A useful λ-notation[*]

Fairouz Kamareddine [†]
Department of Computing Science
17 Lilybank Gardens
University of Glasgow
Glasgow G12 8QQ, Scotland
email: `fairouz@dcs.glasgow.ac.uk`
and

Rob Nederpelt
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O.Box 513
5600 MB Eindhoven, the Netherlands
email: `wsinrpn@win.tue.nl`

September 16, 1996

Abstract

In this article, we introduce a λ-notation that is useful for many concepts of the λ-calculus. The new notation is a simple translation of the classical one. Yet, it provides many nice advantages.

First, we show that definitions such as compatibility, the heart of a term and β-redexes become simpler in item notation.

Second, we show that with this item notation, reduction can be generalised in a nice way. We find a relation $\rightsquigarrow_\beta$ which extends $\to_\beta$, which is Church Rosser and Strongly Normalising. This reduction relation may be the way to new reduction strategies. In classical notation, it is much harder to present this generalised reduction in a convincing manner.

Third, we show that the item notation enables one to represent in a very simple way the canonical type $\tau(\Gamma, A)$ of a term A in context Γ. This canonical type plays the role of a preference type and can be used to split $\Gamma \vdash A : B$ in the two parts: $\Gamma \vdash A$ and $\tau(\Gamma, A) = B$. This means that the question *is A typable with a type B* is divided in two questions: *is A typable* and *is B in the class of types of A*. It turns out that calculating

[*] *Theoretical Computer Science 155*, 1996. We are grateful for the comments of the editor and the anonymous referee on an earlier version of the article.

[†]Kamareddine is grateful to the Department of Mathematics and Computing Science, Eindhoven University of Technology, for their financial support and hospitality from October 1991 to September 1992, and during various short visits since 1993. Furthermore, Kamareddine is grateful to the Department of Mathematics and Computer Science, University of Amsterdam, and in particular to Jan Bergstra and Inge Bethke for their hospitality during the preparation of this article, and to the Dutch organisation of research (NWO) for its financial support. Last but not least, Kamareddine is grateful to the ESPRIT Basic Action for Research project "Types for Proofs and Programming" for its financial support.

this preference type of A in item notation is a straightforward operation. One just goes through A from left to right performing very trivial steps on the items til the end variable (or heart) of A is reached.

Fourth, we can with this item notation, find the parts of a term t relevant for a variable occurrence x° in terms of binding, typing and substitution. Again, this part of t, $t \upharpoonright x^\circ$, is very easy to find in item notation. Just take the part of t to the left of x° and remove all unmatched parentheses.

Fifth, we reflect on the status of variables and show that indeed it is easy to study this status in item notation.

Finally, we show that for a substitution calculus à la de Bruijn with open terms, it is simpler to describe normal forms using item notation.

There are further advantages of item notation that are studied elsewhere. For example, in [9], we show that explicit substitution is easily built in item notation and that global and local strategies of substitution can be accommodated. In [10], we show that with item notation, one can give a unified approach to type theory.

An implementation of this item notation with most of the concepts discussed in this paper can be found in [15].

Keywords: *Item notation, Reduction, Canonical Typing, Term restriction.*

1 The formal machinery of the Cube in classical notation

In this section we introduce the Cube (see [2]) and the usual necessary notions to manipulate terms and types.

The systems of the Cube, are based on a set of *pseudo-expressions* or *terms* $\mathcal{T}$ defined by the following abstract syntax (let π range over both Π and λ):

$$\mathcal{T} = * \mid \Box \mid V \mid \mathcal{T}\mathcal{T} \mid \pi_{V:\mathcal{T}}.\mathcal{T}$$

where V is an infinite collection of variables over which $x, y, z, \ldots$ range. $*$ and $\Box$ are called sorts over which $S, S_1, S_2, \ldots$ are used to range. We take $A, B, C, a, b \ldots$ to range over $\mathcal{T}$.

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. We write $A[x := B]$ to denote the term where all the free occurrences of x in A have been replaced by B. Furthermore, we take terms to be equivalent up to variable renaming. For example, we take $\lambda_{x:A}.x \equiv \lambda_{y:A}.y$ where $\equiv$ is used to denote syntactical equality of terms. We assume moreover, the Barendregt variable convention which is formally stated as follows:

Convention 1.1 *(BC: Barendregt's Convention)*
Names of bound variables will always be chosen such that they differ from the free ones in a term. Moreover, different λ's have different variables as subscript. Hence, we will not have $(\lambda_{x:A}.x)x$, but $(\lambda_{y:A}.y)x$ instead.

The following notions play an important role in the typing of terms:

Definition 1.2 *(Type of Bound Variables, $\heartsuit$)*

1. *If x occurs free in B, then all its occurrences are bound with type A in $\pi_{x:A}.B$.*
2. *If an occurrence of x is bound with type A in B, then it is also bound with type A in $\pi_{y:C}.B$ for $y \not\equiv x$, in BD, and in DB.*

3. Define $\heartsuit(*) = *$, $\heartsuit(\Box) = \Box$, $\heartsuit(x) = x$, $\heartsuit(\pi_{x:A}.B) = \heartsuit(B)$ and $\heartsuit(AB) = \heartsuit(A)$.

Terms can be related via a reduction relation. An example is β-reduction (see Definition 1.4). A reduction relation satisfies compatibility:

Definition 1.3 *(Compatibility of a reduction relation in classical notation)*
We say that a reduction relation $\to$ on terms is compatible iff the following holds:

$$\frac{A_1 \to A_2}{A_1 B \to A_2 B} \qquad \frac{B_1 \to B_2}{AB_1 \to AB_2}$$

$$\frac{A_1 \to A_2}{\pi_{x:A_1}.B \to \pi_{x:A_2}.B} \qquad \frac{B_1 \to B_2}{\pi_{x:A}.B_1 \to \pi_{x:A}.B_2}$$

Definition 1.4 *(β-redexes, β-reduction $\to_\beta$ for the Cube)*
A β-redex is of the form $(\lambda_{x:B}.A)C$. β-reduction $\to_\beta$, is the least compatible relation generated out of the following axiom:

$$(\beta) \qquad (\lambda_{x:B}.A)C \to_\beta A[x := C]$$

We take $\twoheadrightarrow_\beta$ to be the reflexive transitive closure of $\to_\beta$ and we take $=_\beta$ to be the least equivalence relation generated by $\twoheadrightarrow_\beta$.

A *statement* is of the form $A : B$ with $A, B \in \mathcal{T}$. A is the *subject* and B is the *predicate* of $A : B$. Moreover, A *declaration* is of the form $\lambda_{x:A}$ with $A \in \mathcal{T}$ and $x \in V$. A *pseudo-context* is a finite ordered sequence of declarations, all with distinct subjects. The *empty* context is denoted by $<>$. If $\Gamma = \lambda_{x_1:A_1}.\ldots.\lambda_{x_n:A_n}$ then $\Gamma.\lambda_{x:B} = \lambda_{x_1:A_1}.\ldots.\lambda_{x_n:A_n}.\lambda_{x:B}$ and $dom(\Gamma) = \{x_1, \ldots, x_n\}$. We use $\Gamma, \Delta, \Gamma', \Gamma_1, \Gamma_2, \ldots$ to range over pseudo-contexts.

A *typability* relation $\vdash$ is a relation between pseudo-contexts and pseudo-expressions written as $\Gamma \vdash A$. The *rules of typability* establish which *judgements* $\Gamma \vdash A$ can be derived. A judgement $\Gamma \vdash A$ states that A is typable in the pseudo-context Γ. When $\Gamma \vdash A$ then A is called a (legal) *expression* and Γ is a (legal) *context*.

A *type assignment* relation is a relation between a pseudo-context and two pseudo-expressions written as $\Gamma \vdash A : B$. The *rules of type assignment* establish which *judgements* $\Gamma \vdash A : B$ can be derived. A judgement $\Gamma \vdash A : B$ states that $A : B$ can be derived from the pseudo-context Γ. When $\Gamma \vdash A : B$ then A and B are called (legal) *expressions* and Γ is a (legal) *context*.

We write $\Gamma \vdash A : B : C$ for $\Gamma \vdash A : B \wedge \Gamma \vdash B : C$. If $\Delta \equiv \lambda_{x_1:A_1}.\ldots.\lambda_{x_n:A_n}$ with $n \geq 0$ is a pseudo-context, then $\Gamma \vdash \Delta$, for Γ a type assignment, means $\Gamma \vdash x_i : A_i$ for $1 \leq i \leq n$. If $A \to B$ then we also say $\Gamma_1.\lambda_{x:A}.\Gamma_2 \to \Gamma_1.\lambda_{x:B}.\Gamma_2$ and define $\twoheadrightarrow$ on pseudo-contexts to be the reflexive transitive closure of $\to$.

Remark 1.5 Note that we differ from [2] in that we take a declaration to be $\lambda_{x:A}$ rather than $x : A$. The reason for this is that we want pseudo-contexts to be as close as possible to terms. In fact the context Γ can be mapped to the term $\Gamma.*$ for example, and definitions of boundness/freeness of variables in a term and the Barendregt convention are thus easily extended to pseudo-contexts.

The systems of the cube, as established by the type assignment in Definition 1.6 below, are distinguished by the set of *sort-rules* (S_1, S_2) allowed in the formation rule. Since $(*, *)$ is always taken to be a sort-rule, there are 8 different choices for this set, which correspond to the vertices of a cube.

Definition 1.6 *($\vdash_\beta$) The type assignement relation $\vdash_\beta$ is defined by the following rules:*

(axiom) $$<> \vdash_\beta * : \Box$$

(start rule) $$\frac{\Gamma \vdash_\beta A : S}{\Gamma.\lambda_{x:A} \vdash_\beta x : A}\ x \not\in \Gamma$$

(weakening rule) $$\frac{\Gamma \vdash_\beta A : S \qquad \Gamma \vdash_\beta D : E}{\Gamma.\lambda_{x:A} \vdash_\beta D : E}\ x \not\in \Gamma$$

(application rule) $$\frac{\Gamma \vdash_\beta F : \Pi_{x:A}.B \qquad \Gamma \vdash_\beta a : A}{\Gamma \vdash_\beta Fa : B[x := a]}$$

(abstraction rule) $$\frac{\Gamma.\lambda_{x:A} \vdash_\beta b : B \qquad \Gamma \vdash_\beta \Pi_{x:A}.B : S}{\Gamma \vdash_\beta \lambda_{x:A}.b : \Pi_{x:A}.B}$$

(conversion rule) $$\frac{\Gamma \vdash_\beta A : B \qquad \Gamma \vdash_\beta B' : S \qquad B =_\beta B'}{\Gamma \vdash_\beta A : B'}$$

(formation rule) $$\frac{\Gamma \vdash_\beta A : S_1 \qquad \Gamma.\lambda_{x:A} \vdash_\beta B : S_2}{\Gamma \vdash_\beta \Pi_{x:A}.B : S_2}\ \textit{if } (S_1, S_2) \textit{ is a rule}$$

2 The item notation

Our new notation (the *item notation*) is not that different from the classical one. Nonetheless, it has some attractive features. In this section, we introduce the notation and point out some of the notions of Section 1 (compatibility, ♡, the visibility of a β-redex) that become simpler in item notation. The item notation is really an improvement over the classical one as can be seen from the following section. For this section however, let us start by giving the translation from classical to item notation.

Definition 2.1 *(Item notation)*
Define $\mathcal{I}$ which translates terms from classical notation to item notation such that:

$$\begin{array}{lcll} \mathcal{I}(A) & = & A & \textit{if } A \in \{*, \Box\} \cup V \\ \mathcal{I}(\pi_{x:A}.B) & = & (\mathcal{I}(A)\pi_x)\mathcal{I}(B) & \\ \mathcal{I}(AB) & = & (\mathcal{I}(B)\delta)\mathcal{I}(A) & \end{array}$$

The reason for using this format is, that both abstraction and application can be seen as the process of fixing a certain part (an "**item**") to a term:

- the abstraction $\pi_{v:t'}.t$ is obtained by prefixing the abstraction-item $\pi_{v:t'}$ to the term t. Hence, $(t'\pi_v)t$ is obtained by prefixing $(t'\pi_v)$ to t.
- the application tt' (in "classical" notation) is obtained by postfixing the argument-item t' to the term t. Now $(t'\delta)t$ is obtained by prefixing $(t'\delta)$ to t.

(It should be noted that in the *Automath*-tradition, in which also the 'argument' t' precedes the 'function' t in an application (see [16]), an abstraction-item $\lambda_{v:t'}$ (or $(t'\lambda_v)$ in our new notation) is called an *abstractor* and denoted as $[v : t']$. An argument-item t' (or $(t'\delta)$ in our notation) is called an *applicator* and denoted either as $\{t'\}$ or as $< t' >$.)

Example 2.2

$$\begin{array}{lcl} \mathcal{I}((\lambda_{x:y}.x)u) & \equiv & (u\delta)(y\lambda_x)x \\ \mathcal{I}(u(\lambda_{x:y}.x)) & \equiv & ((y\lambda_x)x\delta)u \\ \mathcal{I}((\lambda_{y:z}.\lambda_{x:z}.y)u) & \equiv & (u\delta)(z\lambda_y)(z\lambda_x)y \end{array}$$

It may be helpful to see the item notation in terms of trees. Take $(\lambda_{x:z}.xy)u$ and its graphical representation as in Figure 1.

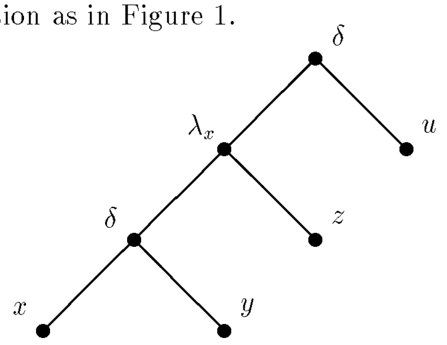


Figure 1: binary tree of $(\lambda_{x:z}.xy)u$

Now, instead of drawing trees as in Figure 1, we will rotate them anticlockwise by 135 degree hence obtaining for Figure 1, the picture given in Figure 2.

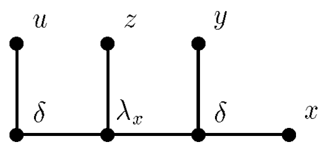


Figure 2: layered tree of $(\lambda_{x:z}.xy)u$

We call such trees *layered trees*. This representation of trees is very important for our purposes. It will turn out to have essential advantages in developing a term, theoretically as well as in practical applications of typed lambda calculi. (This observation is due to de Bruijn, see [4] or [5].) Those layered trees furthermore, correspond to the item notation. In fact, look at the tree in Figure 2 and write every vertical line as an item starting from left and from top. What you get is nothing but the item notation of the term. That is: $(u\delta)(z\lambda_x)(y\delta)x$.

Even though $\mathcal{I}$ is simple, $\mathcal{I}(A)$ (or A in item notation) will have many attractive characteristics that A in classical notation does not have. Notice first that the definition of compatibility of a reduction relation (Definition 1.3) becomes simpler in item notation:

Definition 2.3 *(Compatibility of a reduction relation in item notation)*
Let $\omega \in \{\delta\} \cup \{\pi_x \mid x \in V\}$. A reduction relation $\rightarrow$ is compatible iff the following holds:

$$\frac{A_1 \rightarrow A_2}{(A_1\omega)B \rightarrow (A_2\omega)B} \qquad\qquad \frac{B_1 \rightarrow B_2}{(A\omega)B_1 \rightarrow (A\omega)B_2}$$

Remark 2.4 Definition 2.3 may not be seen as a great improvement over Definition 1.3. But just imagine that in the λ-calculus you had not only λ and δ as internal operators but also σ for substitution, τ for typing and so on. In fact, internalising substitution (i.e. making it explicit) has been a topic of research in the last decade (see [1], [8], [7], [9]). Now, internalising extra operators means that in classical notation, in Definition 1.3, two extra rules are added for each new operator. In item notation on the other hand, Definition 2.3 does not depend on the number of operators. Simply, the set of operators to which ω belongs will increase.

As item notation is a translation of classical notation, all definitions of Section 1 (written in item notation) hold. Let us however define some characteristic notions of item notation:

Definition 2.5 *((main) items, (main, $\delta\pi$-)segments, heart, weight)*

- *If x is a variable and A is a pseudo-expression then $(A\lambda_x)$, $(A\Pi_x)$ and $(A\delta)$ are* **items** *(called λ-item, Π-item and δ-item respectively). We use $s, s_1, s_i, \ldots$ to range over items.*
- *A concatenation of zero or more items is a* **segment**. *We use $\overline{s}, \overline{s}_1, \overline{s}_i, \ldots$ as meta-variables for segments. We write $\emptyset$ for the empty segment.*
- *Each pseudo-expression A is the concatenation of zero or more items and a variable or sort: $A \equiv s_1 s_2 \cdots s_n x$ or $A \equiv s_1 s_2 \cdots s_n S$. These items $s_1, s_2, \ldots, s_n$ are called the* **main items** *of A, x (or S) is called the* **heart** *of A, notation $\heartsuit(A)$.*
- *Analogously, a segment $\overline{s}$ is a concatenation of zero or more items: $\overline{s} \equiv s_1 s_2 \cdots s_n$; again, these items $s_1, s_2, \ldots, s_n$ (if any) are called the* **main items**, *this time of $\overline{s}$.*
- *A concatenation of adjacent main items $s_m \cdots s_{m+k}$, is called a* **main segment**.
- *A* **$\delta\pi$-segment** *is a δ-item immediately followed by a π-item.*
- *The* **weight** *of a segment $\overline{s}$, $\mathtt{weight}(\overline{s})$, is the number of main items that compose the segment. Moreover, we define $\mathtt{weight}(\overline{s}x) = \mathtt{weight}(\overline{s})$.*

Remark 2.6 Note that the heart of a variable is immediately visible in item notation. There was no need to follow Definition 1.2. For example, Let $A = \Pi_{z:*}.(\lambda_{y:*}.(\lambda_{x:*}.x)y)(\Pi_{w:*}.(\lambda_{x:*}.x)y)$. Then $\mathcal{I}(A) \equiv (*\Pi_z)((*\Pi_w)(y\delta)(*\lambda_x)x\delta)(*\lambda_y)(y\delta)(*\lambda_x)x$. Now, $\heartsuit(A) = x$ is much easier to find in item notation as it is the last variable in the term.

Now we come to β-reduction. Let us write Definition 1.4 in item notation:

Definition 2.7 *(β-redexes, reducible segment, β-reduction $\to_\beta$ in item notation)*
A β-redex is of the form $(C\delta)(B\lambda_x)A$. We call $(C\delta)(B\lambda_x)A$ a reducible segment. β-reduction $\to_\beta$, is the least compatible relation generated out of the following axiom:

$$(\beta) \qquad (C\delta)(B\lambda_x)A \to_\beta A[x := C]$$

We take $\twoheadrightarrow_\beta$ to be the reflexive transitive closure of $\to_\beta$ and we take $=_\beta$ to be the least equivalence relation generated by $\twoheadrightarrow_\beta$.

Note here that in item notation, a β-redex always starts with a δ-item immediately followed by a λ-item ($\delta\lambda$-segment). Hence, in item notation it is easy to see a redex. That is, the body of a term (A above) does not separate the $\lambda_{x:B}$ from its potential argument C.

In item notation, we can do even better than making redexes more visible. We can find new redexes that are not visible in classical notation. This is done in Section 3.

3 Reduction

As types do not play a big role in the illustartion of our point, we shall in this section, ignore them. I.e., we write λ-items as (λ_x). The following example illustrates the need for generalised reduction.

Example 3.1 In the classical term $t \equiv ((\lambda_x.(\lambda_y.\lambda_z.zd)c)b)a$, we have the following redexes (the fact that neither y nor x appear as free variables in their respective scopes does not matter here; this is just to keep the example simple and clear):

1. $(\lambda_y.\lambda_z.zd)c$
2. $(\lambda_x.(\lambda_y.\lambda_z.zd)c)b$

Written in item notation, t becomes $(a\delta)(b\delta)(\lambda_x)(c\delta)(\lambda_y)(\lambda_z)(d\delta)z$. Here, the two classical redexes correspond to $\delta\lambda$-pairs as follows:

1. $(\lambda_y.\lambda_z.zd)c$ corresponds to $(c\delta)(\lambda_y)$. We ignore $(\lambda_z)(d\delta)z$ as it is easily retrievable in item notation. It is the maximal subterm of t to the right of (λ_y).
2. $(\lambda_x.(\lambda_y.\lambda_z.zd)c)b$ corresponds to $(b\delta)(\lambda_x)$. Again $(c\delta)(\lambda_y)(\lambda_z)(d\delta)z$ is ignored for the same reason as above.

There is however a third redex which is not visible in the classical term. Namely, $(\lambda_z.zd)a$. Such a redex will only be visible after we have contracted the above two redexes (we will not discuss the order here). In fact, assume we contract the second redex in the first step, and the first redex in the second step. I.e.

Classical Notation		*Item Notation*	
$(\underline{(\lambda_x.(\lambda_y.\lambda_z.zd)c)b})a$	$\to_\beta$	$(a\delta)\underline{(b\delta)(\lambda_x)}(c\delta)(\lambda_y)(\lambda_z)(d\delta)z$	$\to_\beta$
$(\underline{(\lambda_y.\lambda_z.zd)c})a$	$\to_\beta$	$(a\delta)\underline{(c\delta)(\lambda_y)}(\lambda_z)(d\delta)z$	$\to_\beta$
$\underline{(\lambda_z.zd)a}$	$\to_\beta ad$	$\underline{(a\delta)(\lambda_z)}(d\delta)z$	$\to_\beta (d\delta)a$

Now, even though all these redexes (i.e. the first, second and third) are *needed* in order to get the normal form of t, only the first two were visible in the classical term at first sight. The third could only be seen once we have contracted the first two reductions. In item notation, the third redex $(\lambda_z.zd)a$ corresponds to $(a\delta)(\lambda_z)$ but the δ-item and the λ-item are separated by the segment $(b\delta)(\lambda_x)(c\delta)(\lambda_y)$. By extending the notion of a redex and of β-reduction, we can make this redex visible and we can contract it before the other redexes. Figure 3 shows the possible redexes.

The idea is simple; we generalise the notion of a reducible segment $(b\delta)(\lambda_v)$ to a **reducible couple** being an item $(b\delta)$ and an item (λ_v) separated by a segment $\overline{s}$ which is a **well-balanced segment**. Here is the definition of well-balanced segments:

Definition 3.2 *(well-balanced segments)*

- *The empty segment $\emptyset$ is a well-balanced segment.*
- *If $\overline{s}$ is well-balanced, then $(A\delta)\overline{s}(B\pi_x)$ is well-balanced.*

- *The concatenation of well-balanced segments is a well-balanced segment.*

A well-balanced segment has the same structure as a matching composite of opening and closing brackets, each δ- (or π-)item corresponding with an opening (resp. closing) bracket. That is, we see immediately that the redexes in t originate from the couples $(b\delta)(\lambda_x)$, $(c\delta)(\lambda_y)$

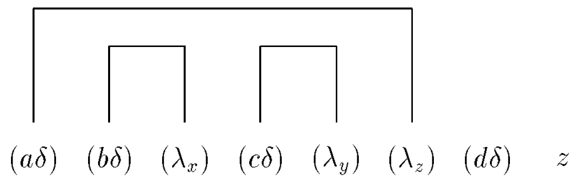


Figure 3: Redexes in item notation

and $(a\delta)(\lambda_z)$. This natural matching was not present in the classical notation of t.

Having argued above that β-reduction should not be restricted to the reducible segments but may take into account other candidates, we can extend our notion of β-reduction in this vein. That is to say, we may allow reducible *couples* to have the same "reduction rights" as reducible *segments*. That is, the β-reduction of Definition 2.7 changes to the following:

Definition 3.3 *(Extended redexes and general β-reduction $\rightsquigarrow_\beta$)*
An extended redex is of the form $(b\delta)\overline{s}(\lambda_v)a$, where $\overline{s}$ is well-balanced. We call $(b\delta)(\lambda_v)a$ a reducible couple. Moreover, one-step general β-reduction $\rightsquigarrow_\beta$, is the least compatible relation generated out of the following axiom:

$$(\textit{general } \beta) \qquad (b\delta)\overline{s}(\lambda_v)a \rightsquigarrow_\beta \overline{s}\{a[v := b]\} \qquad \textit{if } \overline{s} \textit{ is well-balanced}$$

Many step general β-reduction $\rightsquigarrow\!\!\!\!\rightarrow_\beta$ is the reflexive transitive closure of $\rightsquigarrow_\beta$.

Example 3.4 Take Example 3.1. As $(b\delta)(\lambda_x)(c\delta)(\lambda_y)$ is a well-balanced segment, then $(a\delta)(\lambda_z)$ is a reducible couple and

$$\begin{array}{ll} t \equiv (a\delta)(b\delta)(\lambda_x)(c\delta)(\lambda_y)(\lambda_z)(d\delta)z & \rightsquigarrow_\beta \\ (b\delta)(\lambda_x)(c\delta)(\lambda_y)\{((d\delta)z)[z := a]\} & \equiv \\ (b\delta)(\lambda_x)(c\delta)(\lambda_y)(d\delta)a & \end{array}$$

The *reducible couple* $(a\delta)(\lambda_z)$ also has a corresponding ("generalized") redex in the traditional notation, which will appear after two one-step β-reductions, leading to $(\lambda_z.zd)a$. With our generalised one-step β-reduction we could reduce $((\lambda_x.(\lambda_y.\lambda_z.zd)c)b)a$ to $(\lambda_x.(\lambda_y.ad)c)b$. This reduction is difficult to carry out in the classical λ-calculus. We believe that this generalised reduction can only be obtained tidily in a system formulated using our item notation: it is the item notation which enables us to extend reduction smoothly beyond $\twoheadrightarrow_\beta$. Because a well-balanced segment may be empty, the general β-reduction rule presented above is really an extension of the classical β-reduction rule. In [11], we show that:

1. If $a \rightarrow_\beta b$ then $a \rightsquigarrow_\beta b$.
2. If $a \rightsquigarrow\!\!\!\!\rightarrow_\beta$ then $a =_\beta b$.
3. $\rightsquigarrow_\beta$ is Church Rosser.

An alternative to the generalised notion of β-reduction can be obtained by keeping the old β-reduction and by *reshuffling* the term in hand. This reshuffling transports δ-items of $\delta\lambda$-couples through the term until they *immediately precede* their corresponding λ-items. So $(a\delta)(b\delta)(\lambda_x)(c\delta)(\lambda_y)(\lambda_z)(d\delta)z$ can be reshuffled to $(b\delta)(\lambda_x)(c\delta)(\lambda_y)(a\delta)(\lambda_z)(d\delta)z$ by moving $(a\delta)$ to the right, in order to transform the bracketing structure {{ }{ }} into { }{ }{ }, where all the redexes correspond to adjacent '{' and '}'. In other words, Figure 3 can be redrawn using term reshuffling in Figure 4. Such a reshuffling is more difficult to describe in

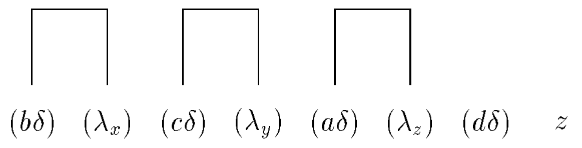


Figure 4: Term reshuffling in item notation

classical notation. I.e. it is hard to say what exactly happened when $((\lambda_x.(\lambda_y.\lambda_z.zd)c)b)a$, is reshuffled to $(\lambda_x.(\lambda_y.(\lambda_z.zd)a)c)b$. This is another attractive feature of our item notation. In [11], we define a reshuffled form $TS(a)$ of a such that all the application items occur next to their matching abstraction items. We show moreover, that if $a \rightsquigarrow_\beta b$ then $(\exists c)[(TS(a) \rightarrow_\beta c) \wedge TS(c) \equiv TS(b)]$.

We illustrated in this section that reduction can take new dimensions in item notation. We have used however only the type free calculus in this section and said that our resulting reduction is Church Rosser (CR). One might ask what will happen if we use this extended reduction in type systems. In other words, if we extend the cube of Section 1 with this reduction, do we get all the original properties of the cube? In [3], we studied the cube with this general reduction and we obtained that all the properties of the cube including Strong Normalisation SN, except Subject Reduction SR, still hold with this general reduction. We did find however that if definitions are also added to the cube, then SR holds. The addition of definitions should not be looked at as a negative result. In fact, most implementations of important type systems do use definitions. Picture 5 illustrates our results about the cube. We call the cube of Section 1, C, the cube extended with general reduction, $C_{\rightsquigarrow_\beta}$, the cube extended with definitions, C_{def} and the cube extended with both definitions and general reduction, $C_{\rightsquigarrow_\beta, def}$. Picture 5 shows that C, C_{def} and $C_{\rightsquigarrow_\beta, def}$ all satisfy CR, SN and CR. The cube C extended with general reduction, $C_{\rightsquigarrow_\beta}$, satisfies all the properties except SR.

$$\begin{tikzcd}
 & C(\text{CR, SN, SR}) \arrow[dl] \arrow[dr] & \\
C_{\rightsquigarrow_\beta}(\text{CR, SN}) \arrow[dr] & & C_{def}(\text{CR, SN, SR}) \arrow[dl] \\
 & C_{\rightsquigarrow_\beta def}\ (\text{CR, SN, SR}) &
\end{tikzcd}$$

Figure 5: Properties of the Cube with various extensions

4 The structure of terms

We may categorize the main items of a term t into different classes:

1. The "partnered" items (i.e. the application and abstraction items which are partners, hence "coupled" to a matching one).
2. The "bachelors" (i.e. the abstraction and application items which have no matching counterpart).

Let us first give this definition:

Definition 4.1 *(match, $\delta\pi$- (reducible) couple, partner, partnered, bachelor)*
Let $A \in \mathcal{T}$. Let $\overline{s} \equiv s_1 \cdots s_n$ be a segment occurring in A.

- *We say that s_i and s_j* **match**, *when $1 \leq i < j \leq n$, s_i is a δ-item, s_j is an π-item, and $s_{i+1} \cdots s_{j-1}$ is a well-balanced segment.*
- *If s_i and s_j match, we call $s_i s_j$ a* $\delta\pi$-**couple**. *A $\delta\lambda$-couple is called a* **reducible couple**.
- *If s_i and s_j match, we call s_i and s_j the* **partners** *or* **partnered** *items.*
- *All non-partnered π- (or δ-)items s_k in A, are called* **bachelor** *π- (resp. δ-)items.*
- *A segment consisting of bachelor items only, is called a* **bachelor segment**.

Lemma 4.2 *Let $\overline{s}$ be the body of a term a. Then the following holds in $\overline{s}$:*

1. *Each bachelor main abstraction item precedes each bachelor main application item.*
2. *The removal from $\overline{s}$ of all bachelor main items, leaves behind a* well-balanced *segment.*
3. *The removal from $\overline{s}$ of all main reducible couples, leaves behind $(\lambda_{v_1}) \ldots (\lambda_{v_n})(a_1\delta) \ldots (a_m\delta)$, the segment consisting of all bachelor main abstraction and application items.*

Proof: *1 is by induction on $\texttt{weight}(\overline{s'})$ for $\overline{s} \equiv \overline{s'}(\lambda_v)\overline{s''}$ and (λ_v) bachelor in $\overline{s}$. 2 and 3 are by induction on $\texttt{weight}(\overline{s})$.* □

Note that we have assumed $\emptyset$ well-balanced. We assume it moreover non-bachelor.

Corollary 4.3 *For each non-empty segment $\overline{s}$, there is a unique* partitioning *in segments $\overline{s_0}, \overline{s_1}, \cdots, \overline{s_n}$, such that*

1. *$\overline{s} \equiv \overline{s_0}\,\overline{s_1} \cdots \overline{s_n}$,*
2. *For all $0 \leq i \leq n$, $\overline{s_i}$ is well-balanced in $\overline{s}$ for even i and $\overline{s_i}$ is bachelor in $\overline{s}$ for odd i.*
3. *If $\overline{s_i}$ and $\overline{s_j}$ for $0 \leq i, j \leq n$ are bachelor abstraction resp. application segments, then $\overline{s_i}$ precedes $\overline{s_j}$ in $\overline{s}$.*
4. *If $i \geq 1$ then $\overline{s_{2i}} \not\equiv \emptyset$.*
5. *$\overline{s_n} \not\equiv \emptyset$.* □

This is actually a very nice corollary. It tells us a lot about the structure of our terms.

Example 4.4 $\overline{s} \equiv (\lambda_x)(\lambda_y)(a\delta)(\lambda_z)(\lambda_{x'})(b\delta)(c\delta)(d\delta)(\lambda_{y'})(\lambda_{z'})(e\delta)$, has the partitioning:

- well-balanced segment $\overline{s_0} \equiv \emptyset$
- bachelor segment $\overline{s_1} \equiv (\lambda_x)(\lambda_y)$,
- well-balanced segment $\overline{s_2} \equiv (a\delta)(\lambda_z)$,
- bachelor segment $\overline{s_3} \equiv (\lambda_{x'})(b\delta)$,
- well-balanced segment $\overline{s_4} \equiv (c\delta)(d\delta)(\lambda_{y'})(\lambda_{z'})$,
- bachelor segment $\overline{s_5} \equiv (e\delta)$.

5 The canonical typing operator τ

In this section, we introduce a notion that will play an important role in the question of typability of terms. This notion enables one to separate the judgement $\Gamma \vdash A : B$ in two ($\Gamma \vdash A$ and $\tau(\Gamma, A) = B$). This division of $\Gamma \vdash A : B$ has been studied in detail for the classical notation in [12]. Here, we introduce canonical typing and show that calculating the canonical type of a term in item notation is a lot simpler than in classical notation.

Definition 5.1 *(Canonical Type Operator) For any pseudo-context Γ and pseudo-expression A, we define the canonical type of A in Γ, $\tau(\Gamma, A)$ as follows:*

$$\begin{array}{lcll}
\tau(\Gamma, *) & \equiv & \Box & \\
\tau(\Gamma, x) & \equiv & A \textit{ if } (A\lambda_x) \in \Gamma & \\
\tau(\Gamma, (a\delta)F) & \equiv & (a\delta)\tau(\Gamma, F) & \\
\tau(\Gamma, (A\lambda_x)B) & \equiv & (A\Pi_x)\tau(\Gamma(A\lambda_x), B) & \textit{if } x \notin dom(\Gamma) \\
\tau(\Gamma, (A\Pi_x)B) & \equiv & \tau(\Gamma(A\lambda_x), B) & \textit{if } x \notin dom(\Gamma)
\end{array}$$

When $\tau(\Gamma, A)$ is defined, we write $\downarrow \tau(\Gamma, A)$.

Note that $\tau(\Gamma, A)$ might contain a $\delta\Pi$-segment and hence we may need to talk about $\rightarrow_\beta$ as well as $\rightarrow_{\beta\Pi}$. We will not discuss Π-reduction here (see [12]), except in Example 5.8.

Here are some of the properties of τ:

Lemma 5.2 *(τ-weakening)*
Let Γ, Γ' be pseudo-contexts. $\Gamma \subseteq \Gamma' \wedge \downarrow \tau(\Gamma, A) \Rightarrow [\downarrow \tau(\Gamma', A)$ and $\tau(\Gamma, A) \equiv \tau(\Gamma', A)]$.

Proof: *By induction on A, noting that bound variables in A can always be renamed so that they don't occur in $dom(\Gamma')$.* □

Lemma 5.3 *(Context-reduction for τ)*
For Γ, Γ' be pseudo-contexts, $\Gamma \twoheadrightarrow_\beta \Gamma' \wedge \downarrow \tau(\Gamma, A) \Rightarrow [\downarrow \tau(\Gamma', A) \wedge \tau(\Gamma, A) \twoheadrightarrow_\beta \tau(\Gamma', A)]$.

Proof: *By induction on $\tau(\Gamma, A)$.* □

Lemma 5.4 *(τ-restriction)*
If $\downarrow \tau(\Gamma, A)$ then $\tau(\Gamma \lceil FV(A), A) \equiv \tau(\Gamma, A)$.

Proof: *By induction on A.* □

Lemma 5.5 *(τ-Substitution Lemma) Let $\sim$ be $\twoheadrightarrow_{\beta\Pi}, =_{\beta\Pi}$ or $\equiv$.*
If $\tau(\Gamma(A\lambda_x)\Delta, B) \equiv C$ and $\tau(\Gamma, D) \sim A$ then $\tau(\Gamma(\Delta[x := D]), B[x := D]) \sim C[x := D]$.

Proof: *By induction on the structure of A.* $\Box$

Example 5.6 In usual type theory, the type of $(*\lambda_x)(x\lambda_y)y$ is $(*\Pi_x)(x\Pi_y)x$ and the type of $(*\Pi_x)(x\Pi_y)x$ is $*$. Now, with our τ, we get the same result:
$\tau(<>, (*\lambda_x)(x\lambda_y)y) \equiv (*\Pi_x)\tau((*\lambda_x), (x\lambda_y)y) \equiv (*\Pi_x)(x\Pi_y)\tau((*\lambda_x)(x\lambda_y), y) \equiv (*\Pi_x)(x\Pi_y)x$
$\tau(<>, (*\Pi_x)(x\Pi_y)x) \equiv \tau((*\lambda_x), (x\Pi_y)x) \equiv \tau((*\lambda_x)(x\lambda_y), x) \equiv *$

Now, here is an example written in both item and classical notation.

Example 5.7

$$\begin{array}{lllll} \tau(<>, \Pi_{z:*}. & (\lambda_{y:\Box}. & (\lambda_{x:\Box}. & y &)*)(\Pi_{w:*}.(\lambda_{x:*}.x)w)) \equiv \\ & (\Pi_{y:\Box}. & (\Pi_{x:\Box}. & \Box &)*)(\Pi_{w:*}.(\lambda_{x:*}.x)w) \end{array}$$

$$\begin{array}{lllllll} \mathcal{I}(A) & \equiv (*\Pi_z) & ((*\Pi_w)(w\delta)(*\lambda_x)x\delta) & (\Box\lambda_y) & (*\delta) & (\Box\lambda_x) & y \\ \tau(<>, \mathcal{I}(A)) & \equiv & ((*\Pi_w)(w\delta)(*\lambda_x)x\delta) & (\Box\Pi_y) & (*\delta) & (\Box\Pi_x) & \tau((*\lambda_z)(\Box\lambda_y)(\Box\lambda_x), x) \\ & \equiv & ((*\Pi_w)(w\delta)(*\lambda_x)x\delta) & (\Box\Pi_y) & (*\delta) & (\Box\Pi_x) & \Box \end{array}$$

Example 5.8 *With $\beta\Pi$-reduction, $(\Pi_{x:D}.B)C$ reduces to $B[x := C]$, hence for A of Example 5.7, $\tau(<>, A)$ reduces to $\Box$ and so does $\tau(<>, \mathcal{I}(A))$.*

It is easier to calculate the canonical type in item notation than in classical notation. In fact, in item notation, we go through A from left to right and for every main item s_i we reach, we keep it unchanged if it is a δ-item, we remove it if it is a Π-item and we change the λ to Π if it is a λ-item. Finally, we replace $\heartsuit(A)$ (let us say x) by $\tau(\Gamma', x)$ where $\Gamma' \equiv \Gamma s'_{i_1} \ldots s'_{i_k}$ and s'_{i_j} are all the main π-items of A where Π is changed to λ. In item notation, every term is of the form $\overline{s}x$ or $\overline{s}S$ where $\overline{s}$ is a segment, i.e. a sequence of items and $S \in \{*, \Box\}$. For a segment $\overline{s}$, we define $\overline{s}^\lambda$ as $\overline{s}$ where all the main π-items are written as λ-items and where all the main δ-items are removed. We define $\overline{s}^\Pi$ as $\overline{s}$ where all the main λ-items are replaced by Π-items, all the main δ-items remain unchanged and all the main Π-items are removed. For example, if $\overline{s} \equiv (x\delta)(y\lambda_z)(z\Pi_r)$ then $\overline{s}^\lambda \equiv (y\lambda_z)(z\lambda_r)$ and $\overline{s}^\Pi \equiv (x\delta)(y\Pi_z)$. With these notations, $\tau(\Gamma, \overline{s}x) \equiv \overline{s}^\Pi\tau(\Gamma\overline{s}^\lambda, x)$.

Hence, $\tau(\Gamma, A)$ is easy to construct out of A in item notation: just drop all the main Π-items, change the main λ-items into Π-items and make sure you alter your context accordingly. Finally make sure you replace the heart variable (which is very obvious in item notation) by its canonical type in your updated context.

As there has been many arguments in the literature for making substitutions explicit, one may also find arguments for making typing explicit. Hence, we can imagine that our items are not only δ and λ-items but may also be τ-items which find the type of a term. That is, for any term A, we have that $(A\tau)$ is an item. According to Remark 1.5 we may treat a context as a term and hence $(\Gamma\tau)$ is also an item. Now, look at how we can redefine τ of Definition 5.1 in a step-wise fashion:

Definition 5.9 *(Step-wise canonical typing)*

$$\begin{array}{llllll}
\textit{Propagation rules:} & (\Gamma\tau)(A\delta) & \rightarrow_\tau & (A\delta) & (\Gamma\tau) \\
 & (\Gamma\tau)(A\lambda_x) & \rightarrow_\tau & (A\Pi_x) & (\Gamma(A\lambda_x)\tau) \\
 & (\Gamma\tau)(A\Pi_x) & \rightarrow_\tau & & (\Gamma(A\lambda_x)\tau)
\end{array}$$

$$\begin{array}{lllll}
\textit{Destruction rules:} & (\Gamma\tau)* & \rightarrow_\tau & \Box & \\
 & (\Gamma\tau)x & \rightarrow_\tau & A & \textit{if } (A\lambda_x) \in \Gamma
\end{array}$$

Example 5.10
Let $\Gamma_0 \equiv <>$, $\Gamma_1 \equiv (*\lambda_z)$, $\Gamma_2 \equiv (*\lambda_z)(*\lambda_y)$, $\Gamma_3 \equiv \Gamma_2(*\lambda_x)$. *We want to find the canonical type of* $(*\Pi_z)(B\delta)(*\lambda_y)(y\delta)(*\lambda_x)x$ *in the empty context* $<>$.

$$\begin{array}{ccccccccccccc}
(\Gamma_0\tau) & (*\Pi_z) & & (B\delta) & & (*\lambda_y) & & (y\delta) & & (*\lambda_x) & & x & \rightarrow_\tau \\
 & & (\Gamma_1\tau) & (B\delta) & & (*\lambda_y) & & (y\delta) & & (*\lambda_x) & & x & \rightarrow_\tau \\
 & & & (B\delta) & (\Gamma_1\tau) & (*\lambda_y) & & (y\delta) & & (*\lambda_x) & & x & \rightarrow_\tau \\
 & & & (B\delta) & & (*\Pi_y) & (\Gamma_2\tau) & (y\delta) & & (*\lambda_x) & & x & \rightarrow_\tau \\
 & & & (B\delta) & & (*\Pi_y) & & (y\delta) & (\Gamma_2\tau) & (*\lambda_x) & & x & \rightarrow_\tau \\
 & & & (B\delta) & & (*\Pi_y) & & (y\delta) & & (*\Pi_x) & (\Gamma_3\tau) & x & \rightarrow_\tau \\
 & & & (B\delta) & & (*\Pi_y) & & (y\delta) & & (*\Pi_x) & & * &
\end{array}$$

Like this, we have made the τ-items first class citizens as we did with λ and δ-items and as we can do with any other notions of the lambda calculus (such as substitution, searching for the binding λ and so on). This illustrates the modularity of our notation. Furthermore, the step-wise definition of τ has a pattern that can be adapted by all the other concepts that we can define as first class citizens. We will always have *propagation* and *destruction* rules. Often we will also have *generation* rules which say how a certain item is generated. For example, a substitution item is generated by a $\delta\lambda$-segment as follows (see [9]):

$$(A\delta)(B\lambda_x) \rightarrow_\sigma (A\sigma_x)$$

Now that we have elaborated that finding the canonical type in item notation is clearer than in classical notation, let us reflect a bit on why canonical typing is useful. Basically the idea is that a judgement $\Gamma \vdash A : B$ says that A is typable and that B is one of its types. We find that this question could better be divided in two:

1. Is A typable?

2. Given B, is B one of the types of A?

It turns out that this division provides some simplification in the typing rules of Definition 1.6 and that $\tau(\Gamma, A)$ plays the role of a preference type of A. In fact, the conversion rule is no longer needed in Definition 1.6. In our opinion, the approach of the traditional framework is, in a sense, ambiguous in that for a variable x and a context Γ, there is a preference type for x; namely, the A where $(B\lambda_x) \in \Gamma$. For terms in general however, no such preference type is given, but a whole collection of types, which are typable themselves and linked by means of β-reduction.

Here are now the rules which replace $\vdash_\beta$ (note how conversion is removed):

Definition 5.11 *($\vdash$) The Typability relation $\vdash$ is defined by the following rules:*

($\vdash$-axiom) $\quad <> \vdash *$

($\vdash$-start rule) $\quad \dfrac{\Gamma \vdash A}{\Gamma(A\lambda_x) \vdash x}$ *if vc*

($\vdash$-weakening rule) $\quad \dfrac{\Gamma \vdash A \qquad \Gamma \vdash D}{\Gamma(A\lambda_x) \vdash D}$ *if vc*

($\vdash$-application rule) $\quad \dfrac{\Gamma \vdash F \qquad \Gamma \vdash a}{\Gamma \vdash (a\delta)F}$ *if ap*

($\vdash$-abstraction rule) $\quad \dfrac{\Gamma(A\lambda_x) \vdash b \qquad \Gamma \vdash (A\Pi_x)B}{\Gamma \vdash (A\lambda_x)b}$ *if ab*

($\vdash$-formation) $\quad \dfrac{\Gamma \vdash A \qquad \Gamma(A\lambda_x) \vdash B}{\Gamma \vdash (A\Pi_x)B}$ *if fc*

vc (variable condition): $x \notin \Gamma$ and $\tau(\Gamma, A) \twoheadrightarrow_{\beta\Pi} S$ for some S
ap (application condition): $\tau(\Gamma, F) =_{\beta\Pi} (A\Pi_x)B$ and $\tau(\Gamma, a) =_{\beta\Pi} A$ for some A, B.
ab (abstraction condition): $\tau(\Gamma(A\lambda_x), b) =_{\beta\Pi} B$ and $\tau(\Gamma, (A\Pi_x)B) \twoheadrightarrow_{\beta\Pi} S$ for some S.
fc (formation condition): $\tau(\Gamma, A) \twoheadrightarrow_{\beta\Pi} S_1$ and $\tau(\Gamma(A\lambda_x), B) \twoheadrightarrow_{\beta\Pi} S_2$ for some rule (S_1, S_2).

When $\Gamma \vdash A$, we say that A is typable in Γ.

Now, $\vdash_\beta$, $\vdash$ and τ are related by the following lemma:

Lemma 5.12 $\Gamma \vdash_\beta A : B \Longleftrightarrow \Gamma \vdash A \wedge \tau(\Gamma, A) =_{\beta\Pi} B \wedge B$ *is $\vdash_\beta$-legal type.* $\square$

The condition B is $\vdash_\beta$-legal type is necessary because if $\tau(\Gamma, A) =_{\beta\Pi} B$ and B has a Π-redex, then we can't derive $\Gamma \vdash_\beta A : B$. In fact, if $\Gamma \vdash_\beta A : B$ then neither A nor B have Π-redexes. For a study of the cube resulting from $\vdash$ and τ (but in classical notation) see [12].

6 The restriction of a term

In the present section we explain how to derive the restriction $t \upharpoonright x^\circ$ of a term t to a variable occurrence x° in t. This restriction is itself a term, consisting of precisely those "parts" of t that may be relevant for this x°, especially as regards binding, typing and substitution.

The restriction of a term t to a particular occurrence of a variable x° (denoted $t \upharpoonright x^\circ$) is defined to be the part of t which contains all the information relevant for x° in t. In particular,

- the type of x° in t is the type of x° in $t \upharpoonright x^\circ$,
- the λ's relevant to x° in t appear also in $t \upharpoonright x^\circ$ and have the same binding relation to x°,
- If in t, any substitution for x° is possible, then it is also possible in $t \upharpoonright x^\circ$.

In other words, $t \upharpoonright x^\circ$ is everything relevant to x° in t in terms of binding, typing and substitution. We show how easy it is to calculate $t \upharpoonright x^\circ$ in our calculus. Moreover, $t \upharpoonright x^\circ$ is calculated using a step-wise approach.

When a variable x occurs in term t, then it is not the case that all the "information" contained in t is necessarily relevant for a specific occurrence x° of x in t. The following example illustrates the point:

Example 6.1 In the term $t \equiv (*\lambda_x)(x\lambda_v)(x\delta)(*\lambda_y)((x\lambda_z)y^\circ\delta)(y\lambda_u)u$, only the items $(*\lambda_x)$, $(x\lambda_v)$, $(x\delta)$, $(*\lambda_y)$ and $(x\lambda_z)$ are of importance for the variable occurrence y°. y° is in the scope of $(*\lambda_x)$, $(x\lambda_v)$, $(*\lambda_y)$ and $(x\lambda_z)$. Moreover, the x is a candidate for substitution for y°, due to the presence of the $\delta\lambda$-segment $(x\delta)(*\lambda_y)$ meaning that the x will substitute y in $((x\lambda_z)y^\circ\delta)(y\lambda_u)u$. Hence $(x\delta)$ is also relevant for y°. Nothing else in t is relevant to y°. The term t in classical notation is written as: $\lambda_{x:*}.\lambda_{v:x}.(\lambda_{y:*}.(\lambda_{u:y}.u)\lambda_{z:x}.y^\circ)x$.

Now the restriction of a term t to a variable x is very easily found in our notation as we shall see below. In fact, look back at Example 6.1 and notice that all the relevant items to t, can be found *to the left* of y° in t. In fact, the term restriction will be: $(*\lambda_x)(x\lambda_v)(x\delta)(*\lambda_y)(x\lambda_z)$. That is: everything to the right of y° is cut out leaving $(*\lambda_x)(x\lambda_v)(x\delta)(*\lambda_y)((x\lambda_z)$. Then all extra parentheses are removed.

Example 6.2 In classical notation, t of Example 6.1 is: $\lambda_{x:*}.\lambda_{v:x}.(\lambda_{y:*}.(\lambda_{u:y}.u)\lambda_{z:x}.y^\circ)x$, the restriction of t to y° is less obvious. Compare how easily it could be calculated in our notation.

Now as we are interested in formalisation and implementation, we need to write a formal procedure to find $t\restriction x^\circ$. This is relatively easy:

Definition 6.3

$$x^\circ \restriction x^\circ \equiv x$$

$$(t_1\omega)t_2 \restriction x^\circ \equiv \begin{cases} t_1 \restriction x^\circ & \text{if } x^\circ \textit{ occurs in } t_1 \\ (t_1\omega)(t_2 \restriction x^\circ) & \text{if } x^\circ \textit{ occurs in } t_2 \end{cases}$$

Example 6.4 Let t be the following term:

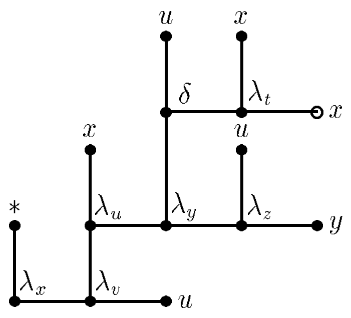
$$(*\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\lambda_v)u. \tag{1}$$

$$\begin{aligned} \text{Then } t\restriction x^\circ &\equiv ((*\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\lambda_v)u)\restriction x^\circ \\ &\equiv (*\lambda_x)(((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\lambda_v)u\restriction x^\circ) \\ &\equiv (*\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\restriction x^\circ) \\ &\equiv (*\lambda_x)(x\lambda_u)(((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\restriction x^\circ) \\ &\equiv (*\lambda_x)(x\lambda_u)((u\delta)(x\lambda_t)x^\circ\restriction x^\circ) \\ &\equiv (*\lambda_x)(x\lambda_u)(u\delta)((x\lambda_t)x^\circ\restriction x^\circ) \\ &\equiv (*\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)(x^\circ\restriction x^\circ) \\ &\equiv (*\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x \end{aligned}$$

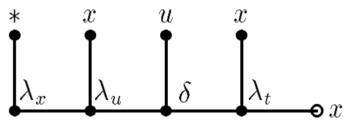
Now as said earlier, it is very easy to obtain the full restriction $t\restriction x^\circ$ using our item-notation: just take the substring of string t from the beginning of t until x° and delete all unmatched opening parentheses. This is an advantage of our new notation.

It is illustrative to draw the tree of t (see Figure 6) and to see what happens when the restriction process is executed with this tree. In Figure 6, the intended occurrence of x° in the trees is the rightmost one. One could describe the procedure as follows: Firstly, the part of the tree below the root path of x° is completely erased; secondly, all vertical branches in the same root path are contracted into single nodes. (Note of course that $t\restriction x^\circ \equiv \overline{s}x^\circ$.)

Intuitively, the body $\overline{s}x$ of $t\restriction x^\circ$ is the only thing that matters for x° in t; the rest of (the tree of) the term t may be neglected, as far as the x° is concerned. As said before, this is



$$t \equiv (*\lambda_x)((x\lambda_u)((u\delta)(x\lambda_t)x^\circ\lambda_y)(u\lambda_z)y\lambda_v)u \equiv (*\lambda)((1\lambda)((1\delta)(2\lambda)3^\circ\lambda)(2\lambda)4\lambda)3$$



$$t \upharpoonright x^\circ \equiv (*\lambda_x)(x\lambda_u)(u\delta)(x\lambda_t)x^\circ \equiv (*\lambda)(1\lambda)(1\delta)(2\lambda)3^\circ$$

Figure 6: A term and its restriction to a variable

essentially the importance of the restriction: $t \upharpoonright x$ is a term with x as its heart, that contains all "information" relevant for x. For example, when x is bound, then the bond between x and the λ binding this x does not change in the process of restriction. So the λ binding this x can be found in $t \upharpoonright x$; the same holds for the type of this x. Moreover, when x is a candidate for a substitution caused by a reduction, then the $\delta\lambda$-segment connected with this reduction can be found, again, in $t \upharpoonright x$.

Full restriction is, of course, idempotent; more generally, the following holds:

Lemma 6.5 *If y occurs in t, and x occurs in $t \upharpoonright y$, then $(t \upharpoonright y) \upharpoonright x \equiv t \upharpoonright x$.*

Proof: *By induction on t.* □

The described notion 'restriction of a term to a variable' has an obvious generalisation: 'restriction of a term to a *subterm*':

Definition 6.6 *(restriction of a term to a subterm)*
Let $\underline{t_0}$ be an occurrence of subterm t_0 in term t. Let $x^\circ \equiv \heartsuit(\underline{t_0})$. Then $t \upharpoonright \underline{t_0}$ is defined as $t \upharpoonright x^\circ$.

Note that a term $t \upharpoonright \underline{t_0}$ contains all "information" necessary for $\underline{t_0}$.

Now, to summarize this section, we introduced the notion of restriction of a term t to a variable occurrence x°, $t \upharpoonright x^\circ$. $t \upharpoonright x^\circ$ contains all the information relevant for x° in t. No other information in t is relevant for x°. In fact, the λ's relevant to x° in t, the type of x° in t and what terms might be substituted for x° in t, are all present in $t \upharpoonright x^\circ$. We showed that calculating $t \upharpoonright x^\circ$ is very simple in our formulation. Once we introduce the bound and free variables in the next section, we will get back to $t \upharpoonright x^\circ$, to prove that

- x° is free (resp. bound) in t iff x° is free (resp. bound) in $t \upharpoonright x^\circ$ and
- the type of x° in t is the type of x° in $t \upharpoonright x^\circ$.

7 Bound and free variables

An important notion in lambda calculus is that of bound and free variables; for a bound variable the "binding place" is relevant. Variables and their status are the subject of this section. Of course, for this study of variables to make sense, we shall (in this section only) not assume the Barendregt convention.

Calculating bound and free variables in a term, calculating the λ binding a particular variable occurrence and the variables bound by a particular λ are very important concepts in the λ-calculus. We show how easy it is to calculate the bound and free variables in our notation and how the variables bound by a λ and the λ binding a variable can be found by step-wise procedures. These step-wise procedures closely follow the usual implementation of these concepts. We just scan branches and nodes one by one.

Let us start by defining $\texttt{sieveseg}_\pi(t)$ to be the main π-items of t, written in the order in which they appear in t. For example, $\texttt{sieveseg}_\pi((a\lambda_x)(b\delta)(c\Pi_y)) = (a\lambda_x)(c\Pi_y)$. Let us also for an item $(A\omega)$, define A to be $\texttt{body}((A\omega))$ and ω to be $\texttt{endop}((A\omega))$.

Definition 7.1 *($IB(v,t)$, the item binding a variable)*
Let t be a term and let x° be a variable occurrence in t and assume that $\texttt{sieveseg}_\pi(t \upharpoonright x^\circ) \equiv s_m \ldots s_1$ (for convenience numbered downwards). $IB(x^\circ,t) = s_i$ for i being the smallest k in $\{1,2,\ldots,m\}$ such that $\texttt{endop}(s_k) = \lambda_x$.

We write $IB(x^\circ,t)\downarrow$ when $IB(x^\circ,t)$ is defined.

Example 7.2 In $t \equiv (x_5\lambda_{x_4})(x_1\lambda_{x_4})((x_2\lambda_{x_6})(x_4\delta)x_4^\circ\lambda_{x_5})x_1^\circ$, $IB(x_4^\circ,t) = (x_1\lambda_{x_4})$ whereas $IB(x_1^\circ,t)$ is undefined.

Definition 7.3 *(bound and free variables, type, open and closed terms)*
Let x° be a variable occurrence in a term t.

- *x° is* **bound** *in t if $IB(x^\circ,t)\downarrow$. In such a case,*
  - *The* **binding item** *of x° in t is $IB(x^\circ,t)$.*
  - *The operator that* **binds** *x° in t is $\texttt{endop}(IB(x^\circ,t))$.*
  - *The* **type** *of x° in t is $\texttt{body}(IB(x^\circ,t))$.*
- *x° is* **free** *in t if $IB(x^\circ,t)$ is not defined. In this case, the type of x° in t is undefined.*
- *Term t is* **closed** *when all occurrences of variables of $\mathcal{V}$ in t are bound in t. Otherwise t is* **open** *or* **has free variables***.*

Examples 7.4 and 7.5 below show that **it is easier** to account for free and bound variables and for the λ that binds a particular occurrence of a variable than in the classical notation.

Example 7.4 Let $t \equiv (*\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(*\lambda_{x_3})((x_1\lambda_{x_4})x_3^\circ\delta)(x_3\lambda_{x_5})x_5$.
t written in classical notation is $\lambda_{x_1:*}.\lambda_{x_2:x_1}.(\lambda_{x_3:*}.(\lambda_{x_5:x_3}.x_5)(\lambda_{x_4:x_1}.x_3^\circ))x_1$.
Now it is straightforward to find $t \upharpoonright x_3^\circ$ in item notation. Just take the substring to the left of x_3° and remove all unmatched parenthesis. This results in $(*\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(*\lambda_{x_3})(x_1\lambda_{x_4})x_3^\circ$.
Now if we follow Definition 7.3, we find that x_3° is bound in t, its binding item is $(*\lambda_{x_3})$ and its type is $*$.

The item notation moreover, enables one to clearly see the connection between variables and their binding λ's whereas in the classical notation the relation between a variable and its binding λ may not be obvious to the eye. The following example demonstrates the point:

Example 7.5 Consider the following term, which we have written in classical notation: $\lambda_{x_4:*}.\lambda_{x_2:x_4}.(\lambda_{x_2:*}.(\lambda_{x_3:x_2}.x_3)\lambda_{x_2:x_4}.x_3^\circ)x_2^\circ$. Now, the x_3° is free in the term, but the presence of λ_{x_3} might confuse us to this fact. Moreover, λ_{x_2} occurs three times so which is the one binding x_2°? In item notation this is: $(\lambda_{x_4})(x_4\lambda_{x_2})(x_2{}^\circ\delta)(\lambda_{x_2})((x_4\lambda_{x_2})x_3^\circ\delta)(x_2\lambda_{x_3})x_3$. This term shows clearly the λ_{x_2} binding x_2°, the type of x_2° and that x_3° is free.

Note that (one-step or more-step) restriction does not affect whether a variable occurrence is free or bound, as the following lemma shows:

Lemma 7.6
The following holds for a particular occurrence x° of a variable v in t:

- *x° is bound (resp. free) in t iff x° is bound (resp. free) in $t \upharpoonright x^\circ$.*
- *The type of x° in t is the type of x° in $t \upharpoonright x^\circ$.*
- *$IB(x^\circ, t) = IB(x^\circ, t \upharpoonright x^\circ)$.*

Proof: *By induction on t.* □

Hence, we can look in $t \upharpoonright x^\circ$ rather than in t for all the information relevant to x°.

There is a simple procedure for finding the variable occurrences bound by a certain λ in a term t. In the following definition, this procedure is given as a step-by-step search.

For this purpose, we temporarily extend the language with a special **search item** or ζ-item and with a new relation, $\rightarrow_\zeta$, between (extended) terms.

The search begins with the generation of a ζ-item, just behind the λ-item in question. Thereupon this ζ-item is pushed through all subterms of the term "in the scope of" the λ-item. The ζ-generation works as follows: a $(\zeta^{(v)})$ is generated out of $(t\lambda_v)$, Furthermore, $(\zeta^{(v)})(t\lambda_v)$ ζ-reduces to $((\zeta^{(v)})t\lambda_v)$ and not to $((\zeta^{(v)})t\lambda_v)(\zeta^{(v)})$ because all the variables to the right of $(t\lambda_v)$ are bound by the λ_v of $(t\lambda_v)$ and not by the original λ_v which generated the $(\zeta^{(v)})$. When ending at a variable v', the superscript v of the ζ-item decides whether v' is bound by the λ of the above-mentioned λ-item, or not. If this is the case, then the variable is capped with the symbol ^.

Definition 7.7 *(ζ-reduction)*
The ζ-reduction relation $\rightarrow_\zeta$ is the reduction relation *generated out of the following rules which relate segments and terms to other segments and terms.*

(ζ-generation rules:)

$$(t\lambda_v) \rightarrow_\zeta (t\lambda_v)(\zeta^{(v)})$$

(ζ-transition rules:)

$$\begin{array}{lll} (\zeta^{(v)})(t\lambda_v) & \rightarrow_\zeta & ((\zeta^{(v)})t\lambda_v) \\ (\zeta^{(v)})(t\lambda_{v'}) & \rightarrow_\zeta & ((\zeta^{(v)})t\lambda_{v'})(\zeta^{(v)}) \quad \textit{if } v \not\equiv v' \\ (\zeta^{(v)})(t\delta) & \rightarrow_\zeta & ((\zeta^{(v)})t\delta)(\zeta^{(v)}) \end{array}$$

(ζ-destruction rules:)

$$(\zeta^{(v)})v \rightarrow_\zeta \hat{v}$$
$$(\zeta^{(v)})v' \rightarrow_\zeta v' \ \textit{ if } v' \not\equiv v.$$

In order to prevent undesired effects, we only allow an application of the ζ-generation rule in a term t *when there is no other ζ-item present in t.* The undesired effects come from the fact that if we allow ζ to pass other ζ, then the cap that we obtain as a result of a ζ-destruction will not be clearly associated with the right λ.

Example 7.8 Let $t \equiv (\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})x_3\delta)(x_3\lambda_{x_3})x_3$. If we want to find all variables bound by the λ_{x_3} of (λ_{x_3}) in t, we can apply the following sequence of ζ-reductions:

$$\begin{array}{l}
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})x_3\delta)(x_3\lambda_{x_3})x_3 \rightarrow_\zeta \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})(\zeta^{(x_3)})((x_1\lambda_{x_4})x_3\delta)(x_3\lambda_{x_3})x_3 \rightarrow_\zeta \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((\zeta^{(x_3)})(x_1\lambda_{x_4})x_3\delta)(\zeta^{(x_3)})(x_3\lambda_{x_3})x_3 \rightarrow_\zeta \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})(((\zeta^{(x_3)})x_1\lambda_{x_4})(\zeta^{(x_3)})x_3\delta)(\zeta^{(x_3)})(x_3\lambda_{x_3})x_3 \twoheadrightarrow_\zeta \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})\hat{x}_3\delta)((\zeta^{(x_3)})x_3\lambda_{x_3})x_3 \rightarrow_\zeta \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})\hat{x}_3\delta)(\hat{x}_3\lambda_{x_3})x_3
\end{array}$$

Note that the last x_3 is not capped. The reason for this is that it is bound by the last λ_{x_3} of the term, instead of the λ_{x_3} we are interested in. Note furthermore, that if the Barendregt Convention is assumed then it is trivial to look for all the variables v bound by λ_v because every v in the term is bound by the λ_v. In other words, λ_v does not occur more than once in a term.

A similar procedure can be given for searching for the λ binding a certain occurrence v° of a variable v in a term t. For this purpose we introduce an **inverse search item** or $\zeta_\star$-item.

The inverse search item has to move in the opposite direction. A special provision has to be made for the case that the variable in question happens to be free; in that case the reverse search item becomes the initial item of the term, and must be destructed. This case is not provided for in the following definition:

Definition 7.9 *($\zeta_\star$-reduction)*
The $\zeta_\star$-reduction relation $\rightarrow_{\zeta_\star}$ is the reduction relation *generated out of the following rules which relate segments and terms to other segments and terms.*

($\zeta_\star$-generation rule:)

$$v^\circ \rightarrow_{\zeta_\star} (\zeta_\star^{(v)})v^\circ$$

($\zeta_\star$-transition rules:)

$$\begin{array}{l}
(t\lambda_{v'})(\zeta_\star^{(v)}) \rightarrow_{\zeta_\star} (\zeta_\star^{(v)})(t\lambda_{v'}) \textit{ if } v \not\equiv v' \\
(t\delta)(\zeta_\star^{(v)}) \rightarrow_{\zeta_\star} (\zeta_\star^{(v)})(t\delta) \\
((\zeta_\star^{(v)})t\omega) \rightarrow_{\zeta_\star} (\zeta_\star^{(v)})(t\omega)
\end{array}$$

($\zeta_\star$-destruction rules:)

$$(t\lambda_v)(\zeta_\star^{(v)}) \rightarrow_{\zeta_\star} (t\hat{\lambda}_v)$$

Example 7.10 If $t \equiv (\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})x_3^\circ\delta)(x_3\lambda_{x_3})x_3$ then the search for the λ binding x_3° can be given by the following sequence of $\zeta_\star$-reductions:

$$\begin{array}{l}
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})x_3^\circ\delta)(x_3\lambda_{x_3})x_3 \rightarrow_{\zeta_\star} \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((x_1\lambda_{x_4})(\zeta_\star^{(x_3^\circ)})x_3^\circ\delta)(x_3\lambda_{x_3})x_3 \rightarrow_{\zeta_\star} \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})((\zeta_\star^{(x_3^\circ)})(x_1\lambda_{x_4})x_3^\circ\delta)(x_3\lambda_{x_3})x_3 \rightarrow_{\zeta_\star} \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\lambda_{x_3})(\zeta_\star^{(x_3^\circ)})((x_1\lambda_{x_4})x_3^\circ\delta)(x_3\lambda_{x_3})x_3 \rightarrow_{\zeta_\star} \\
(\lambda_{x_1})(x_1\lambda_{x_2})(x_1\delta)(\hat{\lambda}_{x_3})((x_1\lambda_{x_4})x_3^\circ\delta)(x_3\lambda_{x_3})x_3
\end{array}$$

Note here that this term is written as $\lambda_{x_1:\varepsilon}.\lambda_{x_2:x_1}.(\lambda_{x_3:\varepsilon}.(\lambda_{x_3:x_3}.x_3)(\lambda_{x_4:x_1}.x_3^\circ))x_1$ in classical notation. In the latter notation it is not clear at first sight which one of the two λ_{x_3}'s which occur before x_3° is the binding λ. Such a confusion does not occur when t is written in item notation as there is only one λ_{x_3} before x_3°.

Note that the search for a binding λ is easier than the search for all variables bound by a certain λ. This is because the latter search follows only one path in the layered tree, in the direction of the root; the former search disperses a ζ-item over all branches of the subtree with this λ as its root.

8 Describing normal forms in a substitution calculus

Lambda calculi with explicit substitutions attempt to close the gap between the classical λ-calculus and concrete implementations. Recently, there has been various attempts at providing calculi of explicit substitution ([6], [7], [9], [13], [14]).

Most of the above mentioned work (except [9]), uses classical notation. [13] provided λs, a calculus of substitution à la de Bruijn, which remains as close as possible to the classical λ-calculus. Here is a descrition of λs (we assume familiarity with de Bruijn indices):

Definition 8.1 *The set of terms, noted* Λs *, of the* λs*-calculus is given as follows:*

$$\Lambda s ::= \mathbb{N} \mid \Lambda s \Lambda s \mid \lambda \Lambda s \mid \Lambda s\, \sigma^i \Lambda s \mid \varphi^i_k \Lambda s \qquad where \quad i \geq 1\,, \quad k \geq 0\,.$$

$\mathbb{N}$ *denotes the set of positive natural numbers. We take* a, b, c *to range over* Λs. *A term of the form* $a\,\sigma^i b$ *is called a* closure. *Furthermore, a term containing neither* σ*'s nor* φ*'s is called a* pure term. *The set of pure terms is denoted by* Λ.

Definition 8.2 *The* λs-calculus *is given by the following rewriting rules:*

$$\begin{array}{lllll}
\sigma\text{-}generation & (\lambda a)\, b & \longrightarrow & a\,\sigma^1 b \\
\sigma\text{-}\lambda\text{-}transition & (\lambda a)\,\sigma^i b & \longrightarrow & \lambda(a\,\sigma^{i+1} b) \\
\sigma\text{-}app\text{-}transition & (a_1\, a_2)\,\sigma^i b & \longrightarrow & (a_1\,\sigma^i b)\,(a_2\,\sigma^i b) \\
\sigma\text{-}destruction & \mathtt{n}\,\sigma^i b & \longrightarrow & \begin{cases} \mathtt{n-1} & if \;\; n > i \\ \varphi^i_0\, b & if \;\; n = i \\ \mathtt{n} & if \;\; n < i \end{cases} \\
\varphi\text{-}\lambda\text{-}transition & \varphi^i_k\,(\lambda a) & \longrightarrow & \lambda(\varphi^i_{k+1}\, a) \\
\varphi\text{-}app\text{-}transition & \varphi^i_k\,(a_1\, a_2) & \longrightarrow & (\varphi^i_k\, a_1)\,(\varphi^i_k\, a_2) \\
\varphi\text{-}destruction & \varphi^i_k\, \mathtt{n} & \longrightarrow & \begin{cases} \mathtt{n+i-1} & if \;\; n > k \\ \mathtt{n} & if \;\; n \leq k \end{cases}
\end{array}$$

We use λs *to denote this set of rules. The calculus of substitutions associated with the* λs*-calculus is the rewriting system whose rules are* $\lambda s - \{\sigma\text{-}generation\}$ *and we call it the s-calculus.*

[13] has shown that the s-normal forms of the λs-terms are exactly the pure terms in Λ. Furthermore, [14] studied the extension of λs with open terms (i.e. adding variable terms to the calculus). Extra rules were needed to guarantee the local confluence (see Definition 8.4). The syntax of the new terms and the new calculus are given in the following definitions:

Definition 8.3 *The set of open terms, noted Λs_{op} is given as follows:*

$$\Lambda s_{op} ::= \mathbf{V} \mid \mathbb{N} \mid \Lambda s_{op}\Lambda s_{op} \mid \lambda\Lambda s_{op} \mid \Lambda s_{op}\sigma\Lambda s_{op} \mid \varphi^i_k\Lambda s_{op} \qquad \textit{where} \quad i \geq 1\,, \quad k \geq 0$$

and where $\mathbf{V}$ *stands for a set of variables, over which X, Y, ... range. We take a, b, c to range over Λs_{op}. Furthermore,* closures *and* pure terms *are defined as for Λs.*

Definition 8.4 *The* λs_e-calculus *is obtained by adding the following rules to those of the λs-calculus given in Definition 8.1:*

$$\begin{array}{llcll}
\sigma\text{-}\sigma\text{-}\textit{transition} & (a\sigma b)\,\sigma^j\,c & \longrightarrow & (a\,\sigma^{j+1}\,c)\,\sigma\,(b\,\sigma^{j-i+1}\,c) & \textit{if} \quad i \leq j \\
\sigma\text{-}\varphi\text{-}\textit{transition 1} & (\varphi^i_k\,a)\,\sigma^j\,b & \longrightarrow & \varphi^{i-1}_k\,a & \textit{if} \quad k < j < k+i \\
\sigma\text{-}\varphi\text{-}\textit{transition 2} & (\varphi^i_k\,a)\,\sigma^j\,b & \longrightarrow & \varphi^i_k\,(a\,\sigma^{j-i+1}\,b) & \textit{if} \quad k+i \leq j \\
\varphi\text{-}\sigma\text{-}\textit{transition} & \varphi^i_k\,(a\,\sigma^j\,b) & \longrightarrow & (\varphi^i_{k+1}\,a)\,\sigma^j\,(\varphi^i_{k+1-j}\,b) & \textit{if} \quad j \leq k+1 \\
\varphi\text{-}\varphi\text{-}\textit{transition 1} & \varphi^i_k\,(\varphi^j_l\,a) & \longrightarrow & \varphi^j_l\,(\varphi^i_{k+1-j}\,a) & \textit{if} \quad l+j \leq k \\
\varphi\text{-}\varphi\text{-}\textit{transition 2} & \varphi^i_k\,(\varphi^j_l\,a) & \longrightarrow & \varphi^{j+i-1}_l\,a & \textit{if} \quad l \leq k < l+j
\end{array}$$

We use λs_e to denote this set of rules. The calculus of substitutions associated with the λs_e-calculus is the rewriting system whose rules are $\lambda s_e - \{\sigma\text{-generation}\}$ and we call it s_e-calculus.

[14] has shown that it is more cumbersome to describe the s_e-normal forms of the open terms. This description however is needed to establish the weak normalisation of the s_e-calculus. Here is how these normal forms are described in classical notation.

Theorem 8.5 *A term $a \in \Lambda s_{op}$ is an s_e-normal form iff one of the following holds:*

- *$a \in \mathbf{V} \cup \mathbb{N}$, i.e. a is a variable or a de Bruijn number.*
- *$a = b\,c$, where b and c are s_e-normal forms.*
- *$a = \lambda b$, where b is an s_e-normal form.*
- *$a = b\,\sigma^j c$, where c is an s_e-nf and b is an s_e-nf of the form X, or $d\,\sigma^i e$ with $j < i$, or $\varphi^i_k d$ with $j \leq k$.*
- *$a = \varphi^i_k b$, where b is an s_e-nf of the form X, or $c\,\sigma^j d$ with $j > k+1$, or $\varphi^j_l c$ with $k < l$.*

Proof: *Proceed by analising the structure of a. When a is an application or an abstraction there are no restrictions since there are no s_e-rules with applications or abstractions at the root. When $a = b\,\sigma^j c$ or $a = \varphi^i_k b$, the restrictions on b are necessary to avoid σ-redexes (rules whose name begin with σ) or φ-redexes (rules whose name begin with φ), respectively.* $\square$

There is a simple way to describe the s_e-nf's using item notation. Let us just say here that with this notation we have $a\,\sigma^i b = (b\,\sigma^i)a$ and $\varphi^i_k a = (\varphi^i_k)a$. $(c\,\sigma^i)$ and (φ^i_k) are called σ- and φ-items respectively. b and c are the *bodies* of these respective items.

A *normal $\sigma\varphi$-segment* $\overline{s}$ is a sequence of σ- and φ-items such that every pair of adjacent items in $\overline{s}$ are of the form:

$$(\varphi^i_k)(\varphi^j_l) \text{ and } k < l \qquad (\varphi^i_k)(b\,\sigma^j) \text{ and } k < j-1 \qquad (b\,\sigma^i)(c\,\sigma^j) \text{ and } i < j \qquad (b\,\sigma^j)(\varphi^i_k) \text{ and } j \leq k.$$

For example, $(\varphi^2_3)(\varphi^1_4)(\varphi^6_7)(b\sigma^9)(c\sigma^{11})(\varphi^2_{11})(\varphi^5_{16})$ and $(b\sigma^1)(c\sigma^3)(d\sigma^4)(\varphi^2_5)(\varphi^1_6)(\varphi^4_7)(a\sigma^{10})$ are normal $\sigma\varphi$-segments.

Here is the theorem (taken from [14]) which describes the s_e-nf's in a simple way:

Theorem 8.6 *The s_e-nf's can be described by the following syntax:*

$$NF ::= \mathbf{V} \mid \mathrm{I\!N} \mid (NF\,\delta)NF \mid (\lambda)NF \mid \overline{s}\,\mathbf{V}$$

where $\overline{s}$ is a normal $\sigma\varphi$-segment whose bodies belong to NF.

Proof: *It is easy to see that these are in fact normal forms since the conditions on the inidices of a normal $\sigma\varphi$-segment prevent the existence of redexes. To check that if a term is an s_e-nf then it is generated by this grammar, use Theorem 8.5.* $\square$

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy, "Explicit Substitutions", *Functional Programming 1 (4)* (1991) 375-416.

[2] H. Barendregt, Lambda calculi with types, *Handbook of Logic in Computer Science*, volume II, eds. Abramsky S., Gabbay D.M., and Maibaum T.S.E., Oxford University Press, 118-414, 1992.

[3] R. Bloo, F. Kamareddine and R. Nederpelt, The Barendregt Cube with Definitions and Generalised Reduction, submitted for publication.

[4] N.G. de Bruijn, A survey of the project AUTOMATH, in: J.R. Hindley and J.P. Seldin, ed., *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, (Academic Press, New York/London, 1980) 29-61.

[5] N.G. de Bruijn, Generalizing Automath by means of a lambda-typed lambda calculus, in: D.W. Kueker, E.G.K. Lopez-Escobar and C.H. Smith, ed., *Mathematical Logic and Theoretical Computer Science*, Lecture Notes in Pure and Applied Mathematics, 106, (Marcel Dekker, New York, 1978) 71-92.

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

[7] P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman, 1986. Revised edition: Birkhäuser, 1993.

[8] Th. Hardin, and J.-J. Lévy, "A confluent calculus of substitutions", Lecture notes of the INRIA-ICOT symposium, Izu, Japan, November (1989).

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

[10] F. Kamareddine and R. Nederpelt, A unified approach to type theory through a refined λ-calculus, *Theoretical Computer Science* 136, 183-216, 1994.

[11] F. Kamareddine and R. Nederpelt, Generalising reduction in λ-calculus, *Functional Programming 5 (4)*, 1995.

[12] F. Kamareddine and R. Nederpelt, Canonical Typing and π-conversion in the Barendregt Cube, *Functional Programming 6*, 1996.

[13] F. Kamareddine and A. Rios, λ-calculus à la de Bruijn with explicit substitution, to appear in the proceedings of PLILP '95, LNCS, Springer-Verlag.

[14] F. Kamareddine and A. Rios, The λ_s-calculus: its typed and its extended versions, submitted for publication.

[15] B.-J. de Leuw, Generalisations in the λ-calculus and its type theory, Master Thesis, Computing Science, University of Glasgow, 1995.

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