Math. Struct. in Comp. Science (2002), *vol.* 12, *pp.* 797–822. © 2002 Cambridge University Press
DOI: 10.1017/S0960129502003754 Printed in the United Kingdom

Encoding left reduction in the λ-calculus with interaction nets

SYLVAIN LIPPI

Institut de Mathématiques de Luminy, UPR 9016 du CNRS,
163 avenue de Luminy, case 930, 13288 MARSEILLE CEDEX 9, France
Email: lippi@iml.univ-mrs.fr

Received 15 February 2001; revised 2 January 2002

This paper presents a simple implementation of the λ-calculus in the interaction net paradigm. It is based on a two-fold translation. λ-terms are *coded* (for duplication) or *decoded* (for execution), and reduction is achieved by switching between these two states: decoding corresponds to head reduction and encoding to left reduction.

1. Introduction

We present a simple translation of the λ-calculus into the interaction net paradigm. There are already a number of other translations: such as the one into proof-nets with boxes of linear logic from Girard (1987), the elimination of boxes due to Abramsky, which has been studied in detail in Mackie (1994) or the optimal reduction from Gonthier *et al* (1992) inspired by the algorithm given in Lamping (1990). More recently, Lang has given another translation into the interaction net paradigm (Lang 1998), where, as in our translation, two different symbols are introduced for the application: one for the execution and one for the duplication.

Here, we do not need any indices for the duplication and the burden of the so-called 'bureaucracy' is drastically reduced. Morevover, there is no need for a readback algorithm that recovers a λ-term from a net. In practice, it is probably easier to read the net than the textual representation. On the other hand, we do not capture optimal reduction since redexes can be duplicated. So, this translation can be considered as a graphical syntax for the λ-calculus.

A complete graphical implementation of a λ-calculus interpreter can be found in Lippi (2001). It is composed of an interpreter for the interaction nets, and a source file including the interaction rules that implement a parser and a normaliser for the λ-calculus.

Notation: Throughout this paper, given any binary relation $\rightarrow$, its reflexive and transitive closure is denoted by $\stackrel{*}{\rightarrow}$. λ-terms (denoted by $u, v, w, t, t_1, t_2, \ldots$) are written with the Krivine notation: $x, y, z, x_1, x_2, \ldots$ are variables, $(u)v$ is the application of u to v, and $\lambda x\, u$ is a λ-abstraction.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

2. The interaction net paradigm

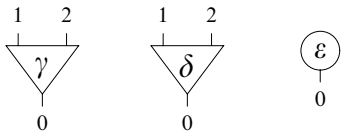
In this section, we present the interaction net paradigm in a very practical way by introducing a simple example: the interaction combinators system (Lafont 1997). We shall also give some useful properties of this system. For a more detailed presentation of interaction nets, see Lafont (1995).

2.1. *Basic notions*

2.1.1. *Syntax*

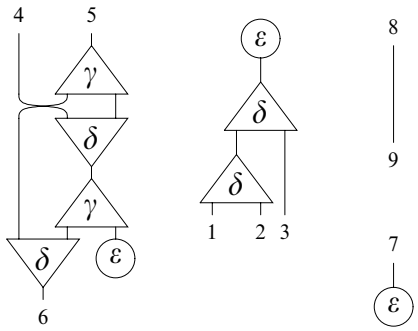
— The basic ingredient of an interaction system is a *symbol* with its *arity*. Our system has three symbols: γ and δ of arity 2, and ε of arity 0.

— Occurrences of symbols are called *cells*. Each cell has one *principal port* together with a number of *auxiliary ports* given by the arity of its symbol:



γ-cells and δ-cells have three *ports*: one *principal port* (0) and two *auxiliary ports* (1 and 2). ε-cells have only one (principal) port. For aesthetic reasons, cells with no auxiliary port are shown as a circle.

— A *net* is built with cells and *free ports*. Ports (principal ports, auxiliary ports or free ports) are connected pairwise by *wires*.

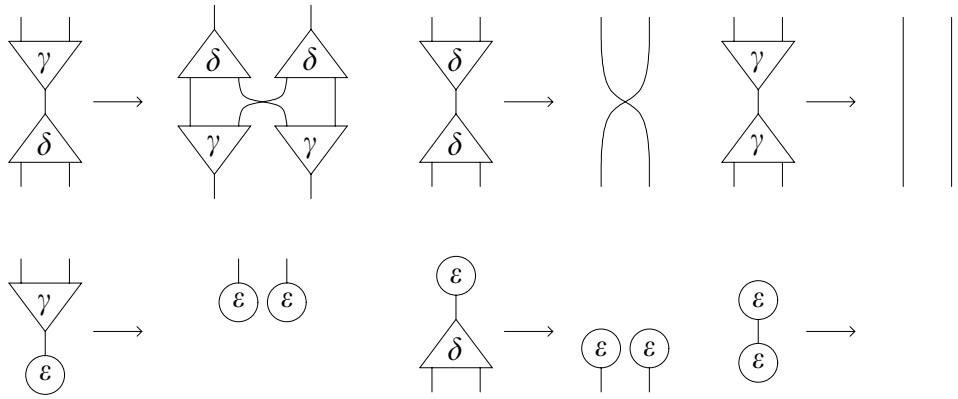


This net is built from two γ-cells, four δ-cells, three ε-cells and nine free ports.

— A *rule* is a pair of nets (*left member* $\rightarrow$ *right member*) with the same number of free ports. The important restriction is that the left member must be built from two cells connected by their principal ports[†]. An interaction system has, at most, one rule for each pair of symbols.

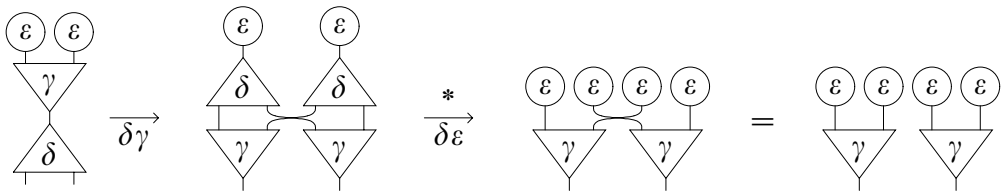
[†] If the two cells of the left member have the same symbol, there is also a symmetry condition on the right member (Lafont 1997).

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

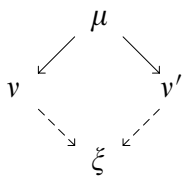


Note that the $\delta\varepsilon$ and $\gamma\varepsilon$ rules are the same except that the symbols γ and δ are exchanged.

2.1.2. *Execution* Once a set of symbols and rules has been fixed, we can apply one of the rules to a net to obtain another net, and so on until we have reached one that is irreducible. The *reduction* relation is denoted $\longrightarrow$. The following is an example where a net built from one γ-cell and two ε-cells is duplicated by a δ-cell:



Proposition 1 (Strong confluence). If a net μ reduces in one-step to ν and ν', with $\nu \neq \nu'$, then ν and ν' reduce in one-step to a common net ξ.



Proof. The two instances of the left members are necessarily disjoint, so we can apply the corresponding rules independently. □

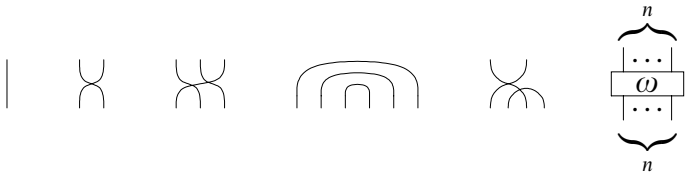
Corollary 1 (Reduction). If a net μ reduces to an irreducible net ν in n steps, then any reduction starting from μ eventually reaches ν in n steps.

2.2. *Taxonomy of nets*

Here are the important classes of nets that we shall meet later in this paper (each definition is accompanied by examples):

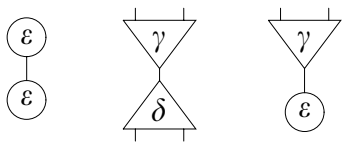
https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

— The *empty net* is built with no cell and no free port (so, there is no wire either).
— A *wiring* is constructed from free ports and wires only; there is no cell in a wiring.

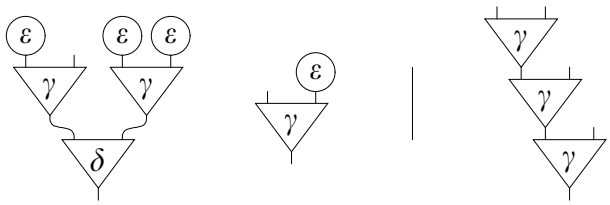


The last picture represents a wiring that corresponds to a *permutation* ω of $\{1, ..., n\}$. Free ports in the upper part are connected pairwise to free ports in the lower part.

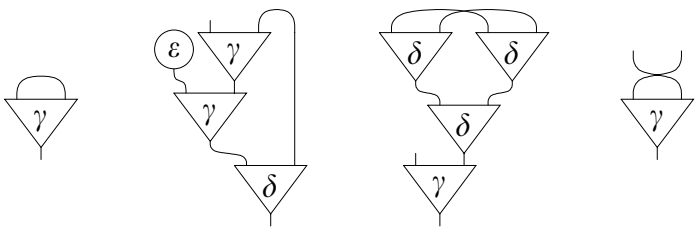
— A *cut* is constructed from two cells connected by their principal ports (so, the left member of a rule is a cut).



— A *tree* has one distinguished free port, called the *root*: it is either a single wire, in this case the root is fixed arbitrarily, or it is obtained by plugging the auxiliary ports of a cell into the roots of smaller trees.

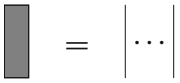


— A *principal net* is obtained by plugging the auxiliary ports of a tree with a wiring; so, a tree is also a principal net. Wires that connect two auxiliary ports are called *axiom links*, and principal nets with no axiom link are called *generalised trees*. Note that there is no cut in a principal net.



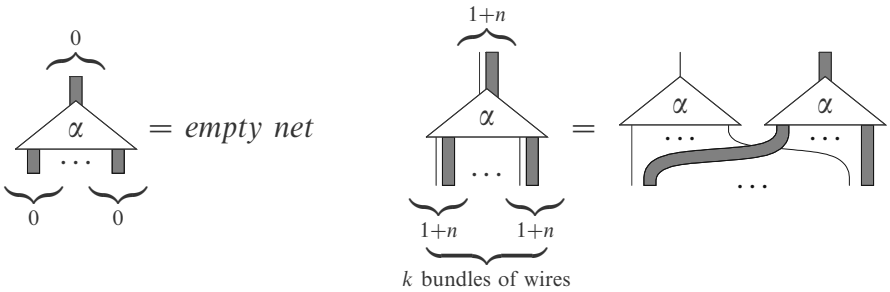
2.3. *Some properties of the system*

We shall use the following notation for a *bundle of wires*:

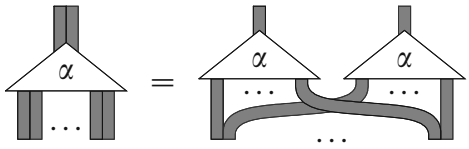


https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

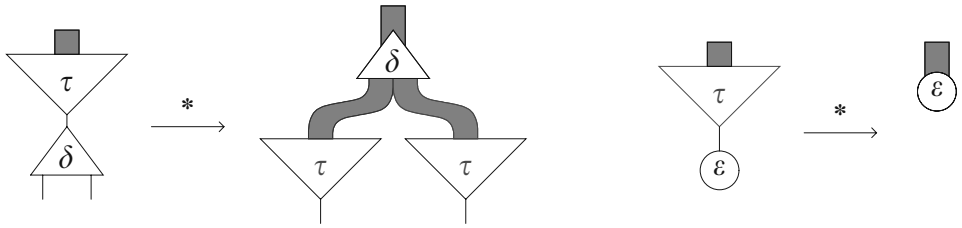
Moreover, for every symbol α of arity k, we define a *bundle of n α-cells* by induction on n as follows[†]:



Obviously, a bundle of a single α-cell ($n = 1$) corresponds exactly to the net built from one α-cell and $k + 1$ free ports connected to the ports of the α-cell. Moreover, a bundle can be decomposed easily as follows:



Lemma 1 (Duplication/erasing of trees). For any tree τ without a δ-cell:



Proof. The proof is by induction on τ using $\delta\gamma$ and $\delta\varepsilon$ rules for the first statement, $\varepsilon\gamma$ and $\varepsilon\varepsilon$ for the second one. □

Remark 1. The requirement that τ does not contain any δ-cell applies to duplication only, and not to erasing; indeed, an ε-cell can erase a tree with ε-cells in it.

Lemma 2 (Duplication/erasing of axiom links).

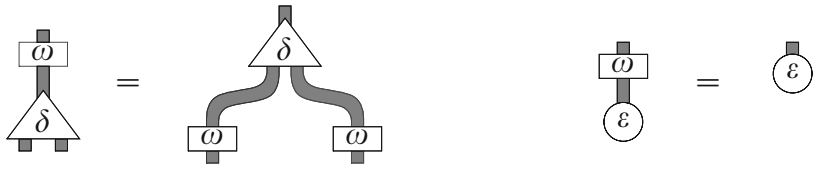


Proof. The proof is by induction on the number of wires using $\delta\delta$ and $\varepsilon\varepsilon$. □

[†] The permutation connected to the auxiliary ports of the α-cells is called a *perfect shuffle* or a *butterfly net*.

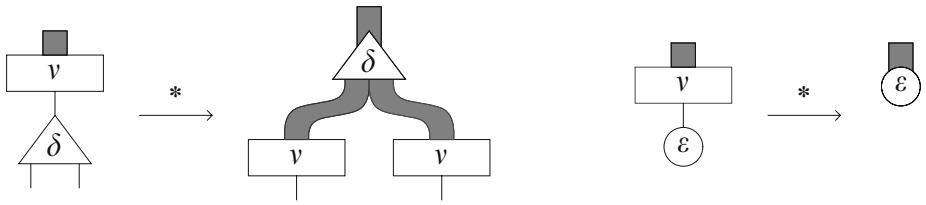
https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

Lemma 3 (Duplication/erasing of permutations). For any permutation ω:

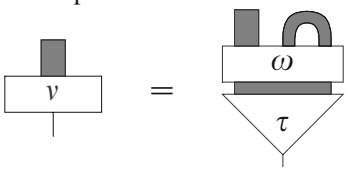


Proof. The proof is by induction on the number of wires. □

Proposition 2 (Duplication/erasing of principal nets). For any principal net v without δ-cell:



Proof. The proof is by Lemmas 1 (duplication/erasing of trees), 2 (duplication/erasing of axioms links) and 3 (duplication/erasing of permutations), since a principal net v can be decomposed as follows:



where τ is a tree and ω a permutation. □

3. Implementation

This section is the heart of the paper, and is devoted entirely to the description of an interaction system that normalises λ-terms. In other words, we translate a λ-term into an interaction net that is reduced until the translation of the normal form is reached.

3.1. *Coded and decoded translations*

We present two translations of λ-terms into interaction nets: a *coded translation* and a *decoded* one. As we saw in Section 2, we are able to write an interaction system in which principal nets can be duplicated or erased. During β-reduction, some parts of a λ-term are duplicated or erased; those parts are translated into principal nets called *coded translations*. On the other hand, there is no cut in a principal net: no interaction can occur in a coded translation. This lead us to introduce a *decoded translation* in which redexes occurring on the left branch of the λ-term correspond to cuts so that it reduces to the translation of the head normal form. The coded and decoded translations of a λ-term t are represented by a rectangle and an oval, respectively, like this:

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

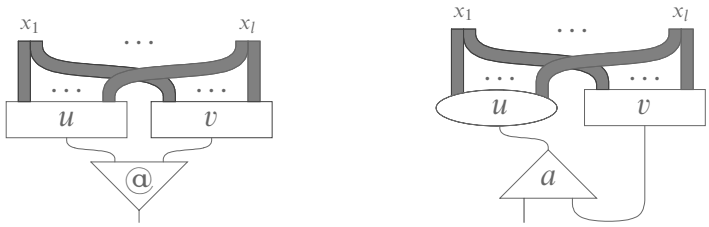


The unique free port at the bottom corresponds to the root of the λ-term, and each free port at the top corresponds to an occurrence of a free variable. Those free ports are grouped into bundles corresponding to variable names. We now define the two translations of a λ-term in parallel. We shall write coded (respectively, decoded) λ-term as shorthand for coded (respectively, decoded) translation of a λ-term.

— A coded variable is a single wire and a decoded variable is a unary cell called a *decoder*. So, we introduce a unary symbol ∘ (decoders are simply represented by a white cell without any symbol). This difference is due to the fact that variables are substituted by coded λ-terms. Consequently, decoded variables are marked by a decoder that transforms a coded λ-term into a decoded one.



— Two binary symbols are introduced for the application: @ (coded application or *application*) and *a* (decoded application or *applicator*). Let us remember that a coded translation is a principal net, so a coded application is obtained by plugging the coded function and argument into the auxiliary ports of @.
In the decoded case, an interaction can occur between the applicator and the decoded function. Moreover, if there is an interaction, the argument is likely to be duplicated or erased. Consequently, the decoded function is connected to the principal port of the applicator and the coded argument to the auxiliary port.



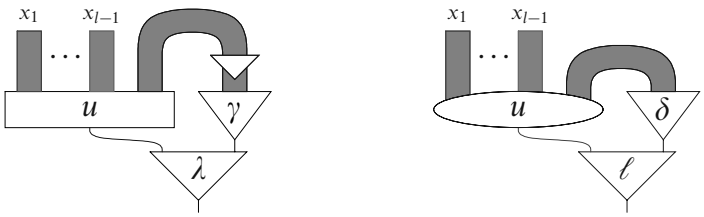
— There are several ways to tackle the problem of substitution without capturing free variables. For instance, De-Bruijn indices in the Krivine machine (Krivine 1985). Here we take advantage of the graphical representation: occurrences of bound variables are directly plugged into their corresponding linker. As previously, let us introduce two binary symbols for the abstraction: λ (coded abstraction or *abstraction*) and ℓ (decoded abstraction or *abstractor*). The basic idea for translating $\lambda x\, u$ is the same in the two cases: the coded (respectively, decoded) translation of *u* is plugged into the left auxiliary port of the abstraction (respectively, abstractor) and free occurrences of *x* are plugged into the right auxiliary port.

Remark 2. The role of the auxiliary ports is permuted from the usual practice; this is just to avoid complicated wirings during the reduction.

There may be more than one free occurrence of *x*, so we need two more binary

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

symbols to construct a generalised tree and 'gather' all free occurrences of x: γ (coded duplication or *sharing*) and δ (decoded duplication or *duplicator*). Intuitively, γ-cells represent duplication and δ-cells perform duplication. We also introduce the symbol ε (*eraser*) of arity 0 in case there is no free occurrence of x. The geometry of the coded abstraction is a bit different from the decoded one because occurrences of x are marked by the dual of the decoder (for simplicity, we use the same symbol $\circ$ for the decoder and its dual). We shall see the technical role of those cells in the decoding rules.

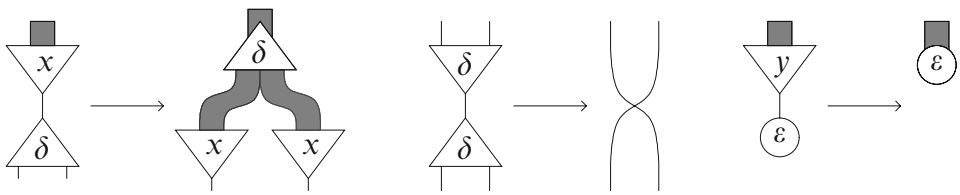


Γ is a *generalised sharing*, that is, a generalised tree built with sharings and erasers, and Δ is a *generalised duplicator*, that is, a generalised tree built with duplicators and erasers.

The complete translation is summed up in Figure 1, and examples of translations and generalised duplicators can be found in Figures 2 and 3.

3.2. *Duplication and erasing*

In Section 2, we presented a simple system in which principal nets built from γ- and ε-cells can be duplicated (respectively, erased) using δ-cells (respectively, ε-cells). Since coded translations are principal nets built from @-, λ-, γ-, ε- and $\circ$-cells, we can use the same principles and define duplication and erasing using the following schemes of rules:



where $x, y \in \{@, \lambda, \gamma, \varepsilon, \circ\}$.

Remark 3. Duplication is more complicated than erasing since not all the δ-rules follow the same scheme[†]. This explains the extra condition for duplication in proposition 2.

3.3. *Decoding: head reduction*

Remember that decoded λ-terms reduce to their decoded head normal form.

[†] In combinator terminology (Lafont 1997), we would say that the δx-rules are commutation rules whereas the $\delta\delta$-rule is an annihilation rule.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

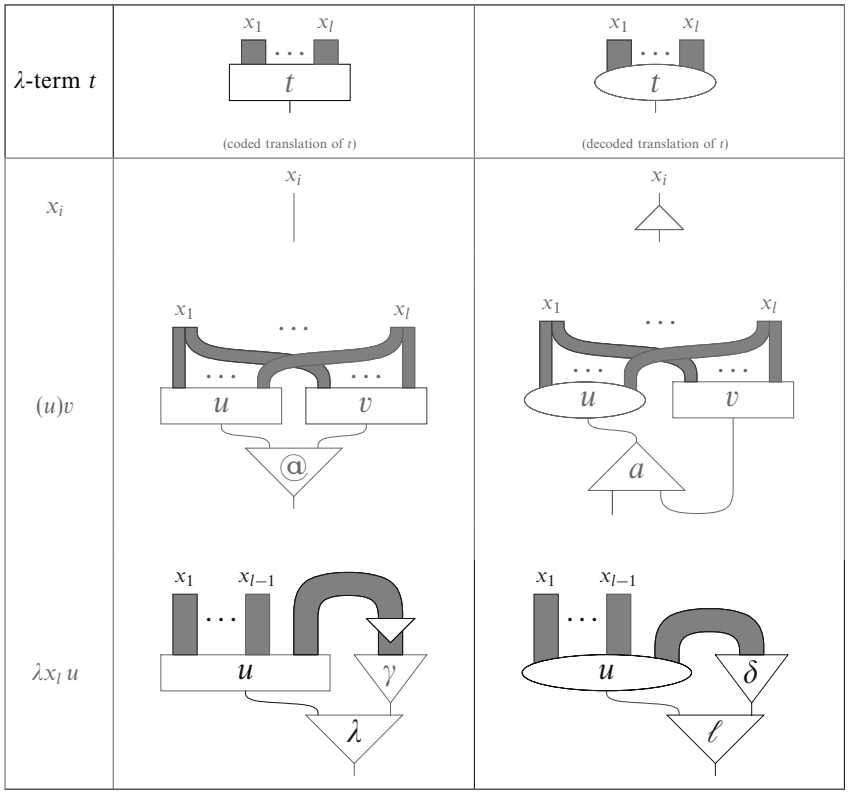


Fig. 1. *Coded and decoded translations*

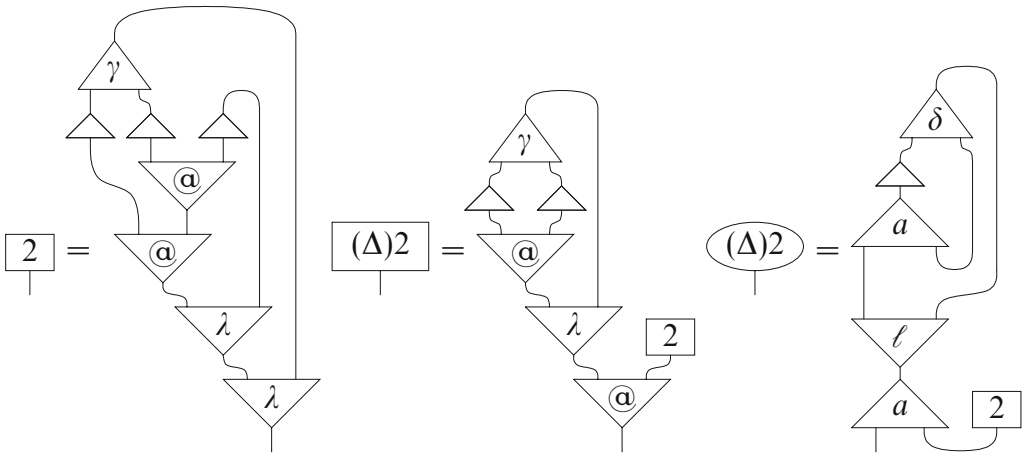


Fig. 2. *Examples of coded and decoded translations*

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

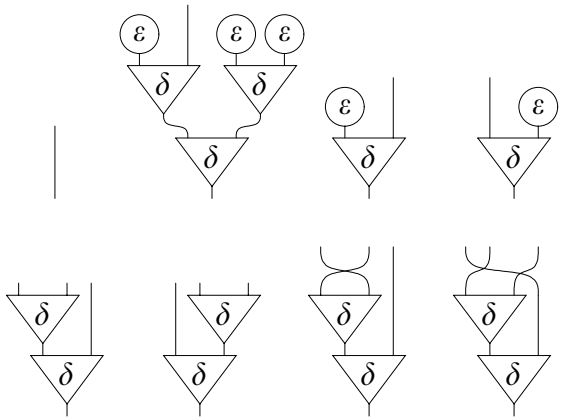
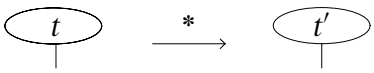
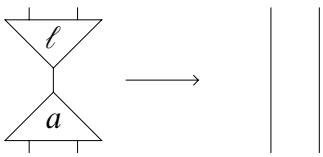


Fig. 3. *Some generalised duplicators*

Theorem 1 (Homogeneous head reduction). For each pair of closed λ-terms t and t', where t' is the head normal form of t:

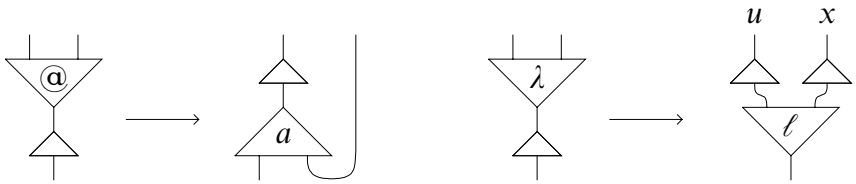


In a decoded λ-term we can only have an interaction between an applicator (a) and an abstractor (ℓ). Such cuts correspond to redexes in a λ-term. They are reduced by connecting the (coded) argument to the linker part of the (decoded) abstraction and the (decoded) body of the abstraction to the result of the (decoded) application. Formally, β-reduction is implemented by the ℓa-rule.

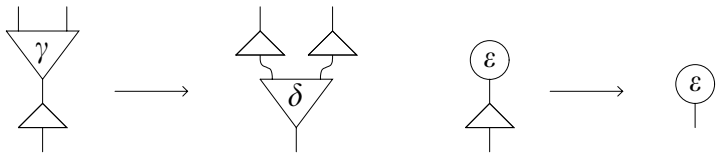


During the reduction, a coded λ-term may interact with a decoder: this happens when a decoded variable is substituted by a (coded) λ-term. In fact, unlike the dual of the decoder, which may appear several times in a coded translation, the decoder appears only once and corresponds to the leftmost occurrence of a variable. It enables us to keep on reducing until we have reached a head normal form. Formally, the decoder is 'moving' on the left branch of the λ-term towards the head variable. Consequently, we need five rules ($\circ@$, $\circ\lambda$, $\circ\gamma$, $\circ\varepsilon$ and $\circ\circ$) that implement the decoding of a coded λ-term. The decoding of an application and of an abstraction clearly follows the definition of the translation. Note that we only need to decode the function part of an application since the applicator (a) is connected to the coded argument.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

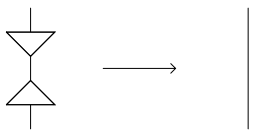


The set of decoding rules is not complete yet since the linker part of an abstraction must be decoded too. In other words, the decoder transforms a generalised sharing into a generalised duplicator.



Remark 4. The rule $\circ\varepsilon$ can be interpreted as decoding an eraser or erasing the dual of a decoder; this is due to the fact that we have chosen the same symbol for the decoder and its dual, and for the coded and decoded eraser.

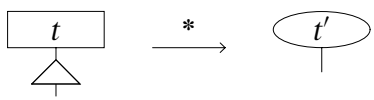
Finally, decoders are annihilated by their duals when a generalised sharing has been completely decoded.



This is why occurrences of variables are marked with the dual decoder when they are linked by a coded abstraction. The complete set of decoding rules is summed up in Figure 4, and an example of reduction is given in Figure 5.

To conclude, we give a corollary where the reduction starts from a coded translation; this allows us a free choice of whether to use either the coded or decoded translation as a graphical syntax for the λ-calculus.

Corollary 2 (Non-homogeneous head reduction). For each pair of closed λ-terms t and t', where t' is the head normal form of t:



3.4. *Encoding: left reduction*

The implementation of left reduction is based on the following remark: for each λ-term t that is not in head normal form, the head redex is the leftmost one. Consequently, to perform left reduction, we begin with head reduction. We saw in Section 3.3 that a

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

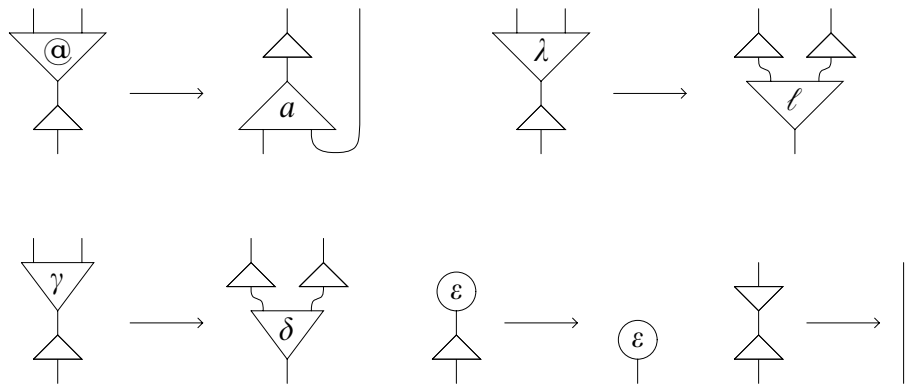


Fig. 4. *Decoding rules*

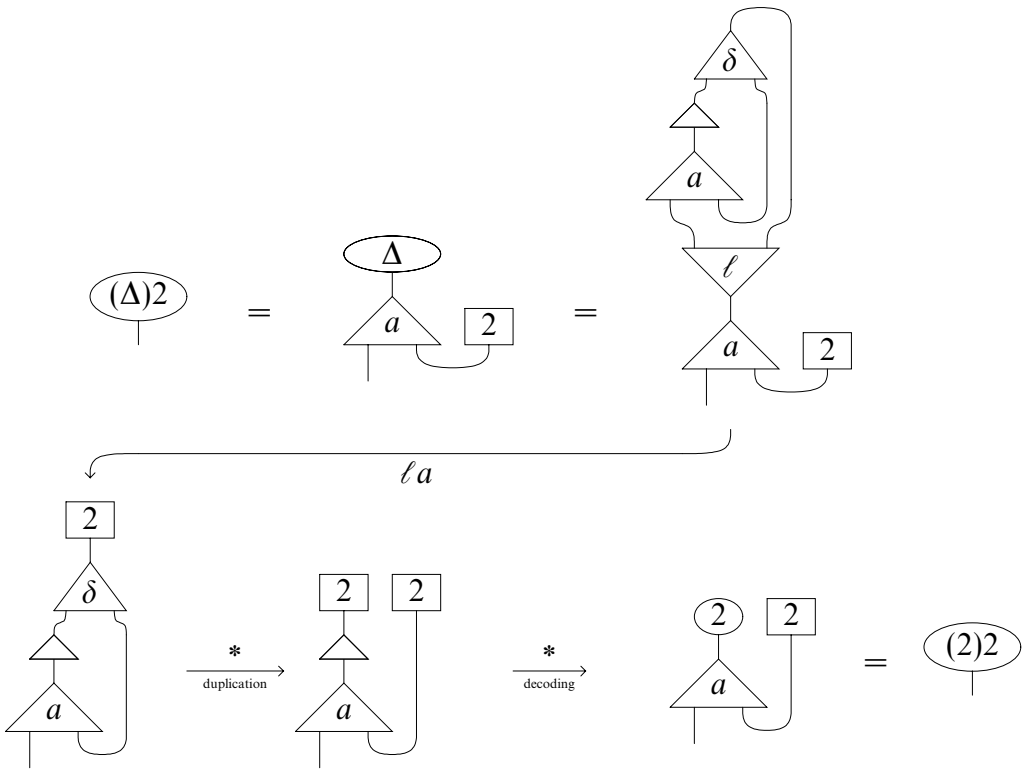


Fig. 5. *One-step head reduction of* (Δ)2

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

decoded λ-term reduces to its decoded head normal form. So the question now is how to obtain the coded normal form starting from the decoded head normal form?

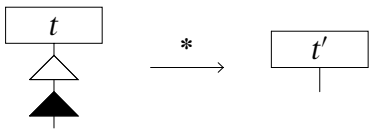
Decoding corresponds to head reduction; similarly, encoding corresponds to left reduction. However, encoding is a bit different since the reduction cannot start directly from a coded λ-term; there is no cut in a coded λ-term. Consequently, we introduce in the following theorem a unary symbol • (•-cells are represented by a black cell without any symbol) called an *encoder* that transforms a decoded λ-term into its coded normal form.

Theorem 2 (Non-homogeneous left reduction). For each pair of closed λ-terms t and t', where t' is the normal form of t:

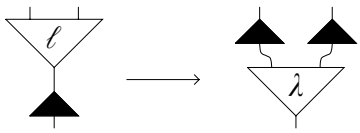


As with head reduction, there is also a homogeneous version of Theorem 2.

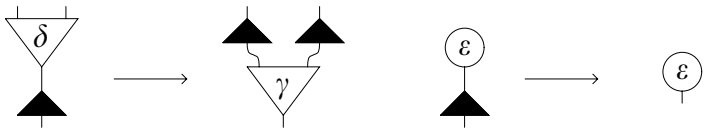
Corollary 3 (Homogeneous left reduction). For each pair of closed λ-terms t and t', where t' is the normal form of t:



Let us begin with the encoding of an abstraction. This comes directly from the definition of the translations:



With respect to the definition of a coded translation, both the body and the linker part of an abstraction must be encoded. In particular, the encoder transforms a generalised duplicator into a generalised sharing:



A closed head normal form t can be written as follows:

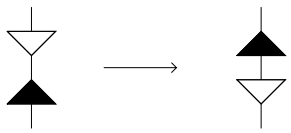
$$t = \lambda x_1 \ldots \lambda x_l\,(x_i)u_1 \ldots u_n,$$

where u_1, ..., u_n are λ-terms and x_i is the head variable.

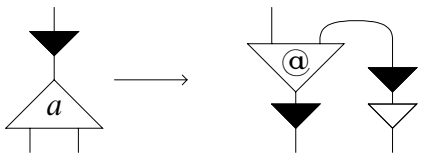
Starting from t, head reduction must be iterated on the arguments $u_1, ..., u_n$ connected on the left branch of t. Then, we must continue on their respective head normal form... and this process terminates on variables. Intuitively, the encoder is 'moving' on the left branch of t replacing each applicator by an application and iterating decoding (that is, head reduction) on their arguments.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

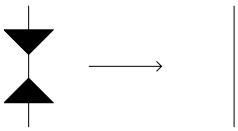
Unlike the decoder, the encoder cannot directly interact with the applicator since it is connected to its auxiliary port. However, the encoder interacts with the decoder. The ○●-rule corresponds to the encoding of a (linked) variable: the decoder is replaced by its dual and the encoding process continue with the encoder dual. Again, we use the same symbol for the encoder and its dual:



Now, we can implement the encoding of an application since the applicator interacts with the encoder dual. The applicator (a) is replaced by a correctly oriented application (@), and decoding is iterated on the coded argument. Finally, with respect to the definition of the coded translation, the argument must be re-encoded.



In this way, the coded arguments $u_1, ..., u_n$ connected to the applicators on the left branch of t are decoded and re-encoded. This process of decoding-encoding terminates when the encoder interacts with its dual:



The complete set of encoding rules is summed up in Figure 6 and the encoding of $\lambda x\,(x)x$ is given in Figure 7.

3.5. *Discussion*

Section 2 described a system where principal nets can be duplicated or erased. But duplication requires an extra condition: duplicated nets must contain no duplicator. This condition forces us to introduce two distinct symbols (γ and δ) for duplication, whereas a single symbol (ε) is enough for erasing.

For a given arity, there are several generalised duplicators and sharings of this arity; in Figure 3, the generalised duplicators of the upper part are all unary and those of the lower part are all ternary. This implies that there are several coded or decoded translations for a single λ-term. We could compare this situation to the α-equivalence in the λ-calculus. So, for example, Theorem 1 should be read 'any decoded translation of t reduces to a decoded translation of t''.

We can split the set of interaction rules into five modular groups: β-reduction, duplication, erasing, decoding and encoding. Obviously, depending on what we want to implement, some groups of rules can be eliminated. For example, encoding rules are

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

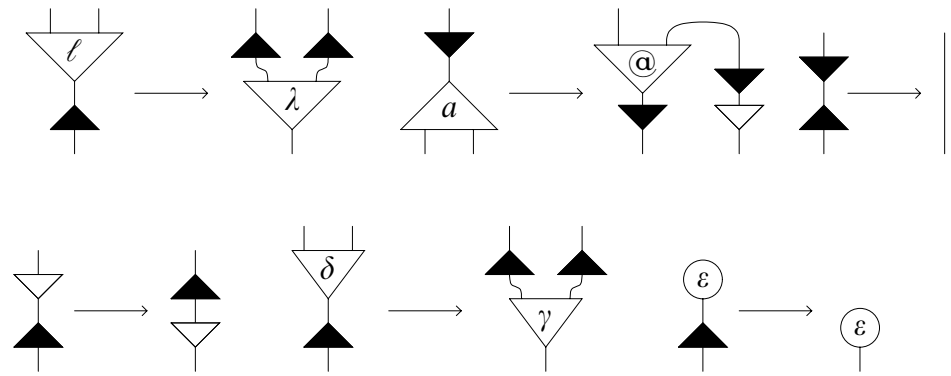


Fig. 6. *Encoding rules*

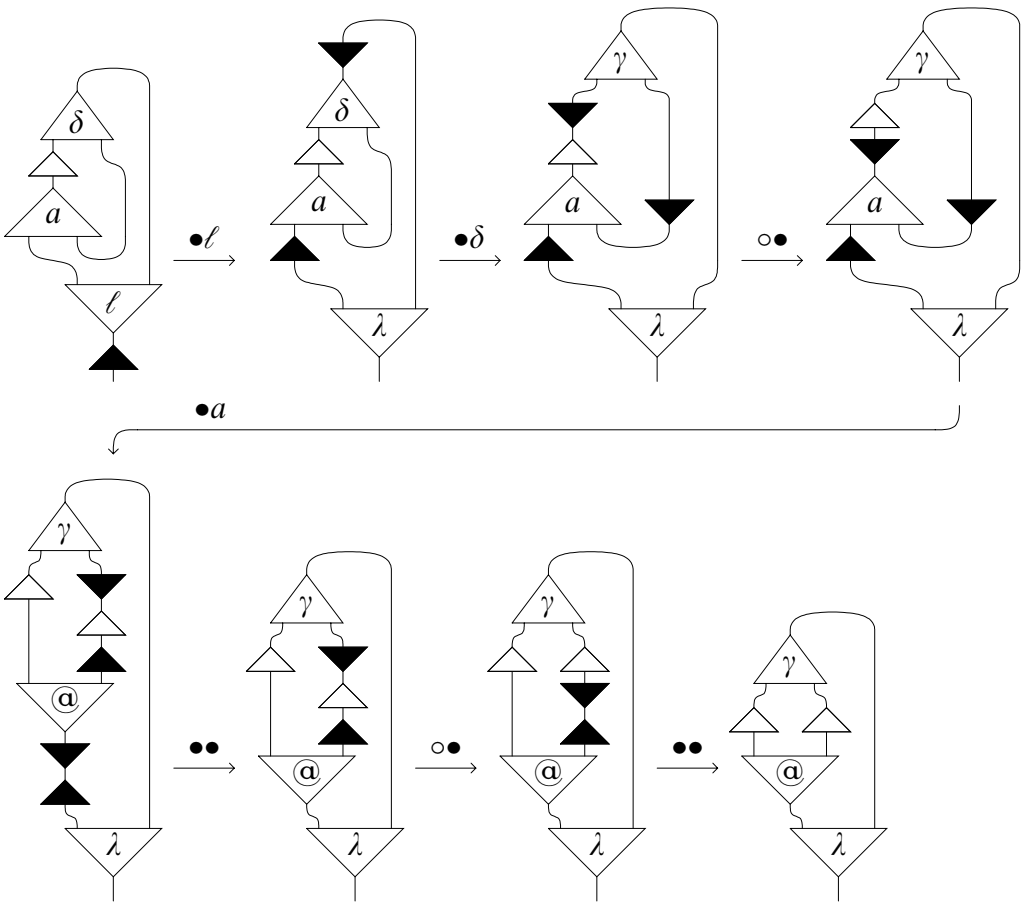


Fig. 7. *Encoding of* $\lambda x\,(x)x$

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

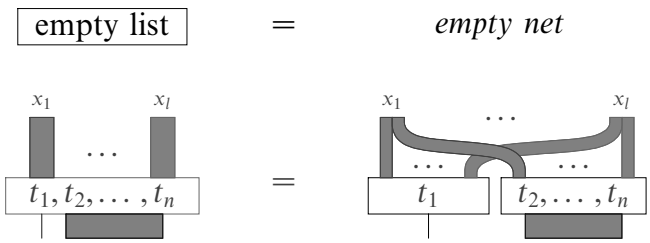
useless if we are just interested in the head normal form of a λ-term. Similarly, if we normalise λI-terms (that is, λ-terms of the form $\lambda x\, u$ in which there is no free occurrence of x in u are forbidden), erasing rules are useless since each argument is used at least once. In fact, the role of the erasing rules is not essential in the general case either. We can take a slightly different translation that includes a *garbage collector* where all unused terms are connected during the reduction[†].

The decoded translation corresponds to the translation into proof-nets (Girard 1987) where *boxes* are coded translations, *derelictions* are decoders, *tensors* are applicators, *pars* are abstractors, *contractions* are duplicators and *weakenings* are erasers. Let us go further in the comparison and compare the cut elimination rules, as presented in Lafont (1995), with the interaction rules of our system. The rule between the *tensor* and the *par* corresponds exactly to the *a*ℓ-rule but the *dereliction-box* rule corresponds to several steps of reduction performed with the decoding rules. Similarly, the *contraction-box* (respectively, the *weakening-box*) rule corresponds to several steps of reduction performed with the duplication (respectively, erasing) rules. Commutative reduction (when an auxiliary port of a box is connected to the principal port of another box) can be considered as an equality. Indeed, substitution is achieved with no reduction in a coded λ-term. Consequently, unlike the case for Linear Logic, there are only two levels of *boxes*: coded or decoded. Another difference is that there is no internal reduction (that is, inside a *box*) since coded translations are principal nets.

4. Proofs

In this section, we give the complete proofs of Theorems 1 and 2 and their respective corollaries. The encoding rules are useless in the proof of Theorem 1 since it deals only with head reduction, that is, decoding.

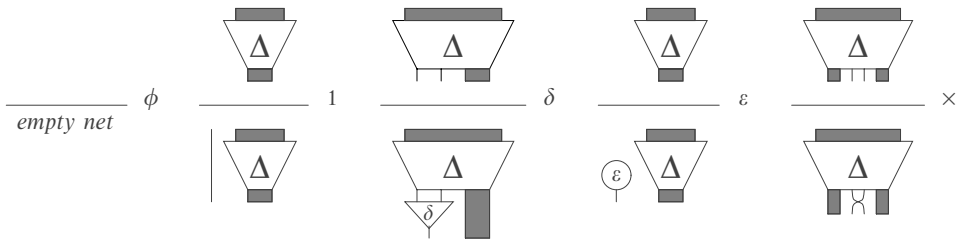
We shall use coded translations of lists of λ-terms: it is simply the juxtaposition of several coded translations. There is nothing deep in this definition; this is just to avoid cumbersome wirings. Here is the definition by induction on the length of the list:



[†] We can use this kind of 'explicit garbage collection' with the generalised sharing of Appendix A to implement weak head reduction using six rules only.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

In the translations, we use generalised duplicators that are generalised trees. More generally, *lists of generalised trees* built with δ- and ε-cells are also called generalised duplicators. Here are the construction rules of the generalised duplicators:

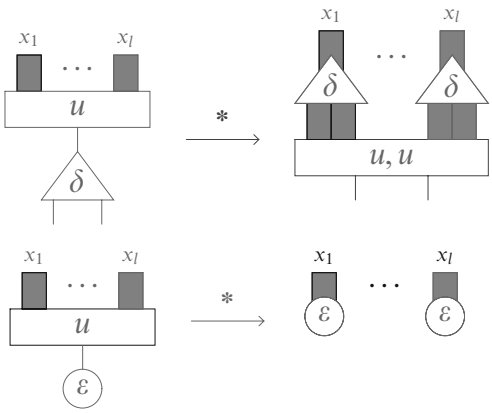


The definition of the generalised sharings is the same except that δ-cells are replaced by γ-cells.

Remark 5. Permutations, bundles of ε-cells or δ-cells are also generalised duplicators.

4.1. *Duplication, erasing*

Lemma 4 (Duplication – erasing). For every λ-term u:

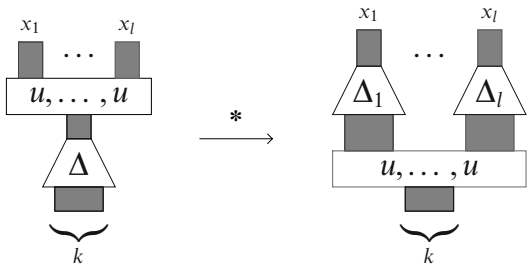


Proof. The proof is by Proposition 2 (using δ-rules and ε-rules) since coded translations are principal nets without any duplicator. □

Lemma 5 is a natural generalisation of Lemma 4. Intuitively, a generalised duplicator takes j 'copies' of a coded λ-term u and gives k copies of u.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

Lemma 5 (Generalised duplication – erasing). For every λ-term u and generalised duplicator Δ:



where $\Delta_1, \ldots, \Delta_l$ are generalised duplicators.

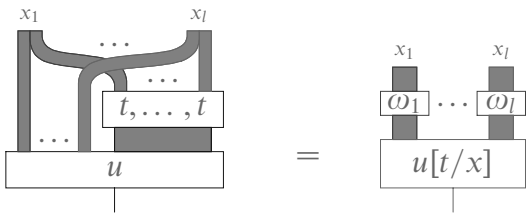
Proof. The proof is by induction on Δ using Lemma 4 (duplication – erasing). □

Remark 6. The number of free occurrences of x_i is not necessarily the same in j copies of u and in k copies of u. That is why the generalised duplicators $\Delta_1, \ldots, \Delta_l$ appear on the right member.

4.2. *Substitution*

Unlike duplication, substitution can be performed both in coded and decoded translations. Consequently, there are two lemmas for substitution called *coded substitution* and *decoded substitution.*

Lemma 6 (Coded substitution). For every λ-term u and t:



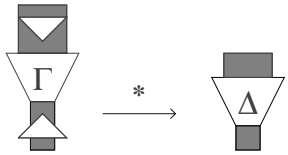
where $\omega_1, \ldots, \omega_l$ are permutations.

Proof. The proof is by induction on u (we do not use any rule since it is an equality). □

Let us examine the x_i bundle of wires, where $i \in 1 \ldots l$, in the above equality. In the left member, it is split into two 'sub-bundles': the left one corresponds to occurrences of x_i in u; and the right one to occurrences of x_i in $t, \ldots, t$. This is not the case in the translation of $u[t/x]$, where all occurrences of x_i in u and in $t, \ldots, t$ are mixed together. So, we have to plug wirings $\omega_1, \ldots, \omega_l$ on each bundle of wires to make this equality correct.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

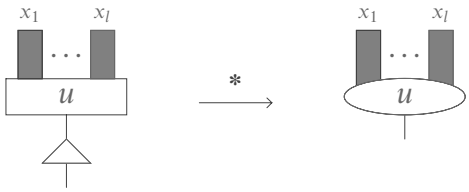
Lemma 7 (Decoding of a generalised sharing). For every generalised sharing Γ:



where Δ is a generalised duplicator.

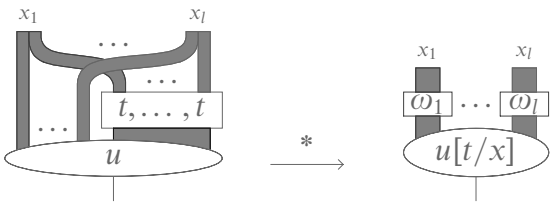
Proof. The proof is by induction on Γ using $\circ\gamma$, $\circ\varepsilon$ and $\circ\circ$. □

Lemma 8 (Decoding). For every λ-term u:



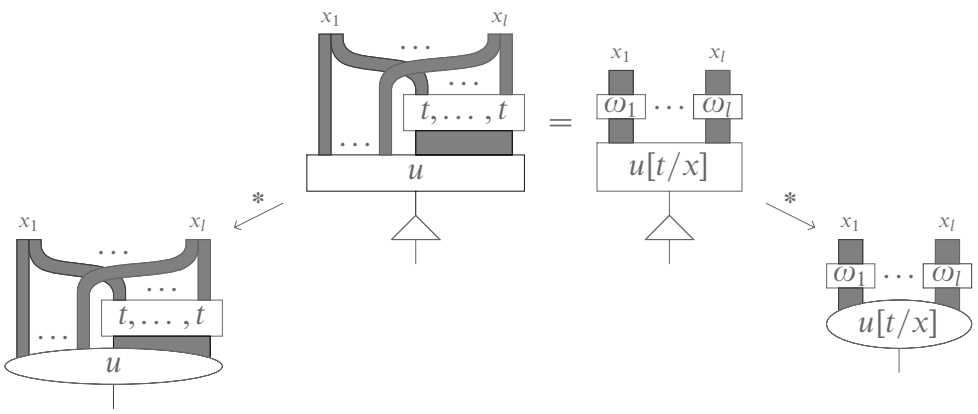
Proof. The proof is by induction on u using $\circ@$, $\circ\lambda$ and Lemma 7. □

Lemma 9 (Decoded substitution). For every λ-term u and t:



where $\omega_1, \ldots, \omega_l$ are permutations.

Proof. By Lemmas 6 (coded substitution) and 8 (decoding), we have:



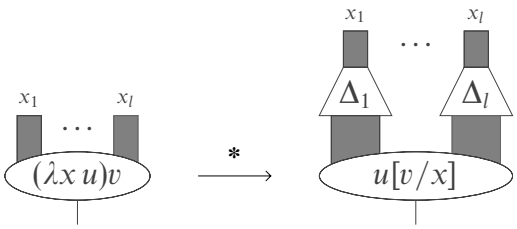
https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

Let us restrict our attention to the sub-system containing decoding rules only. Obviously, Proposition 1 (strong confluence) still holds for this system. Moreover, a decoded translation is irreducible by the decoding rules, so it is the lower right-hand net since it is obtained by plugging permutation on a decoded translation. So, by Corollary 1 (reduction), the lower left-hand net reduces to the lower right-hand net. □

4.3. *Head reduction*

Now we can prove that a decoded redex reduces to its decoded contractum.

Lemma 10 (β-reduction). For every λ-term u and v:



where $\Delta_1, \ldots, \Delta_l$ are generalised duplicators.

Proof. The proof is by $a\ell$, Lemmas 5 (generalised duplication – erasing) and 9 (decoded substitution). □

Let us define one-step head reduction with the following rules:

$$\frac{}{(\lambda x\, u)v \to u[v/x]}\beta \qquad \frac{u \to u'}{\lambda x\, u \to \lambda x\, u'}\lambda \qquad \frac{u \to u'}{(u)v \to (u')v}a$$

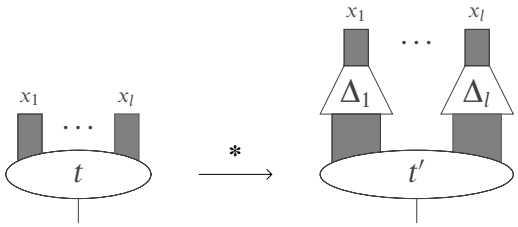
In fact, $t \to t'$ if and only if t' is obtained from t by contracting one of the redexes on the left branch of t[†]. So, this relation is more general than the usual one as defined in Krivine (1993) since the head redex is not the only one that is likely to be reduced. Nevertheless, given a pair of λ-terms t and t' such that t' is the head normal form of t, we have $t \xrightarrow{*} t'$.

Theorem 1 (head reduction) is proved by the following lemma in the restricted case, where t and t' are closed λ-terms. Corollary 2 is simply an application of Lemma 8 (decoding).

[†] The exact terminology for $\to$ is one-step spinal reduction.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

Lemma 11 (Head reduction). For each pair of λ-terms t and t', such that $t \to t'$:



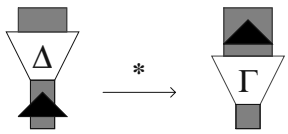
where $\Delta_1, \ldots, \Delta_l$ are generalised duplicators.

Proof. The proof is by induction on the number of rules used to deduce $t \to t'$. Let us consider the last rule: case β is proved by Lemma 10 (β-reduction), cases λ and a simply by the induction hypothesis. □

4.4. *Left reduction*

As with decoding, we prove that generalised duplicators are correctly encoded by the encoder; this is the analogue of Lemma 7 (decoding of a generalised sharing).

Lemma 12 (Encoding of a generalised duplicator). For every generalised duplicator Δ:



where Γ is a generalised sharing.

Proof. The proof is by induction on Δ using $\bullet\delta$ and $\bullet\varepsilon$. □

We define left reduction using the following rules:

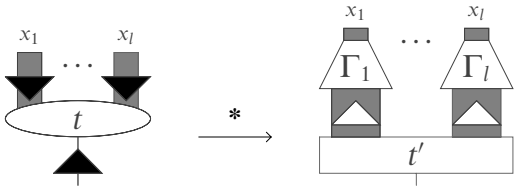
$$\frac{}{x \Rightarrow x}\,x \qquad \frac{u \Rightarrow u'}{\lambda x\, u \Rightarrow \lambda x\, u'}\,\lambda \qquad \frac{u \xrightarrow{*} u_0 \quad u_0 \Rightarrow u'}{u \Rightarrow u'}\,head \qquad \frac{u \Rightarrow u' \quad v \Rightarrow v' \quad u' \neq \lambda x\, t}{(u)v \Rightarrow (u')v'}\,aa$$

The *head* rule corresponds to the idea that we can begin to perform head reduction. This time, we do not define a one-step reduction; for any λ-terms t and t', t' is the normal form of t if and only if $t \Rightarrow t'$.

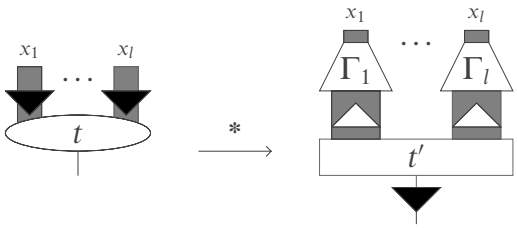
Theorem 2 (left reduction) is proved by the following lemma in the restricted case, where t and t' are closed λ-terms. Corollary 3 is simply an application of Lemma 8 (decoding).

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

Lemma 13 (Left reduction). For each pair of λ-terms t and t', such that $t \Rightarrow t'$:



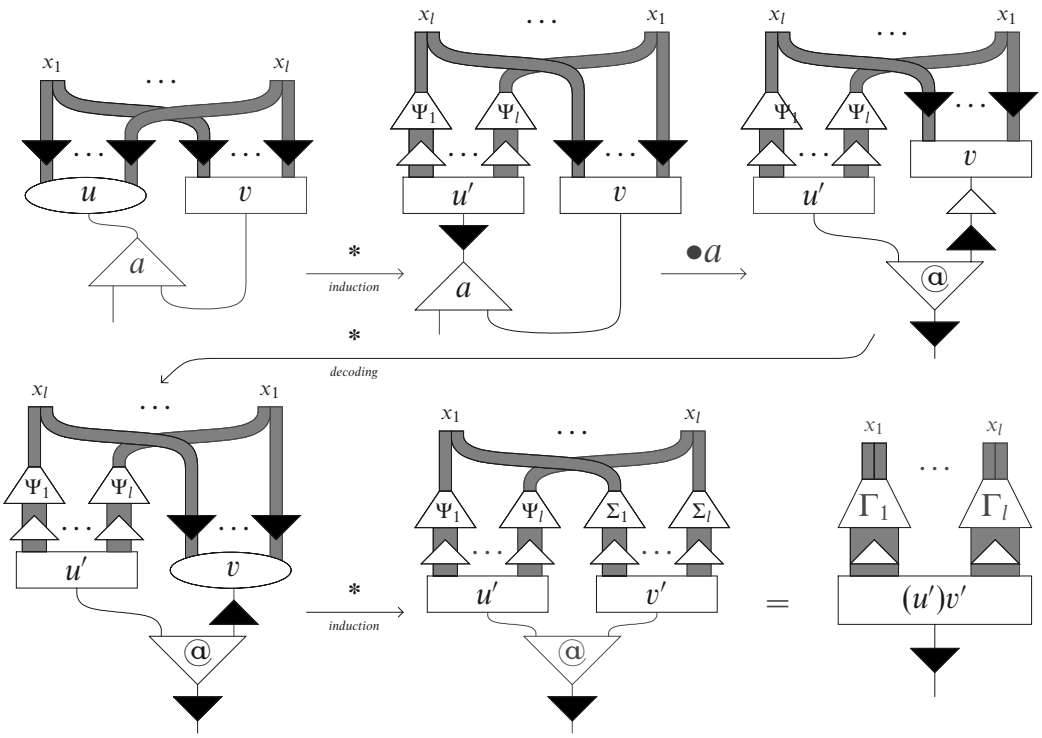
Moreover, if t' is not a λ-abstraction:



where $\Gamma_1, \ldots, \Gamma_l$ are generalised sharings.

Proof. The proof is by induction on the number of rules used to deduce $t \Rightarrow t'$. Let us consider the last rule: case x is proved by $\bullet\circ$, case λ by $\bullet\ell$ and Lemma 12 (encoding of a generalised duplicator), case *head* by Lemma 11 (head reduction) and Lemma 12, and case *aa* by $\bullet a$.

We will give the detailed proof for case *aa*:



where Ψ_i, Σ_i and Γ_i for all $i \in \{1, \ldots, l\}$ are generalised sharings. □

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

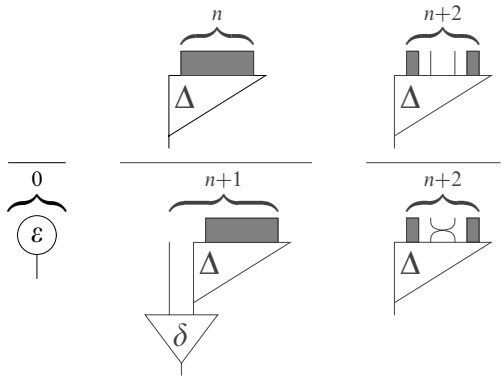


Fig. 8. *The generalised sharing (comb version)*



Fig. 9. *Decoding rules (comb version)*

Appendix A. Another generalised sharing

We present an alternative coded translation for λ-abstraction where decoding rules are simpler and duals of the decoder are useless; going a little further on the way to optimising the number of rules. Moreover, the coded translation of a λ-term is unique up to wires crossing.

From now on, a generalised sharing is a *comb* (that is, a tree growing towards the right) made of several δ-cells and one terminating ε-cell; the construction rules are given in Figure 8. Note that, for a given arity, there is only one generalised sharing up to wires crossing. The translations are the same as those defined in Figure 1, except that the generalised sharing connected to a bundle of ○-cells is replaced by the new generalised sharing: we do not need duals of the decoder any more.

A generalised sharing is a (particular) generalised duplicator. Consequently, decoding rules (Figure 9) are fairly straightforward. The ℓa rule is left unchanged. Erasing rules are the same except that there is no ε○-rule. Note that we are obliged to include erasing rules even in the case of the λI-calculus since translations of λI-terms may contain ε-cells.

Duplication is a bit trickier since a coded translation is a principal net that may contain δ-cells. Again, we need a second symbol (γ) for duplication. γ-cells duplicate δ- and ε-cells, whereas δ-cells duplicate the other cells of a coded translation. Duplication rules are given in Figure 11.

The difficult part is encoding: we have to transform a generalised duplicator into a generalised sharing, that is, a tree into a comb. To this end, we shall introduce another encoder: • of arity 2 (a •-cell or *binder encoder* is represented with a black triangle)

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

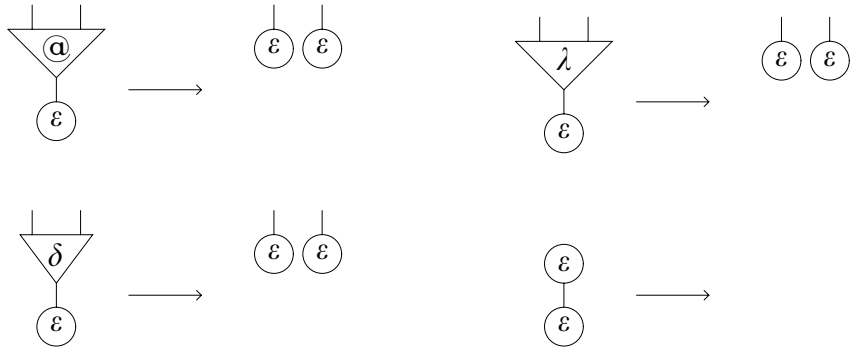


Fig. 10. *Erasing rules (comb version)*

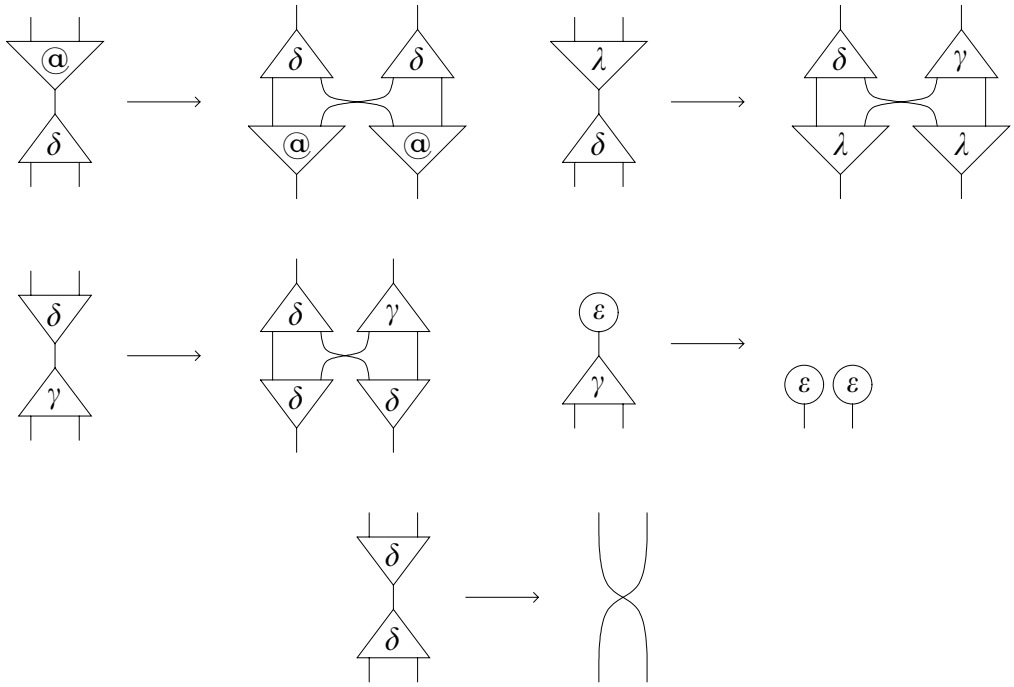
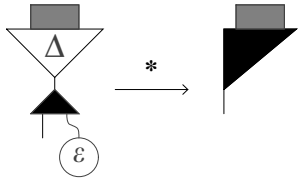


Fig. 11. *Duplication rules (comb version)*

Although we choose the same symbol, there is no risk of confusion between the encoder and the binder encoder because they do not have the same arity. From rules $\bullet\delta$ and $\bullet\varepsilon$ (Figure 12), it easily follows that for every generalised duplicator Δ:



https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

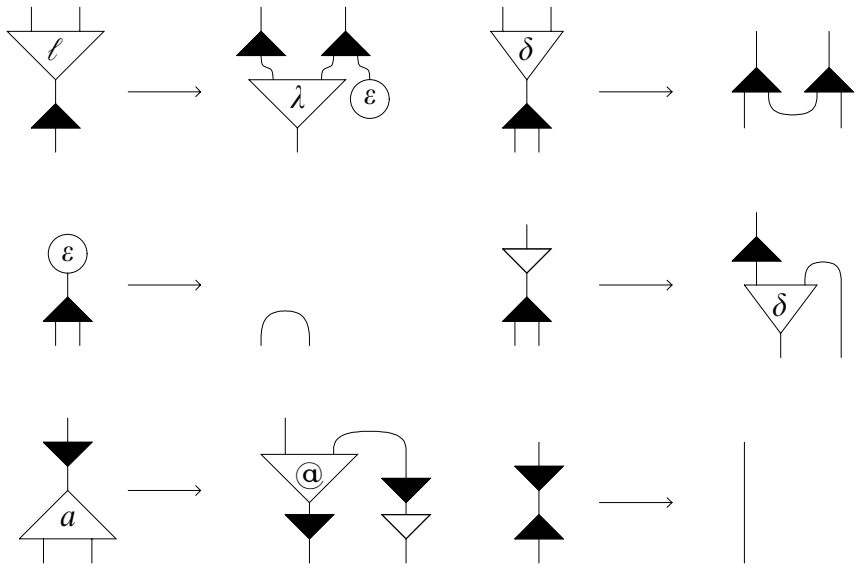


Fig. 12. *Encoding rules (comb version)*

where the right member is a generalised sharing in which  is replaced by .

As before, the decoding–encoding process is activated by the •○ rule (Figure 12). This time, we also have to replace the binder encoder by a duplicator with the correct orientation. The other encoding rules (•*a* and ••) are left unchanged.

Acknowledgment

I would like to thank Yves Lafont for his numerous useful comments.

References

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50** (1) 1–102.

Gonthier, G., Abadi, M. and Lévy, J.-J. (1992) The geometry of optimal lambda reduction. *Proceedings of the 19th Annnual ACM Symposium on Principles of Programming Languages (POPL'92)*, ACM Press 15–26.

Krivine, J.-L. (1985) Un interpréteur pour le λ-calcul. Technical report, Notes de cours de DEA, Université de Paris 7.

Krivine, J.-L. (1993) *Lambda-calculus, types and models*, Ellis Horwood Series in Computers and their Applications. (Translated by Rene Cori from the French into English.)

Lafont, Y. (1990) Interaction nets. *Proceedings of the 17th Annnual ACM Symposium on Principles of Programming Languages, Orlando, U.S.A.* 95–108.

Lafont, Y. (1995) From proof-nets to interaction nets. *Advances in Linear Logic*, London Mathematical Society Lecture Note Series **222**, Cambridge University Press.

Lafont, Y. (1997) Interaction combinators. *Information and Computation* **137** (1) 69–101.

Lamping, J. (1990) An algorithm for optimal lambda-calculus reduction. *Proceedings of the 17th Annnual ACM Symposium on Principles of Programming Languages, Orlando, U.S.A.*

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press

Lang, F. (1998) *Modèles de la β-réduction pour les implantations*, Ph. D. thesis, Ecole normale supérieure de Lyon.

Lippi, S. (2001) Implantation d'un interprète graphique pour les réseaux d'interaction. Technical report available from `http://iml.univ-mrs.fr/~lippi`.

Mackie, I. (1994) *The geometry of implementation*, Ph. D. thesis, Department of Computing, Imperial College of Science, Technology and Medicine.

https://doi.org/10.1017/S0960129502003754 Published online by Cambridge University Press