs
Characterizing $\lambda$-terms with equal reduction behavior*

Fairouz Kamareddine$^1$ and Roel Bloo$^2$ and Rob Nederpelt$^2$

1 Computing and Electrical Engineering, Heriot-Watt Univ., Riccarton, Edinburgh EH14 4AS, Scotland, fairouz@cee.hw.ac.uk
2 Mathematics and Computing Science, Eindhoven Univ. of Technology, P.O.Box 513, 5600 MB Eindhoven, the Netherlands, {c.j.bloo, r.p.nederpelt}@tue.nl

Abstract. We define an equivalence relation on $\lambda$-terms called shuffle-equivalence which attempts to capture the notion of reductional equivalence on strongly normalizing terms. The aim of reductional equivalence is to characterize the evaluation behavior of programs. The shuffle-equivalence classes are shown to divide the classes of $\beta$-equal strongly normalising terms (programs which lead to the same final value/output) into smaller ones consisting of terms with similar evaluation behavior. We refine $\beta$-reduction from a relation on terms to a relation on shuffle-equivalence classes, called shuffle-reduction, and show that this refinement captures existing generalisations of $\beta$-reduction. Shuffle-reduction allows one to make more redexes visible and to contract these newly visible redexes. Moreover, it allows more freedom in choosing the reduction path of a term, which can result in smaller terms along the reduction path if a clever reduction strategy is used. This can benefit both programming languages and theorem provers since this flexibility and freedom in choosing reduction paths can be exploited to produce the shortest program evaluation paths and optimal proofs.

Keywords: specification, semantics and rewriting of programs

1 Introduction

The $\lambda$-calculus plays a major role in the semantics of programming languages through its mechanisms for modeling evaluation strategies (e.g., call by name, call by value, etc.). Due to this basic role, the $\lambda$-calculus must be informative not only of the final value of the program (the normal form of the $\lambda$-term representing the program), but also of the consecutive values before the final value is reached. In particular, if we have two programs $P_1$ and $P_2$ that return the same final value, we want to know if these programs have equivalent evaluation paths in the sense that each evaluation path from $P_1$ to the final value (going through all the intermediate programs), corresponds (in a strong evaluation sense) to an evaluation path from $P_2$ to the final value, and vice versa. This will mean that $P_1$ and $P_2$ are equivalent programs even though they are written differently. Each intermediate value $a_1$ along the evaluation path from one of these programs to the final value corresponds to a unique intermediate value $a_2$ along the evaluation path of the other program to the final value, and the number of evaluation steps to reach $a_1$ from the first program is equal to the number of evaluation steps to reach $a_2$ from the second program. Of course this does not constitute a formal definition of what we call reductional equivalence. Reductional equivalence is difficult to define and is also undecidable. In this paper, we attempt to formally define a decidable approximation of reductional equivalence which we call shuffle equivalence.

For this, we attempt to observe the reduction behavior of $\lambda$-terms (which represent programs). $\beta$-equality of two terms $A$ and $B$ is — by the Church-Rosser property — equivalent to the existence of a common reduct $C$. Nothing can be said about the nature of the two reduction paths $A \rightarrow_\beta C$ and $B \rightarrow_\beta C$. It can be that both paths consist of the same number of steps, or that one of them is larger than the other. Also, the reduction behavior of $A$ and $B$ can be very different, as is the case if $A \equiv \text{KIt}$.

* We are grateful for enlightening discussions and useful feedback and comments received from Henk Barendregt, Twan Laan and Joe Wells.
and $B \equiv KII$, where $K$ and $I$ are the well known combinators with $Kxy \rightarrow x$ and $Ix \rightarrow x$, and $\Omega \equiv (\alpha x x)(\alpha x x)$. Both $A$ and $B$ have the common reduct $I$ but although $B$ is strongly normalizing (i.e., it allows no infinite reduction path), $A$ is not ($\Omega$ infinitely reduces to itself). Our problem is to characterize terms with equivalent reduction behavior in the sense that reduction paths from one can be mapped to reduction paths from the other. This is different from studying the behavior of programs using bisimulation. We are concerned with the syntactic analysis of programs written as $\lambda$-terms.

In order to conduct our syntactic study of reductional equivalence of programs written as $\lambda$-terms, we need to use a $\lambda$-notation that enables us to detect more redexes in a term than can be visible in the known classical notation $\lambda$-calculus [3]. For this purpose, we will use the item notation whose advantages are discussed in [7]; [6] explains that it is not feasible to syntactically describe generalised reduction in classical notation and therefore item notation is indispensable for our study of reductional equivalence which depends on extended and generalised redexes.

In Section 2 we introduce informally reductional equivalence, potential future redexes and the motivation for the preferred reductionally equivalent term $TS(A)$ of a $\lambda$-term $A$. $TS(A)$ makes visible as many redexes of $A$ as possible.

In Section 3 we introduce what is needed of the item notation and other formal machinery that will enable us to syntactically describe potential future redexes and will help us approximate reductional equivalence.

In Section 4 we introduce shuffle equivalence which we will use to approximate reductional equivalence. An important phenomenon that results from this definition is the ability to partition terms elegantly into parts which are informative as to where redexes occur.

In Section 5 we extend the usual $\beta$-reduction $\rightarrow_{\beta}$ on $\lambda$-terms to $\rightarrow_{\beta}^\gamma$ on classes of terms modulo shuffle equivalence. We establish that our extended reduction $\rightarrow_{\beta}^\gamma$ is a generalisation of $\rightarrow_{\beta}$ which enables one to have more freedom in choosing the reduction path of a term. In fact, a clever reduction algorithm might know how to choose the shortest reduction path and it is likely that this reduction path does not exist when usual reduction is used. We also establish that shuffle equivalence is a decidable approximation of reductional equivalence.

In Section 6 we compare our work with existing literature on generalising reduction and conclude that our notion of reduction in this paper subsumes all of this reported previous work.

2 Informal introduction to shuffle and reductional equivalence

2.1 Making as many redexes as possible visible

Example 1 Consider the terms: $A \equiv (\lambda_2 \lambda y. \lambda f.fy)\alpha x$ and $B \equiv (\lambda_2. (\lambda y. \lambda f.fy)x)\alpha$. Both terms have the term $\lambda f.fx$ as a reduct, so $A \equiv B$. However, $B$ has two redexes whereas $A$ has only one. Yet, our paper will establish that the terms $A$ and $B$ are reductionally equivalent. We will do so by extending the notion of redexes and by considering for each $E$, the preferred reductionally equivalent term $TS(E)$ which makes visible as many redexes of $E$ as possible. First, note that initially, one will only see the following redexes of $B$:

- $r_1 = (\lambda_2. (\lambda y. \lambda f.fy)x)\alpha$. Observe that $B \overset{r_1}{\rightarrow} (\lambda y. \lambda f.fy)x$.
- $r_2 = (\lambda y. \lambda f.fy)\alpha$. Observe that $B \overset{r_2}{\rightarrow} (\lambda_2. \lambda f.fx)\alpha$.

In $A$, the only obvious redex is: $r'_1 = (\lambda_2. \lambda y.fy)\alpha$. Observe that $A \overset{r'_1}{\rightarrow} (\lambda y. \lambda f.fy)x$.

Note that $r_1$ in $B$ and $r'_1$ in $A$ are both based on the redex $(\lambda_2. \rightarrow)\alpha$ and contracting $r_1$ in $B$ results in the same term as contracting $r'_1$ in $A$.

A closer look at $A$ enables us to see that in $A$ (as in $B$), $\lambda y$ will get matched with $x$ resulting in a redex $r_2' = (\lambda_2. \rightarrow)x$. There are differences however between $r_2$ in $B$ and $r_2'$ in $A$. $r_2$ in $B$ is completely visible and may be contracted before $r_1$ in $B$. $r_2'$ on the other hand is a future redex in $A$. In fact, it is not a redex of $A$ itself but a redex of a contractum of $A$, namely $(\lambda y. \lambda f.fy)x$, the result of contracting the redex $r'_1$ in $A$.
Reductional behavior for $\lambda$-terms

We could guess from $A$ itself the presence of the future redex. That is, looking at $A$ itself, we see that $A\beta$ is matched with $\alpha$ and $A\gamma$ is matched with $x$.

This has been noted by many researchers and hence rules like $(\lambda x.N)PQ \rightarrow_\beta (\lambda x.NQ)P$ have been introduced in many articles with different purposes [1,5,8,9,11,12,14,16,17,19-21,23]. Such rules enable one to rewrite $A$ so that both redexes become visible in $A$. Note that: $A \equiv (\lambda y.\lambda y.(\lambda y.fy)\alpha) \rightarrow_\beta (\lambda y.(\lambda y.fy)x)\alpha \equiv B$.

These transformations are rather powerful in that they can group together terms with equal reductional behavior. Let us give here this example:

**Example 2** Consider $E_1, E_2, E_3, E_4$ as follows:

- $E_1 \equiv ((\lambda f.(\lambda y.fxy)+m)n)\alpha\beta$
- $E_2 \equiv ((\lambda f.(\lambda y.fxy)m)+n)\alpha\beta$
- $E_3 \equiv (\lambda f.((\lambda y.fxy)m)n)\alpha\beta$
- $E_4 \equiv (\lambda f.(\lambda y.fxy)m)n\alpha\beta$

Note that $E_1 =_\beta E_2 =_\beta E_3 =_\beta E_4$. The visible redexes in each of these terms are as follows:

- In $E_1$: $(\lambda f.\lambda y.fxy)+m$ and $(\lambda y.fxy)m$.
- In $E_2$: $(\lambda f.\lambda y.fxy)m+n$ and $(\lambda y.fxy)m$.
- In $E_3$: $(\lambda f.\lambda y.fxy)m+n$ and $(\lambda y.fxy)m$.
- In $E_4$: $(\lambda f.\lambda y.fxy)m+n$ and $(\lambda y.fxy)m$.

Moreover, one can see potential future redexes as follows:

- In $E_1$: $\lambda y.-$ will eventually be applied to $m$ and $\lambda x.-$ will be eventually be applied to $n$.
- In $E_2$: $\lambda y.-$ will eventually be applied to $n$.
- In $E_3$: $\lambda y.-$ will eventually be applied to $n$.
- In $E_4$: $\lambda y.-$ will eventually be applied to $m$.

Note that $E_1 \rightarrow_\beta E_2 \rightarrow_\beta E_3 \rightarrow_\beta E_4$ and that by transforming $E_1$ to $E_2$ (resp. $E_3$ to $E_4$), an extra redex becomes visible. In $E_4$ all redexes are visible and $E_4$ is in $\theta$-normal form.

Based on this, one wonders if one could classify terms according to their transformational relationship as in $E_1, E_2, E_3, E_4$. One has to be careful however as the example below shows that we need to base reduction on classes.

**Example 3** If $A_1 \equiv ((\lambda x.(\lambda x.\lambda y.(\lambda y.(\lambda y.E)D)C)B)A$ and $A_2 \equiv ((\lambda x.\lambda y.((\lambda y.E)A)D)B)C)$. Then, $A_1 \not\rightarrow_\theta A_2$ and $A_2 \not\rightarrow_\theta A_1$. But, these two terms have the same $\theta$-normal form and are reductionally equivalent.

We will show in Section 4 that classes will be able to represent this.

In this paper, we propose to define for each term $M$, a term $TS(M)$ (the $\theta$-normal form of $M$) that makes as many redexes as possible visible. We consider the equivalence class of a term $M$ to be $\{M' \mid TS(M') \equiv TS(M)\}$. As $\rightarrow_\theta$ is Church Rosser and strongly normalizing, then if $M_1 =_\theta M_2$ we get $TS(M_1) \equiv TS(M_2)$. We set out to show that the notion of equivalence classes modulo $TS$ helps us to capture reductional equivalence on terms that are strongly normalizing. For Examples 1 and 2, $TS$ will help us establish that $A$ and $B$ are reductionally equivalent and that $E_1, \ldots, E_4$ are all reductionally equivalent.

### 2.2 Reductional Equivalence

In order to discuss reductional equivalence between terms, redexes will be extended (cf. Definition 24) so that a potential future redex like $(\lambda y.-)x$ in $A$ of Example 1 will be treated as a first class redex and will be contracted in $A$ even before the originator $(\lambda y.\lambda y.(\lambda y.fy)\alpha)$ has been contracted. Hence, with our extended notion of redexes and reduction we get in $A$ another redex:

$r'_2 = (\lambda y.(\lambda y.fy)x$, which when contracted in $A$ results in $(\lambda y.\lambda y.fy)\alpha)$.

Note that $r'_2$ is $\lambda y$ matched with $x$ (exactly as $r_2$ in $B$). Note moreover that contracting $r'_2$ in $A$ gives the same result as contracting $r_2$ in $B$.

With this notion of extended redex, we observe that there is a bijective correspondence between the (extended) redexes of $A$ and $B$ of Example 1. That is, $r_1$ corresponds to $r'_1$ and $r_2$ corresponds to $r'_2$. Moreover, if one redex is contracted in $A$, the reduct is syntactically equal to the reduct which results from contracting the corresponding redex in $B$ and vice versa. That is, $r_1$ and $r'_1$ yield the same values; similarly $r_2$ and $r'_2$ yield the same values. This is illustrated by this example:
Example 4: The reduction paths from A and B of Example 1 are as follows:

A-Path 1: \((\lambda_2, \lambda_y, \lambda_f.fy)x \xrightarrow{r_1} (\lambda_y, \lambda_f.fy)x \xrightarrow{tr} (\lambda_f.fy)x \xrightarrow{tr} \lambda_f.fx\)

A-Path 2: \((\lambda_2, \lambda_y, \lambda_f.fy)x \xrightarrow{r_2} (\lambda_2, \lambda_f.fy)x \xrightarrow{tr} \lambda_f.fx\)

B-Path 1: \((\lambda_2, \lambda_y, \lambda_f.fy)x \xrightarrow{r_1} (\lambda_y, \lambda_f.fy)x \xrightarrow{tr} \lambda_f.fx\)

B-Path 2: \((\lambda_2, \lambda_y, \lambda_f.fy)x \xrightarrow{r_2} (\lambda_2, \lambda_f.fy)x \xrightarrow{tr} \lambda_f.fx\)

It is clear that A and B have the same number of possible paths before reaching the normal form and that there is a bijective correspondence between the paths A-Path 1 and B-Path 1, and between A-Path 2 and B-Path 2.

These considerations lead us to define reductional equivalence \(\sim_{\text{equiv}}\) by:

**Definition 5**: We say that A and B are reductionally equivalent and write \(A \sim_{\text{equiv}} B\) iff

1. If A or B is in normal form then A is syntactically equal to B.
2. There is a bijective correspondence between the (extended) redexes of A and B.
3. Contracting an (extended) redex in A results in a value syntactically equal (\(\equiv\)) or reductionally equivalent (\(\sim_{\text{equiv}}\)) to the result of contracting the corresponding redex in B and vice versa.

Example 6: \(A \sim_{\text{equiv}} B\) for A, B as in Example 1. Also \(E_1 \sim_{\text{equiv}} E_2 \sim_{\text{equiv}} E_3 \sim_{\text{equiv}} E_4\) for \(E_1, E_2, E_3, E_4\) as in Example 2. But, due to clause 2 of the above definition, it is not the case that \(KII \sim_{\text{equiv}} K\Omega\).

Remark 7: Note that the above clauses cannot handle the following two situations:

- Forbidding the reductional equivalence of terms such as: \((\lambda_2, I)I\) and \((\lambda_2, I)K\).
- Ensuring the compositionality of reductional equivalence in the sense that if \(A_1 \sim_{\text{equiv}} A_2\) then \(A_1B \sim_{\text{equiv}} A_2B\) and \(\lambda_2.A_1 \sim_{\text{equiv}} \lambda_2.A_2\). For example, if \(A_1 \equiv \lambda_2.(\lambda_2.y)z\) and \(A_2 \equiv \lambda_2.(\lambda_2.y)I\), then \(A_1 \sim_{\text{equiv}} A_2\) but \(A_1(\Pi) \neq_{\text{equiv}} A_2(\Pi)\).

In order to handle the above situations, we need to add the following fourth clause to Definition 5:

4. Arguments of corresponding (extended) redexes are syntactically equal or reductionally equivalent.

Clause 4 above solves the problems raised by the above two situations. We will not however be concerned with these situations in this paper and we will not hence include clause 4 in Definition 5.

We conjecture that in general it is undecidable whether two terms are reductionally equivalent.

**Conjecture 8**: It is undecidable whether two terms are reductionally equivalent.

2.3 Shuffle-equivalence, shuffle-reduction and summary of results

We settle in this paper for a new notion that we call *shuffle-equivalence*. Shuffle-equivalence is particularly related to \(\theta\) and to the notion of term-reshuffling of [8]. First, the term-reshuffling of [8] is the \(\theta\)-normal form. Moreover, shuffle-equivalence equates terms modulo term-reshuffling. Hence, if \(A \rightarrow_{\theta} B\) or \(A\) term-reshuffles to \(B\) then \(A\) and \(B\) are shuffle-equivalent. There are many cases however where \(A\) and \(B\) are shuffle-equivalent without \(A\) and \(B\) being \(\theta\)-related since shuffle-equivalence looks for the class of all terms that have the same term-reshuffling.

In order to give an intuition why we take classes modulo term-reshuffling, observe that extended redexes can be shuffled to ”classical” (i.e., non extended) redexes without losing reductional equivalence. This can be seen by our terms \(A\) and \(B\) of Example 1. The extended redex \(r_2\) in \(A\) becomes classical in \(B\). We call \(B\) the reshuffled version of \(A\). We have seen that \(A \sim_{\text{equiv}} B\).

Now to decide on the shuffle-equivalence of two terms \(A\) and \(B\), we reshuffle both \(A\) and \(B\) and if we get in both cases the same result, then we say that \(A\) and \(B\) are shuffle-equivalent. We denote the reshuffled version of a term \(A\) by \(TS(A)\); \(TS\) will be defined in Section 4.

It will be easier to understand what the operation \(TS\) does if we change the classical notation we have been using so far. So we depart from those researchers who use \(\theta\), by using an extended form of \(\theta\) based on the term-reshuffling of [8] (which turns out to be the \(\theta\)-normal form). Furthermore, we depart from [8] by working with the equivalence classes modulo term-reshuffling rather than the terms.
normalizing terms, two shuffle-equivalent terms are reductionally equivalent, that shuffle-equivalence is term reshuffling, etc., to apply to classes rather than terms. As classes capture existing extensions of reductions such as \( T \) sense the same redexes. We believe this is the closest decidable and does not coincide with reductional equivalence (which we conjecture to be undecidable).

Certainly, our notion of shuffle-equivalence captures existing extensions of reductions. For example, if \( A \rightarrow_\beta B \) then \( A \) and \( B \) are shuffle-equivalent. The converse is not true (cf. Example 23).

Once we have an approximation to reductional equivalence, we will extend the notion of \( \beta \)-reduction to apply to classes rather than terms. As classes capture existing extensions of reductions such as \( \theta \), term reshuffling, etc., \( \beta \)-reduction over classes will capture all these notions. We say \( A \) shuffle-reduces to \( A' \) and we write \( A \leftrightarrow_\beta A' \) iff \( \exists B \in [A] \exists B' \in [A'] \) such that \( B \rightarrow_\beta B' \). We show (cf. Lemma 26) that both \( \rightarrow_\beta \) and the generalised reduction \( \leftrightarrow_\beta \) of [4] are captured by \( \leftrightarrow_\beta \).

3 The formal machinery

The classical notation cannot extend the notion of redexes or enable reshuffling in an easy way. Item notation however can ([7] discusses various advantages of this notation). Let \( V \) be an infinite collection of variables over which range. In item notation, terms of the \( \lambda \)-calculus are: 

\[
T := V \cup \{(\tau \delta)T | (\lambda V)T\}. 
\]

We take \( A, B, C, \ldots \) to range over \( T \). We call \( (A\delta) \) a \( \delta \)-item, \( A \) the body of the item and \( (A\delta)B \) means apply \( B \) to \( A \) (note the order). \( (\lambda x) \) is called a \( \lambda \)-item. A \( \delta \)-redex starts with a \( \delta \)-item (i.e., \( (A\delta) \)) next to a \( \lambda \)-item (i.e., \( (\lambda x) \)).

Example 9 It is easy to see that \( A \equiv (u\delta)(w\delta)(z\delta)(\lambda x)(\lambda y)(z\delta)(y\delta)x \) in item notation. By moving the item \( u\delta \) to the right until it is next to its 'matching partner' \( (\lambda x) \), this \( A \) reshuffles to \( TS(A) \equiv (u\delta)(w\delta)(z\delta)(\lambda x)(\lambda y)(u\delta)(z\delta)(y\delta)x \).

Such a reshuffling in item notation is clearer than reshuffling in classical notation where the term \((\lambda x)(\lambda y)(\lambda x).xyz)w \) is reshuffled to \((\lambda x)(\lambda y)(\lambda x.\lambda x.xyz)n)w \).

Note furthermore that the reshuffling is not problematic because we use the Barendregt Convention (see below) which means that no free variable will become unnecessarily bound after reshuffling due to the fact that names of bound and free variables are distinct.

Example 10 \( E_1 \) of Example 2 reads \( (n\delta)(m\delta)(+\delta)(\lambda f)(\lambda y)(\delta x)(y\delta)f \) in item notation. Here, the (classical) redex corresponds to a \( \beta \lambda \)-pair followed by the body of the abstraction, as follows:

\[
(\lambda f)(\lambda x.\lambda y.fxy)m \leftrightarrow (\lambda f)(\lambda x.\lambda y)(\lambda y)(\delta x)(y\delta)f.
\]

Note that the \( \delta \)-item \( +\delta \) and the \( \lambda \)-item \( (\lambda f) \) are now adjacent, which is characteristic for the presence of a classical redex in item notation. (Cf. Figure 1 which represents \( E_1 \). The second and third redexes of \( E_1 \) are obtained by matching \( \delta \) and \( \lambda \)-items which are not adjacent:

- \( (\lambda y.fxy)n \) is visible as it corresponds to the matching \( (n\delta)(\lambda y) \) where \( (n\delta) \) and \( (\lambda y) \) are separated by the segment \( (m\delta)(+\delta)(\lambda f)(\lambda y)(\lambda y)(\delta x)(y\delta)f \) which has the bracketing structure [[ ]] .
- \( (\lambda z.\lambda y.fxy)m \) is visible as it corresponds to the matching \( (m\delta)(\lambda z) \) where \( (m\delta) \) and \( (\lambda z) \) are separated by the segment \( (+\delta)(\lambda f) \).

As above, we will use obvious notions throughout like partner, match, bachelor, etc. In Figure 2, \( (+\delta) \) and \( (\lambda f) \) match or are partnered. So are the items \( (n\delta) \) and \( (\lambda y) \), \( (y\delta) \) and \( (\delta x) \) on the other hand are bachelor. The adjacent item pair \( (+\delta)(\lambda f) \) is called a \( \delta \lambda \)-pair and the non-adjacent items \( (n\delta)(\lambda y) \) form a \( \lambda \lambda \)-couple.

Term reshuffling amounts to moving \( \delta \)-items to occur next to their matching \( \lambda \)-items. Hence \( E_1 \) of Example 2 is reshuffled to \( (+\delta)(\lambda f)(m\delta)(\lambda x)(n\delta)(\lambda y)(y\delta)(\delta x)f \) and Figure 1 changes to Figure 4 (which represents \( E_4 \)). Furthermore, Figures 2 and 3 (which represent \( E_2 \) and \( E_3 \)) also change to Figure 4.

According to our shuffle-equivalence, \( E_1, E_2, E_3 \) and \( E_4 \) belong to the same class and are \( \sim_{\text{equi}} \).
In item notation, each term \( A \) is the concatenation of zero or more items and a variable: \( A \equiv s_1s_2 \cdots s_nx \) where each \( s_i \) is either a \( \lambda \)-item or a \( \delta \)-item, and \( x \in V \). These items \( s_1, s_2, \ldots, s_n \) are called the main items of \( A \), \( x \) is called the heart of \( A \), notation \( \text{\textcopyright}(A) \).\(^1\) We use \( s, s_1, s_2, \ldots \) to range over items. A concatenation of zero or more items \( s_1s_2 \cdots s_n \) is called a segment. We use \( \overline{s}, \overline{s}_1, \overline{s}_2, \ldots \) as meta-variables for segments. We write \( \emptyset \) for the empty segment. The items \( s_1, s_2, \ldots, s_n \) (if any) are called the main items of the segment. A concatenation of adjacent main items \( s_m \cdots s_{m+k} \), is called a main segment. A \( \delta \lambda \)-segment is a \( \delta \)-item immediately followed by a \( \lambda \)-item.

The weight of a segment \( \overline{s} \), \( \text{weight}(\overline{s}) \), is the number of main items that compose the segment. Moreover, we define \( \text{weight}(\overline{sx}) = \text{weight}(\overline{s}) \) for \( x \in V \).

In reduction, the matching of the \( \delta \) and the \( \lambda \) in question is the important thing. Well-balanced segments (w-b) separate matching \( \delta \) and \( \lambda \)-items. Well-balanced segments are given inductively by: (i) \( \emptyset \) is w-b, (ii) if \( \overline{s} \) is w-b then \( (A\delta)\overline{s}(\lambda x) \) is w-b, (iii) if \( \overline{s}_1, \overline{s}_2, \ldots, \overline{s}_n \) are w-b, then the concatenation \( \overline{s}_1\overline{s}_2, \ldots, \overline{s}_n \) is w-b.

In Figures 1, 2, 3, and 4, all segments that occur under a hat are w-b.

**Lemma 11** Every term has one of the following three forms: (i) \( (A_1\delta) \cdots (A_n\delta)x \), where \( x \in V \) and \( n \geq 0 \), (ii) \( (\lambda x)B \), and (iii) \( (A_1\delta) \cdots (A_n\delta)(B\delta)(\lambda x)D \), where \( n \geq 0 \).

**Definition 12** We say that two terms \( A \) and \( B \) are semantically equivalent iff \( A =_\beta B \).

Bound and free variables and substitution are defined as usual. We write \( BV(A) \) and \( FV(A) \) to represent the bound and free variables of \( A \) respectively. Note that in item notation, the scope of a \( \lambda \)-item is anything to the right of it. We write \( A[x := B] \) to denote the term where all the free occurrences of \( x \) in \( A \) have been replaced by \( B \). We take terms to be equivalent up to variable renaming and use \( \equiv \) to denote syntactical equality of terms. We assume the usual Barendregt variable convention \( BC \) and definition of compatibility (cf. [3]). We say that \( A \) is strongly normalizing with respect to a reduction relation \( \rightarrow \) (written \( \text{SN}_\rightarrow(A) \)) iff every \( \rightarrow \)-reduction path starting at \( A \) terminates.

## 4 Shuffle-equivalence

In this section we follow [8] and rewrite terms so that all the newly visible redexes can be subject to \( \rightarrow_\beta \). We shall show that this term rewriting function is correct in the sense that \( A =_\beta T\text{S}(A) \), i.e., \( A \)

\(^1\) Note that the term head variable used in [2] is a special case of our notion of heart. The head variable of a term in head normal form is the heart of the term. It is not the case however that the heart of a term is always a head variable.
Reductional behavior for $\lambda$-terms

and $\text{TS}(A)$ are semantically equivalent (cf. Lemma 18). Furthermore, we show that shuffle-equivalence is stronger than $\rightarrow_e$ and than the term-reshuffling of [8] (cf. Lemma 22 and Example 23) and that shuffle-equivalence classes are decidable (cf. Lemma 21). In Section 5 (cf. Lemma 40), we show that shuffle-equivalence is indeed a (decidable) approximation of reductional equivalence.

**Definition 13** The reshuffling function $\text{TS}$ is defined such that:

\[
\text{TS}((\lambda_x)C) = d_f (\lambda_x)\text{TS}(C), \\
\text{TS}((B_1\delta)\cdots(B_n\delta)x) = d_f (\text{TS}(B_1)\delta)\cdots(\text{TS}(B_n)\delta)x \quad \text{if} \ x \in V, \\
\text{TS}((B_1\delta)\cdots(B_n\delta)(C\delta)(\lambda_x)E) = d_f (\text{TS}(C)\delta)(\lambda_x)\text{TS}((B_1\delta)\cdots(B_n\delta)E).
\]

Note that the second and third clauses also apply for $n = 0$.

Note that in this definition we use the notions of the previous section: if a term starts with a $\lambda_x$-item then this item is bachelor. If a term starts with a partnered $\delta$-item then the last clause above applies. If a term starts with a bachelor $\delta$-item then the term starts with a positive number of bachelor $\delta$-items followed by either a variable in which case the second clause applies or a well-balanced segment in which case the last clause applies. With term reshuffling, well-balanced segments are rewritten so that $\delta\lambda$-couples become $\delta\lambda$-pair and all bachelor main $\delta$-items are moved to the right of all well-balanced main segments.


It can be seen that for any $A$, $\text{TS}(A)$ is of the form $\overline{S_0}S_1x$ where $x \in V$, $S_1$ consists of all bachelor main $\delta$-items of $A$ and $\overline{S_0}$ is of the form $\overline{S_2}S_3\cdots\overline{S_n}$ where $S_i$ is either a $\delta\lambda$-pair or a bachelor main $\lambda$-item. Now, the following lemmas show some properties of $\text{TS}$.

**Lemma 15 (Decidability of TS)** For any $A, B$, it is decidable whether $\text{TS}(A) \equiv \text{TS}(B)$.

**Proof:** This is obvious as $\equiv$ is decidable.

**Lemma 16**

1. For all terms $M$, $\text{TS}(M)$ is well defined.
2. $\text{FV}(M) = \text{FV}(\text{TS}(M))$
3. If $\overline{s}$ is well-balanced, then $\text{TS}((A_1\delta)\cdots(A_n\delta)\overline{s}B) \equiv \text{TS}(\overline{s}(A_1\delta)\cdots(A_n\delta)B)$.

**Proof:** 1. Every time at most one case of the definition of $\text{TS}(M)$ is applicable, and weights of the resulting terms to which $\text{TS}$ is applied become smaller or $\text{TS}$ disappears. 2. Induction on the structure of $M$. 3. By induction on weight($\overline{s}$).

**Lemma 17** For a term $A$, $\text{TS}(A) \equiv \overline{S_0}S_1\overline{s}\overline{\lambda}(A)$, where $\overline{s}$ consists of the term reshufflings of all bachelor main $\delta$-items of $A$ and $\overline{S_0}$ is a sequence of term reshufflings of main $\delta\lambda$-segments and bachelor main $\lambda$-items.

**Proof:** By induction on weight($A$).

- Case $A \equiv (\lambda_x)C$, use IH on $C$. Case $A \equiv (B_1\delta)\cdots(B_n\delta)x$, $x \in V$, then $\overline{S_0}$ is empty.
- $A \equiv (B_1\delta)\cdots(B_n\delta)(C\delta)(\lambda_x)E$. Then $\text{TS}(A) \equiv (\text{TS}(C)\delta)(\lambda_x)\text{TS}((B_1\delta)\cdots(B_n\delta)E)$. By the induction hypothesis $\text{TS}((B_1\delta)\cdots(B_n\delta)E)$ is of the form $\overline{S_0}S_1\overline{s}\overline{\lambda}(E) \equiv \overline{S_0}S_1\overline{s}\overline{\lambda}(A)$.
Lemma 18 For all terms $A, B$ and variable $x$:
1. $TS(A) = TS(TS(A))$,
2. $TS(A[x := B]) = TS(TS(A)[x := TS(B)])$, and
3. $A =_\beta TS(A)$.

Proof:
2. Induction on the number of symbols in $A$, using 1.
3. By induction on the number of symbols in $A$.

If $A \equiv (A_1\delta) \cdots (A_n\delta)x$ where $x \in V$ or $A \equiv (\lambda_x)A_2$ then use the induction hypothesis.

If $A \equiv (A_1\delta) \cdots (A_n\delta)(B\delta)(\lambda_x)A_2$ then $TS(A) \equiv (TS(B)\delta)(\lambda_x)TS((A_1\delta) \cdots (A_n\delta)D) \equiv_\beta$
\[(B\delta)(\lambda_x)(A_1\delta) \cdots (A_n\delta)D =_\beta ((A_1\delta) \cdots (A_n\delta)D)[x := B] \equiv_\beta (A_1\delta) \cdots (A_n\delta)(B\delta)(\lambda_x)D \]
\[\square\]

Corollary 19 For all terms $A, B$, $TS(A) =_\beta TS(B)$ iff $A =_\beta B$.

Definition (Shuffle-class and shuffle-equivalence) For a term $A$, we define $[A]$, the shuffle class of $A$, to be $\{B \mid TS(A) \equiv TS(B)\}$. We say that $A$ and $B$ are shuffle-equivalent iff $[A] = [B]$.

Lemma 21 For any $A, B$, it is decidable whether $A \in [B]$. Moreover, if $A \in [B]$ then $A =_\beta B$.

Proof: Follows from Lemma 18 and Corollary 19. \[\square\]

The following shows that shuffle-equivalence contains $\to_\theta$ and the term-reshuffling of [8].

Lemma 22 If $A \to_\theta B$ or $A$ term-reshuffles to $B$ in the sense of [8] then $A$ and $B$ are shuffle-equivalent.

Proof: $TS$ formalizes term-reshuffling of [8] and the latter captures $\to_\theta$. \[\square\]

The other way round does not always hold however:

Example 23 Let $A \equiv (A_1\delta)(A_2\delta)(A_3\delta)(\lambda_x)(\lambda_y)A_4$ and $B \equiv (A_2\delta)(A_3\delta)(\lambda_x)(\lambda_y)(A_1\delta)(\lambda_x)A_4$. $A$ and $B$ are shuffle-equivalent but are not related by $\to_\theta$ (there exists $C \not\in A$ however such that $B \to_\theta C$ and $A \to_\theta C$). Moreover, neither $A \equiv TS(B)$ nor $B \equiv TS(A)$, but $TS(A) \equiv TS(B)$.

5 Shuffle-reduction

In this section, we introduce shuffle-reduction $\leadsto_\beta$, show that it is Church-Rosser and that shuffle-equivalence preserves reduction in the sense that if $A \to_\beta B$ then $A \leadsto_\beta B$. We show that shuffle-equivalence implies reductional equivalence on $SN$ terms (Definition 5) and that shuffle-reduction on classes makes more redexes visible and allows for smaller terms during reductions.

Definition 24 (Shuffle-reduction, extended redexes and $\to_\beta$) – One-step shuffle-reduction $\leadsto_\beta$ is the least compatible relation generated by:

$A \leadsto_\beta B$ iff $\exists A' \in [A] \exists B' \in [B] (A' \to_\theta B')$.

Many-step shuffle-reduction $\leadsto_\beta$ is the reflexive and transitive closure of $\leadsto_\beta$ and $\equiv_\beta$ is the least equivalence relation generated by $\leadsto_\beta$.

- An extended redex starts with the $\delta$-stem of a $\delta$-$\lambda$-couple (i.e. is of the form $(C\delta)\overline{s}(\lambda_x)A$ where $\overline{s}$ is well-balanced).
- $\leadsto_\beta$ is the least compatible relation generated by $(B_1\delta)\overline{s}(\lambda_x)B_2 \leadsto_\beta \overline{s}(B_2[x := B_1])$ for $\overline{s}$ well-balanced, that is, $\leadsto_\beta$-reduction contracts an (extended) redex. $\to_\beta$ is the reflexive and transitive closure of $\leadsto_\beta$ and $\equiv_\beta$ the least equivalence relation closed under $\to_\beta$.

Example 25 Let $A \equiv (z\delta)(w\delta)(\lambda_x)(\lambda_y)y$. Then $[A] = \{A, (w\delta)(\lambda_x)(z\delta)(\lambda_y)y\}$. Moreover, $A \leadsto_\beta (w\delta)(\lambda_x)y$ and $A \equiv_\beta (z\delta)(\lambda_y)y$.

$\equiv_\beta$ has been used in [4, 8] where it was shown to be more general than other generalised notions of reduction introduced in the literature (such as $(g)$ of Section 6). The following lemma shows that $\leadsto_\beta$ is more general than $\to_\beta$ (the generalised reduction of [4, 8]) and that it captures classical $\beta$-reduction.
Lemma 26 $\rightarrow_\beta \subseteq \Leftarrow_\beta \subseteq \leftarrow_\beta$.

Proof: It suffices to show $(A\delta)(\lambda x)C \leftarrow_\beta C[x := A]$ and $(A\delta)(\lambda x)C \rightarrow_\beta \exists C[x := A]$. $(A\delta)(\lambda x)C \equiv (A\delta)(\lambda x)C \rightarrow_\beta \emptyset C[x := A]$. Also, by Lemma 16, we know that $(A\delta)(\lambda x)C \in \bar{\delta}(A\delta)(\lambda x)C$, and since $(A\delta)(\lambda x)C \rightarrow_\beta \exists C[x := A]$ we have $(A\delta)(\lambda x)C \rightarrow_\beta \exists C[x := A]$. It is easy to show that these inclusions are strict. For example, if $A \equiv (A\delta)(B\delta)(\lambda x)(C\delta)(D\delta)(\lambda y)(\lambda z)(A')B$ and $A \equiv (C\delta)(B\delta)(\lambda x)(D\delta)(\lambda y)(\lambda z)(A')B$ and $A \equiv (C\delta)(B\delta)(\lambda x)(D\delta)(\lambda y)(\lambda z)(E[t := A])$ (which have respectively the bracketing structures $[[[[]]]]$ and $[[[]]]$), then $A \rightarrow_\beta A_1$ but $A \neq A_1$. Similarly, $(A\delta)(B\delta)(\lambda x)C \rightarrow_\beta (B\delta)(\lambda z)C[y := A]$ but $(A\delta)(B\delta)(\lambda x)C \rightarrow_\beta (B\delta)(\lambda z)C[y := A]$. □

Corollary 27 $\rightarrow_\beta \subseteq \leftarrow_\beta \subseteq \leftarrow_\beta$.

Remark 28 It is not in general true that $A \leftarrow_\beta B \Rightarrow \exists A' \in [A] \exists B' \in [B][A' \rightarrow_\beta B']$. This can be seen by the following counterexample:

Let $A \equiv ((\lambda u)(\lambda u)v)(\lambda u)(\lambda u)v(x \rightarrow B)$. Then $A \rightarrow_\beta (\lambda u)(\lambda u)v \rightarrow_\beta \exists B$. But $[B]$ has three elements: $A$, $(w_0)(\lambda u)(\lambda u)(\lambda u)v(x \rightarrow B)$ and $(w_0)(\lambda u)(\lambda u)(\lambda u)v(x \rightarrow B)$. But if $A'$ of $[A]$ then the only $\beta$-reduct of $A'$ is $(w_0)(\lambda u)(\lambda u)v$, which doesn’t $\rightarrow_\beta$-reduce to $B$. In Lemma 36 however, we find a correspondence between $\leftarrow_\beta$ on classes and $\rightarrow_\beta$ on terms.

Lemma 29 $TS(A) \rightarrow_\beta B$ iff $TS(A) \rightarrow_\beta B$.

Proof: This is a direct consequence of Lemma 17. □

Lemma 30 If $A \rightarrow_\beta B$ then for all $A' \in [A]$, for all $B' \in [B]$, $A' \rightarrow_\beta B'$.

Proof: As $A \rightarrow_\beta B$ then $\exists A_1 \in [A] \exists B_1 \in [B][A_1 \rightarrow_\beta B_1]$. Let $A', B' \in [A], [B]$ respectively. Then $A_1 \in [A']$, $B_1 \in [B']$, $A_1 \rightarrow_\beta B_1$. So $A' \rightarrow_\beta B'$. □

Corollary 31 $A \rightarrow_\beta B$ iff $TS(A) \rightarrow_\beta TS(B)$.

Remark 32 Note that $A \rightarrow_\beta B \neq TS(A) \rightarrow_\beta TS(B)$ nor do we have $A \rightarrow_\beta B \Rightarrow TS(A) \rightarrow_\beta TS(B)$. Take for example $A$ and $B$ where $A \equiv (z\lambda u)(z\lambda u)v)(\lambda u)(\lambda u)v$ and $B \equiv (\gamma x)(\gamma x)(z\lambda u)(z\lambda u)v$. It is obvious that $A \rightarrow_\beta B$ (hence $A \rightarrow_\beta B$) yet $TS(A) \equiv A \neq B \rightarrow_\beta TS(B) \equiv (\gamma x)(\lambda u)(\gamma x)(z\lambda u)v$.

The following lemma helps establish that $\leftarrow_\beta$ is Church-Rosser:

Lemma 33 If $A \leftarrow_\beta B$ then $A \Rightarrow B$.

Proof: Say $A' \in [A], B' \in [B], A' \rightarrow_\beta B'$. Then by Lemma 18: $A =_\beta TS(A) \equiv TS(A') =_\beta A' =_\beta B' =_\beta TS(B') \equiv TS(B) =_\beta B$. □

Corollary 34

1. If $A \rightarrow_\beta B$ then $A \Rightarrow B$.
2. $A \Rightarrow B$ iff $A \rightarrow_\beta B$ iff $TS(A) =_\beta TS(B)$.

Theorem 35 (The general Church Rosser theorem for $\leftarrow_\beta$)

If $A \leftarrow_\beta B$ and $A \Leftarrow_\beta C$, then there exists $D$ such that $B \leftarrow_\beta D$ and $C \leftarrow_\beta D$.

Proof: As $A \leftarrow_\beta B$ and $A \Leftarrow_\beta C$, then by Corollary 34, $A =_\beta B$ and $A =_\beta C$. Hence, $B =_\beta C$ and by $CR$ for $\rightarrow_\beta$, there exists $D$ such that $B \rightarrow_\beta D$ and $C \rightarrow_\beta D$. But, $M \rightarrow_\beta N$ implies $M \Rightarrow_\beta N$. Hence we are done. □

As we noted in Remark 32, we can have $TS(C) \rightarrow_\beta D$ where $D \neq TS(D)$. Nevertheless, term reshuffling preserves $\beta$-reduction. This is a generalisation of the result in [8] to equivalence classes.

Lemma 36 If $A, B \in T$ and $A \rightarrow_\beta B$ then $(\exists B' \in [B])[TS(A) \rightarrow_\beta B']$. In other words, the following diagram commutes:

$A \rightarrow_\beta B$

$\Downarrow$

$TS(A) \rightarrow_\beta B' \in [B]$
Hence we have provided a relation between terms which approximates reductional equivalence. Here

**Lemma 38**

Moreover,

**Corollary 37**

Now we show that shuffle-equivalence preserves strong normalization:

**Lemma 38**

Proof: By induction on \( \sim_\beta \).

Now we show that shuffle-equivalence preserves \( \beta \)-strong normalization:

**Lemma 39**

Proof: As \( \sim_\gamma \) is immediate. \( \iff \) is by using a result of [50, 90] which states that the length of the longest reduction of a term is invariant by \( \sigma \)-equivalence and by noting that shuffle-reduction is isomorphic to \( \beta \)-reduction modulo \( \sigma \)-equivalence. Another way of showing \( \iff \) is by induction on \( \langle d(A), A \rangle \) ordered by the lexicographic product ordering where \( d(A) \) denotes the maximum length of a \( \beta \)-reduction of \( A \) to its \( \beta \)-normal form. Case \( d(A) = 0 \) is trivial. Case \( d(A) \neq 0 \), we use induction on the structure of \( A \) as given in Lemma 11. The interesting case when \( A \equiv \langle A_1 \delta \cdots A_n \delta \rangle \langle B \delta \rangle \langle \lambda \rangle \langle D \rangle \), for \( n \geq 0 \) can be done by noting that \( \langle A_1 \delta \cdots A_n \delta \rangle \langle B \delta \rangle \langle \lambda \rangle \langle D \rangle \) then for all \( A' \) that if \( A' \) is in normal form so \( \langle B \delta \rangle C' \) if the redex was inside \( B \), \( \delta \) or \( C \) or of the form \( \delta C' \) if the redex consisted of \( (B \delta) \) and its partnered item.

Then in the first case \( s \langle B \delta \rangle C \) can reduce to \( s \langle B \delta \rangle C' \) by contracting the corresponding redex, now by the induction hypothesis \( (B \delta) s \delta C' \) is reductionally equivalent to \( s' \langle B \delta \rangle C' \). In the second case, \( s \langle B \delta \rangle C \) also reduces to \( s C' \). Hence \( (B \delta) s \delta C \) is reductionally equivalent to \( s \langle B \delta \rangle C \).

Now we show that shuffle-equivalence for SN terms implies reducibility equivalence:

**Lemma 40**

Proof: It is sufficient to show that \( (B \delta) s \delta C \sim_{eq} \delta s \langle B \delta \rangle C \) if \( \delta \) is well-balanced and \( (B \delta) s \delta C \in SN_{\sim_{eq}} \).

We prove this by induction on the maximal length of \( \sim_{\beta} \)-reduction paths of \( (B \delta) s \delta C \).

If \( (B \delta) s \delta C \) is in normal form then \( \delta \equiv \emptyset \) so \( (B \delta) s \delta C \equiv s \langle B \delta \rangle C \).

If \( (B \delta) s \delta C \) is not in normal form then contraction of some redex yields a term which is either of the form \( (B \delta) s \delta C' \) (if the redex was inside \( B \), \( \delta \) or \( C \) or of the form \( \delta C' \) if the redex consisted of \( (B \delta) \) and its partnered item.

Hence we have provided a relation between terms which approximates reducibility equivalence. Here are some facts on this relation and on reducibility equivalence:

**Fact 41**

The following holds:

1. Let \( A \in SN_{\sim_{eq}} \). If \( TS(A) \equiv TS(B) \) then \( A \sim_{eq} B \) (Lemma 40).

2. \( TS(A) \equiv TS(B) \) does not necessarily imply \( A \sim_{eq} B \) (Example 42).

3. \( A \sim_{eq} B \) does not necessarily imply \( TS(A) \equiv TS(B) \) (Example 43 below).

4. \( TS(A) \equiv TS(B) \) is decidable (Lemma 15).

5. Let \( A \in SN_{\sim_{eq}} \). Then for all \( A' \in [A], A' \in SN_{\sim_{eq}} \).

**Example 42**

Let \( \langle \alpha \rangle \langle \beta \rangle \langle \lambda \rangle \langle \lambda \rangle (x)(z)(\bar{z}) \bar{z} (i.e., \langle \lambda \rangle \langle \lambda \rangle \Omega \langle \alpha \rangle \Omega b) \) and \( B \equiv \langle \beta \rangle \langle \lambda \rangle \langle \alpha \rangle (x)(z)(\bar{z}) \bar{z} \langle \lambda \rangle \langle \lambda \rangle (x)(z)(\bar{z}) \bar{z} (i.e., \langle \alpha \rangle \langle \lambda \rangle \Omega \langle \lambda \rangle \Omega \langle \alpha \rangle \Omega b) \) for \( \Omega \equiv \langle \lambda \rangle \langle \lambda \rangle \langle \lambda \rangle \langle \lambda \rangle \Omega \langle \lambda \rangle \langle \lambda \rangle \langle \lambda \rangle \langle \lambda \rangle \Omega \). Now, \( TS(A) \equiv TS(B) \) but \( A \not \sim_{eq} B \) since contracting \( \Omega \) will not result in syntactically equivalent terms. This shows in 1. of Fact 41 that one cannot drop the assumption that \( A \) is strongly normalizing.

**Example 43**

Let \( \equiv \langle \alpha \rangle (\lambda \langle \lambda \rangle (x)(z)(\bar{z}) \bar{z} \langle \lambda \rangle \langle \lambda \rangle \rangle y \) and \( B \equiv \langle \alpha \rangle (\lambda \langle \lambda \rangle (x)(z)(\bar{z}) \bar{z} \langle \lambda \rangle \langle \lambda \rangle \rangle y \). We have \( A \sim_{eq} B \) but \( TS(A) \not \equiv TS(B) \). The same holds for the terms \( \langle \alpha \rangle (\lambda \langle \lambda \rangle (y)(\bar{z}) \bar{z} \langle \lambda \rangle \langle \lambda \rangle \rangle y \) and \( \langle \alpha \rangle (\lambda \langle \lambda \rangle (y)(\bar{z}) \bar{z} \langle \lambda \rangle \langle \lambda \rangle \rangle y \). This shows that the converse of 1. in Fact 41 does not hold.
We finish by showing that due to the fact that shuffle-reduction on classes makes more redexes visible, it allows for smaller terms during reductions.

Example 44 Let \( M \equiv (\lambda_x.\lambda_y.y)(Cxx\cdots x)\) where \( B \) is a BIG term and \( u \) is in normal form, \( z \notin FV(u) \). Then \( M \rightarrow_B (\lambda_y.y)(CBB\cdots B))\lambda_z.u \rightarrow_B (\lambda_z.u)(CBB\cdots B) \rightarrow_B u \). Now the first and second reducts both contain \( CBB\cdots B \), so they are very long. Shuffle reduction allows us to reduce \( M \) in such a way that all the new terms are of smaller size than \( M \). This can be seen as follows: \( TS(M) \equiv (\lambda_x.\lambda_y.y)(Cxx\cdots x)\lambda_z.u \rightarrow_B (\lambda_z.u)(Cxx\cdots x)\lambda_z.u \rightarrow_B (\lambda_z.u)B \rightarrow_B u \). So shuffle reduction might allow us to define clever strategies that reduce terms via paths of relatively small terms.

6 Comparison with previous work and Conclusion

The last decade has seen an explosion in new notions of reductions which can be used for various purposes. Attempts at generalising reduction can be summarized by four axioms:

\[ \begin{align*}
(\theta) \quad & ((\lambda_x.N)P)Q \rightarrow (\lambda_x.NQ)P,
(\gamma) \quad & (\lambda_x.\lambda_y.N)P \rightarrow \lambda_y.(\lambda_x.N)P,
(\gamma_C) \quad & ((\lambda_x.\lambda_y.N)P)Q \rightarrow (\lambda_y.(\lambda_x.N)P)Q,
(g) \quad & ((\lambda_x.\lambda_y.N)P)Q \rightarrow (\lambda_x.N[y:=Q])P
\end{align*} \]

These rules attempt to make more redexes visible and to contract non-visible redexes. \( g \) is a combination of a \( \theta \)-step with a \( \beta \)-step. \( \gamma_C \) makes sure that \( \lambda_y \) and \( Q \) form a redex even before the redex based on \( \lambda_x \) and \( P \) is contracted. By compatibility, \( \gamma \) implies \( \gamma_C \). Moreover, \( ((\lambda_x.\lambda_y.N)P)Q \rightarrow (\lambda_y.(\lambda_x.N)P)Q \) and hence both \( \theta \) and \( \gamma_C \) put \( \lambda \) adjacent to its redex even before the redex based on \( \lambda_y \) and \( Q \) makes sure that \( \lambda_y \) and \( Q \) form a redex. Hence, we study \( \theta \)-like rules in this paper (which imply \( \gamma \)-like rules).

Now, we discuss where generalised reduction has been used (cf. [13,9]).

[19] introduces the notion of a premier redex which is similar to the redex based on \( \lambda_y \) and \( Q \) in the rule \( (\theta) \) above (which we call generalised redex). [20] uses \( \theta \) and \( \gamma \) (and calls the combination \( \sigma \)) to show that the perpetual reduction strategy finds the longest reduction path when the term is Strongly Normalizing (SN). [23] also introduces reductions similar to those of [20]. Furthermore, [11] uses \( \theta \) (and other reductions) to show that typability in ML is equivalent to acyclic semi-unification. [21] uses a reduction which has some common themes with \( \theta \). Nederpelt’s thesis in [18] and [5] uses \( \theta \) whereas [14] uses \( \gamma \) to reduce the problem of \( \beta \)-strong normalization to the problem of weak normalization (WN) for related reductions. [12] uses \( \theta \) and \( \gamma \) to reduce typability in the rank-2 restriction of the 2nd order \( \lambda \)-calculus to the problem of acyclic semi-unification. [16, 24, 22, 15] use related reductions to reduce SN to WN and [10] uses similar notions in SN proofs. [1] uses \( \theta \) (called “let-C”) as a part of an analysis of how to implement sharing in a real language interpreter in a way that directly corresponds to a formal calculus. [8] uses a more extended version of \( \theta \) (called term-reshuffling) and of \( g \) (called generalised reduction) where \( Q \) and \( N \) are not only separated by the redex \( (\lambda_x.N)P \) but by many redexes (ordinary and generalised).

After looking carefully at all these attempts, we realised that none of the extensions of reductions introduced so far can play as a role as approximating reductional equivalence. Of course all these notions have influenced our choice of the relation shuffle equivalence which we consider to be the best approximation of reductional equivalence.

This paper unifies all this previous work by looking for the class of reductionally equivalent terms. Such a class is conjectured to be undecidable but a decidable approximation of it which does indeed capture the existing new notions of reduction, is given in this paper. Our shuffle-equivalence is the closest decidable approximation to reductional equivalence up to now. Moreover, if \( A \rightarrow B \) where \( \rightarrow \) is a new notion of reduction given by the existing accounts (such as those of Moggi, Ariola et al., Regnier, Kfoury and Wells, Vidal, Kamareddine and Nederpelt, etc.) then \( A \) and \( B \) belong to the same shuffle-equivalence class under our approach of this paper. Furthermore, shuffle-equivalence classes partition \( \beta \)-equivalence classes into smaller parts.
In addition to the many notions of reduction where $A \rightarrow B$ implies $A$ and $B$ have the same reductional behavior, we find many extensions $\rightarrow_{e}$ of $\beta$-reduction $\rightarrow_{\beta}$ where if $A \rightarrow_{\beta} B$ then $A \rightarrow_{e} B$ and where the equivalence relation generated by $\rightarrow_{e}$ is just $\beta$-equality. These extensions make more redexes visible and hence allow for more flexibility in reducing a term.

We propose shuffle-reduction as a generalisation of these extensions. Shuffle-reduction does indeed accommodate the existing accounts and achieve their goals. In particular, we show that using shuffle-reduction we indeed may avoid size explosion without the cost of a longer reduction path, that it has the Church-Rosser property, and that the equivalence relation generated by shuffle-reduction is just $\beta$-equality.

We used the item-notation to give a clearer description of shuffle-equivalence and shuffle-reduction. We think that the item-notation is a good candidate for answering the two questions posed in the conclusions of [20] concerning the existence of a syntax for terms realising shuffle-equivalence (which Regnier [20] calls $\sigma$-equivalence).

References

17. E. Moggi. Computational $\lambda$-calculi and monads. LICS'89, 1989.
Computing Science Reports

Department of Mathematics and Computing Science
Eindhoven University of Technology

If you want to receive reports, send an email to: m.m.j.philips@tue.nl (we cannot guarantee the availability of the requested reports)

In this series appeared:

97/02 J. Hooman and O. v. Roosmalen

97/03 J. Blanco and A. v. Deuren
Basic Conditional Process Algebra, p. 20.

97/04 J.C.M. Baeten and J.A. Bergstra

97/05 J.C.M. Baeten and J.J. Vereijken

97/06 M. Franssen

97/07 J.C.M. Baeten and J.A. Bergstra
Bounded Stacks, Bags and Queues, p. 15.

97/08 P. Hoogendijk and R.C. Backhouse
When do datatypes commute? p. 35.

Communication Modeling: The Language/Action Perspective, p. 147.

97/10 P.C.N. v. Gorp, E.J. Luit, D.K. Hammer
Distributed real-time systems: a survey of applications and a general design model, p. 31.

97/11 A. Engels, S. Mauw and M.A. Reniers

97/12 D. Hauschildt, E. Verbeek and
W. van der Aalst

97/13 W.M.P. van der Aalst
Exploring the Process Dimension of Workflow Management, p. 56.

97/14 J.F. Groote, F. Monin and
J. Springintveld
A computer checked algebraic verfication of a distributed summation algorithm, p. 28.

97/15 M. Franssen
\(\lambda P\): A Pure Type System for First Order Logic with Automated Theorem Proving, p.35.

97/16 W.M.P. van der Aalst
On the verification of inter-organizational workflows, p. 23.

97/17 M. Vaccari and R.C. Backhouse
Calculating a Round-Robin Scheduler, p. 23.

97/18 Werkgemeenschap Informatiewetenschap
redactie: P.M.E. De Bra
Informatiewetenschap 1997
Wetenschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie
Informatiewetenschap, p. 60.

98/01 W. Van der Aalst

98/02 M. Voorhoeve
State / Event Net Equivalence, p. 25.

98/03 J.C.M. Baeten and J.A. Bergstra
Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.

98/04 R.C. Backhouse

98/05 D. Dams
Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.

98/06 G. v.d. Bergen, A. Kaldewaij
W.J. Diebissen
Maintenance of the Union of Intervals on a Line Revisited, p. 10.

98/07 Proceedings of the workshop on Workflow Management:
Net-based Concepts, Models, Techniques and Tools (WFM'98)

98/08 Informal proceedings of the Workshop on User Interfaces for Theorem Provers,
An analytical method for assessing business processes, p. 29.

Process Algebra in PVS

The Proof-assistant Yarrow, p. 15

Checking verifications of protocols and distributed systems by computer. Extended version of a tutorial at CONCUR'98, p. 27.

Structured Operational Semantics of X, p. 27

Diagnosing Workflow Processes using Woflan, p. 44

Final Dialgebra: From Categories to Allegories, p. 26

Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81

Efficient Evaluation of Triangular B-splines, p. 13

Inheritance of Workflows: An Approach to tackling problems related to change, p. 66


Requirements Specification and Analysis of Command and Control Systems

The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed μCRL, p. 44.

Process Algebra with Timing: Real Time and Discrete Time, p. 50.


A Framework for Component Based Software Architectures, p. 19

Why men (and octopuses) cannot juggle a four ball cascade, p. 10

An algorithm for the asynchronous Write-All problem based on process collision*, p. 11.


Inheritance of Behavior, p. 83

Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79

Real Time Process Algebra with Time-dependent Conditions, p. 33.

Mode Transfer in process Algebra, p. 14

Process Algebra with Explicit Termination, p. 17.

A process algebra for interworkings, p. 63.

Semantic Aspects of an Architecture for Distributed Embedded Systems*, p. 47.
<table>
<thead>
<tr>
<th>Page</th>
<th>Authors</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>00/05</td>
<td>J.F. Groote and M.A. Reniers</td>
<td>Algebraic Process Verification, p. 65.</td>
</tr>
<tr>
<td>00/06</td>
<td>J.F. Groote and J. v. Warnel</td>
<td>The Parallel Composition of Uniform Processes with Data, p. 19</td>
</tr>
<tr>
<td>00/07</td>
<td>C.A. Middelburg</td>
<td>Variable Binding Operators in Transition System Specifications, p. 27.</td>
</tr>
<tr>
<td>00/08</td>
<td>I.D. van den Ende</td>
<td>Grammars Compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases, p. 33.</td>
</tr>
<tr>
<td>00/09</td>
<td>R.R. Hoogerwoord</td>
<td>A Formal Development of Distributed Summation, p. 35</td>
</tr>
<tr>
<td>00/11</td>
<td>T. Basten and D. Bońański</td>
<td>Enhancing Partial-Order Reduction via Process Clustering, p. 14</td>
</tr>
<tr>
<td>00/12</td>
<td>S. Mauw, M.A. Reniers and T.A.C. Willemse</td>
<td>Message Sequence Charts in the Software Engineering Process, p. 26</td>
</tr>
<tr>
<td>00/13</td>
<td>J.C.M. Baeten, M.A. Reniers</td>
<td>Termination in Timed Process Algebra, p. 36</td>
</tr>
<tr>
<td>00/14</td>
<td>M. Voorhoeve, S. Mauw</td>
<td>Impossible Futures and Determinism, p. 14</td>
</tr>
<tr>
<td>00/15</td>
<td>M. Oostdijk</td>
<td>An Interactive Viewer for Mathematical Content based on Type Theory, p. 24.</td>
</tr>
</tbody>
</table>