

THE GRAPHICAL KRIVINE MACHINE

SYLVAIN LIPPI

*Institut de Mathématiques de Luminy, UMR 6206 du CNRS,
163 avenue de Luminy, case 907, 13288 MARSEILLE CEDEX 9, France*
Email: lippi@iml.univ-mrs.fr *

October 22, 2007

Abstract. We present a natural implementation of the Krivine machine in interaction nets: one rule for each transition and the usual rules for duplication and erasing. There is only one rule devoted to the so-called administration. This way, we obtain a graphical system encoding λ -calculus weak head reduction that can be extended to a λ -calculus normalizer by encoding left reduction.

1 Introduction

Interaction nets [7] have been widely used to encode λ -calculus optimal reduction [4] inspired by the algorithm given by Lamping [10]. Indeed interaction nets have become a standard tool for the study of *local* computation and reveal all the elements of a calculus including garbage collection and copying. The disadvantage of such implementations is the potentially large number of extra reductions dedicated to the management of sharing; the so-called *administration*. Moreover, the use of indexed symbols leads to an infinite number of rewriting rules. There have been several attempts to minimize the burden of administration.

Ian Mackie ([15] and [16]) gives an encoding of *boxes* of multiplicative exponential linear logic proof nets. Consequently, using the translations of the λ -calculus into proof nets [3], he obtains encodings of the λ -calculus. More recently, Ian Mackie and Jorge Sousa Pinto [17] have used interaction combinators (a *universal* interaction system introduced by Yves Lafont [8]) to translate proof nets obtaining an efficient encoding. Our implementation relies on a direct translation of the λ -calculus taking advantage of its syntax; we are not trying to encode all the (multiplicative exponential) proof nets but only those corresponding to λ -terms which is a much simpler problem.

Another encoding presented by Frédéric Lang [11] starts from optimal reduction. Similarly to the encoding we presented in [12], it relies on two different symbols for the application: one of them can be duplicated the other one can interact with abstraction. The main difference is that Lang's encoding is based on (a weaker notion of) optimal reduction whereas we start from a standard reduction strategy: left reduction. Consequently, redexes may be duplicated during the computation (*i.e* reduction is not optimal) but we gain a system where both translation and rules are fairly natural. The starting point of this work was a translation of the Krivine machine; this is what is presented in this paper.

*The author wishes to thank the referees for their numerous useful remarks.

Overview. After a quick presentation of interaction nets, we define a version of the Krivine machine where the usual De-Bruijn notation is replaced by simple variable names. Then we give an implementation of the Krivine machine into interaction nets with its correctness proof and obtain a bridge between an environment machine (the Krivine machine) and a graphical system where all reductions are local (interaction nets). Finally, we show how to exploit this correspondence by giving an extension to the implementation of left reduction (call-by-name) in the λ -calculus.

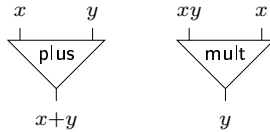
2 The paradigm of interaction

We give a simple introduction to interaction nets via a concrete example. Readers familiar with the paradigm can skip this section. A more detailed presentation as well as other examples can be found in [8] and [14]. Let us write a program that given an arithmetic expression (with variables, $+$ and \times), right expands all the products:

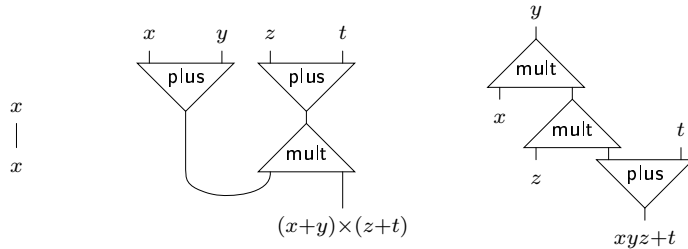
$$x \times (y + z) = x \times y + x \times z \quad (1)$$

2.1 Syntax

What does a program written with interaction nets looks like? Although it is quite related to multiplicative proof nets of Linear Logic, we can introduce it from scratch as a graph rewriting system with some restrictions on the rewriting rules. The basic ingredient of an interaction system is a *symbol* with its *arity*. Occurrences of symbols are called *cells*. Cells have one *principal port* and their number of *auxiliary ports* is determined by the arity of their corresponding symbol. We introduce **plus** and **mult** of arity 2 to represent arithmetic expressions:



The principal port of **plus** corresponds to the result and the auxiliary ports to the arguments. We choose another orientation for **mult**: the principal port corresponds to the second argument. So the difference between principal and auxiliary ports is not related with inputs and outputs but rather with where rewriting can take place. Now with cells and *free ports* we can build a *net* by connecting all the ports (principal, auxiliary or free ports) pairwise:

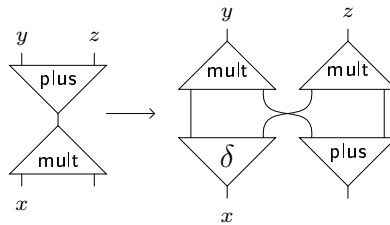


Free ports of the above nets correspond to occurrences of variables or to the root of the expression. Remark that cells can be drawn in any orientation; ports are numbered clockwise

starting from the principal one. Following this convention, the first auxiliary port of a **mult**-cell corresponds to the product and the second one to the first argument. In this example and the other ones, cells are oriented such that nets are close to the syntactic trees of the corresponding expressions.

Wiring. Let us remark that an expression with only one variable without $+$ and \times is represented by a wire between two free ports. More generally, a *wiring* is a net built with only free ports connected pairwise.

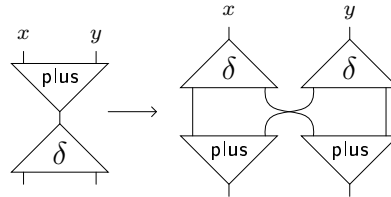
Let us continue with our little example with the core of a program: the *interaction rules*. A rule is a pair of nets (*left member* \rightarrow *right member*) with the same number of free ports. The important restriction is that the left member must be a *cut* *i.e* a net built with two cells connected by their principal ports.¹ An interaction system has, at most, one rule for each pair of symbols. The rule between **mult** and **plus** faithfully corresponds to equation 1:



The second argument of a product is connected to the principal port of **mult** so that an interaction occurs when it is a sum. Let us look at equation 1: argument x appears once in the left member and twice in the right member. So we introduce a binary symbol δ to preserve the number of free ports. The principal port of δ is connected to the expression that must be duplicated and the auxiliary ports to the two resulting copies.

Remark 2.1 *Free ports of a rule corresponding to the arguments are named (the other ones are generally left unnamed) simply to help the reader since they can be identified by their geometric positions.*

Rule δ -**plus** corresponds to the duplication of a sum expression: both arguments are copied and connected to the auxiliary ports of **plus**-cells:

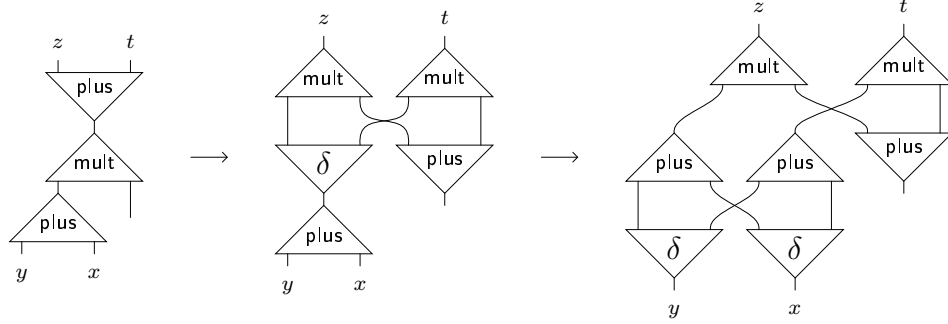


We do not need to specify a rule between δ - and **mult**-cells since the result of a product is connected to the first auxiliary port of **mult**. There is also no rule corresponding to the duplication of variables since variables correspond to free ports. Once the expression is expanded, we obtain a net where variables and products occurring several times are *shared* by δ -cells.

¹if the two cells of the left member have the same symbol, there is also a symmetry condition on the right member, see [9].

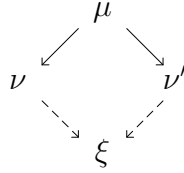
2.2 Execution

Once a set of symbols and rules has been fixed, we can apply one of those rules to a net obtaining another net and so on until we have reached an irreducible one. The *reduction* relation is denoted by \rightarrow . Let us trace an execution from our example program where $(x + y) \times (z + t)$ is expanded into $(x + y) \times z + (x + y) \times t$:



Of course, as shown in figure 1, there may be several cuts in a net. The important property shared by all the interaction systems is that they may be reduced in any order. This strong confluence property is a direct consequence of the restriction on the left member of a rule: the left member is a cut.

Proposition 2.1 (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.

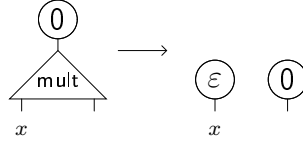
Corollary 2.1 (reduction) *If a net μ reduces to an irreducible net ν in n steps, then any reduction starting from μ eventually reaches ν in n steps.*

2.3 Duplication and erasing

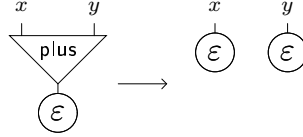
An important consequence of the *linearity restriction* (same number of free ports in the left and right member of a rule) is that arguments that are used several times must be explicitly duplicated. This is what is done by δ -cells. In case an argument is not used, it must be explicitly erased. Let us extend our example program with constant 0 so that products can be simplified if the right argument is 0:

$$x \times 0 = 0 \quad (2)$$

Consequently, we introduce two symbols 0 and ε of arity 0. Similarly to δ -cells, ε -cells are used to erase an argument that is not used. Equation 2 is translated into the following rule:



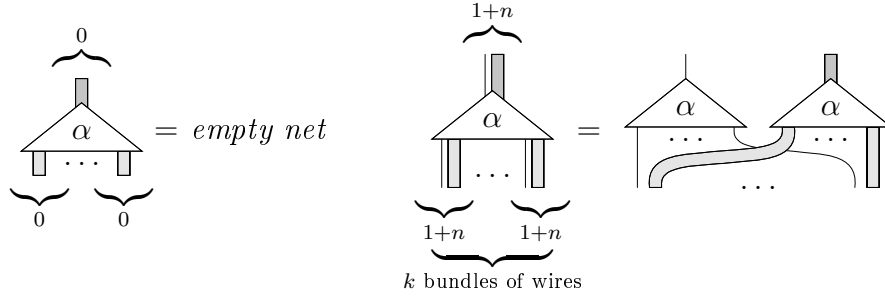
Remark that cells of arity zero (*i.e* with only one port) are pictured with a circle since we do not have to distinguish the principal port from the other ones. As previously, we have to give the rule between ε and **plus**. Remark that this is the same as the previous one except that **mult** and **0** are respectively replaced by **plus** and ε :



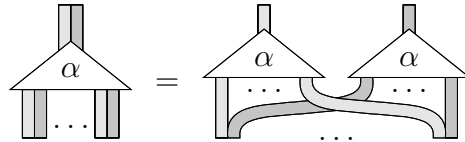
Let us conclude this presentation by proving that duplication and erasing are correctly implemented in our example program. This property only deals with a small class of nets called *trees*. A tree is a wire (in this case the root is fixed arbitrarily) or a cell with auxiliary ports connected to the root of smaller trees and the root is connected to the principal port of the cell. We shall extend duplication and erasing to a wider class of nets for our translation of the Krivine machine. We use the following notation for a *bundle of wires*:

$$\boxed{} = \left| \dots \right|$$

Moreover, for every symbol α of arity k , we define a *bundle of n α -cells* by induction on n in the following way:²

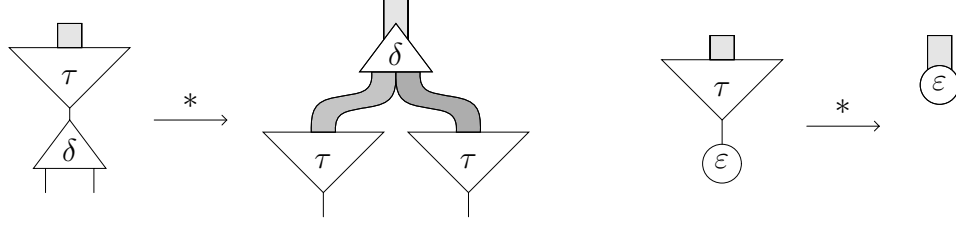


where an *empty net* is a net built with no cell and no free port. Obviously, a bundle of a single α -cell ($n = 1$) corresponds exactly to the net built with one α -cell and $k + 1$ free ports connected to the ports of the α -cell. Moreover, a bundle can be easily decomposed in the following way:



²the permutation connected to the auxiliary ports of the α -cells is called a *perfect shuffle* or a *butterfly net*.

Lemma 2.1 (duplication/erasing of trees) *For any tree τ built with plus-cells:*



Proof. by induction on τ using the rules between **plus** and δ or ε .

3 The naive Krivine machine

We present a simple version of the Krivine machine [5]. One of the main differences with the usual one is that there is no De Bruijn notation for variables; this way we obtain a more natural version using variable names. On the other hand we do not use references to environments so that they are duplicated and not shared [2]. We shall see that this lack of efficiency is circumvented by our translation into interaction nets. We show that this machine evaluates closed λ -terms to weak head normal form in appendix A.

Let us have a set of variable names $\{x_1, \dots, x_\ell\}$.

Definition 3.1 (Closure, Environment)

- $*$ is a closure.
- If T_1, \dots, T_ℓ are closures, then $\langle T_1, \dots, T_\ell \rangle$ is an environment.
- If t is a λ -term and E an environment, then (t, E) is a closure.

Remark 3.1 $*$ is a placeholder; it is used to reserve a place in an environment before a “real” closure takes place. The usual empty environment corresponds to the tuple $\langle *, \dots, * \rangle$.

Convention. λ -terms are noted r, s, t, u, v, w, \dots and closures R, S, T, U, V, W, \dots .

Definition 3.2 (Stack)

- 1 is a stack called the empty stack.
- If T is a closure and Π is a stack, then $T\Pi$ is a stack.

Definition 3.3 (Configuration) A configuration is a pair (T, Π) where T is a closure and Π a stack.

Notation. Let an environment $E = \langle T_1, \dots, T_\ell \rangle$, a variable x_i and a closure U . We define the following two notations:

$$E(x_i) \stackrel{\text{def}}{=} T_i$$

$$E[x_i := U] \stackrel{\text{def}}{=} \langle T_1, \dots, T_{i-1}, U, T_{i+1}, \dots, T_\ell \rangle$$

Intuitively, a closure corresponds to a closed λ -term and an environment gives the values that are necessary to evaluate a λ -term containing free variables. The $*$ closure is a “useless” closure; for example, there is no need to know the value of x_1 for the evaluation of a λ -term with no free occurrence of x_1 . So we can interpret $E(x_i)$ as the value given to x_i in E and $E[x_i := U]$ as the assignment of U to x_i . Those correspondences are precisely defined in appendix A. Let us give the transition rules. We use Krivine’s notation for λ -terms *i.e* term u applied to v is noted $(u)v$.

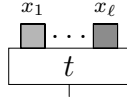
| | | | | |
|------|--------------------|--------------------|------------------|-------------|
| push | $(u)v, E$ | $\Pi \rightarrow$ | u, E | $(v, E)\Pi$ |
| pop | $\lambda x_i u, E$ | $V\Pi \rightarrow$ | $u, E[x_i := V]$ | Π |
| eval | x_i, E | $\Pi \rightarrow$ | $E(x_i)$ | Π |

Unlike the usual Krivine machine, transition *push* must be understood as duplicating the environment E ; we do not use references to environments and each λ -term has its “own” environment. Consequently, there is no need of α -conversion for the *pop* transition. Indeed, there is no free occurrence of x_i in $\lambda x_i u$, so we can directly replace the closure corresponding to x_i in E by U ; this is exactly how $E[x_i := U]$ is defined.

4 Encoding the Krivine machine with interaction nets

4.1 Translation

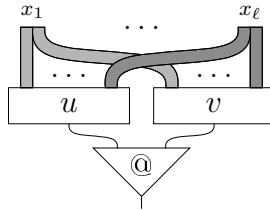
Naturally, the essential ingredients of a Krivine machine are λ -terms. The first idea is to represent a λ -term t by its syntactic tree. We use the following notation for the translation of t :



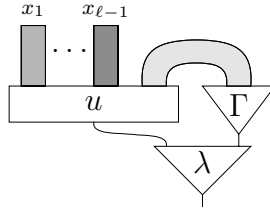
The unique free port of the lower part corresponds to the root of the λ -term and free ports of the upper part correspond to occurrences of free variables. Let us remark that a variable name corresponds to a bundle of wires since there may be several occurrences of this name. Consequently, we translate a variable x_i by a single wire:



Following our “syntactic tree” approach, we introduce @-cells of arity two and translate an application $(u)v$ by connecting the translation of the function part u and of the argument part v to the auxiliary ports of an @-cell:



The usual way to take advantage of the graphical syntax is to connect occurrences of bound variables to their corresponding linker. Consequently, we use λ -cells (of arity two) and connect the first auxiliary port of a λ -cell to the body of the abstraction and the second one to occurrences of the corresponding bound variables.³ There may be several occurrences of bound variables, so we introduce γ -cells (called *sharing*) of arity two; we gather all occurrences of bound variables in a list built with γ -cells and connect the list to the second auxiliary port of the λ -cell. We also introduce an ε -cell of arity zero in case there is no variable bound by a λ -cell. Finally we obtain the following translation of an abstraction $\lambda x_\ell u$:



where Γ (see *generalized sharing* figure 4) is either a comb built with γ -cells (in case there are several free occurrences of x_ℓ in u), a single wire (in case there is exactly one free occurrence of x_ℓ in u) or an ε -cell (in case there is no free occurrence of x_ℓ in u).

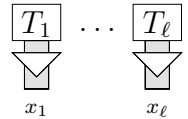
The above use of λ -, γ - and ε -cells is very close from what is usually done in sharing graphs [1]. The important difference is that the principal port of the $@$ -cell corresponds to the root of the application and not to the function. This way, the translation of a closed λ -term is a *package* (i.e a tree with all the leaves connected pairwise) and we can benefit from the duplication rules described in section 2. The translation of a λ -term is summed up in figure 3 where Γ represents a generalized sharing as defined in figure 4. Finally, examples are given in figure 2.

Now we can define the translations of the other components of a Krivine machine: closures, environments, stacks and configurations. The translation of a closure T and an environment E are respectively represented as follows:



The translation of a closure is essentially obtained by plugging juxtaposed translations of a λ -term with the translations of other (smaller) closures that may be used in subsequent substitutions. Remark that we introduce unary \circ -cells (represented by an empty white cell) to mark variables “waiting” for a substitution.

- The translation of the useless closure $*$ is the *empty net*.

- The translation of the environment $E = \langle T_1, \dots, T_\ell \rangle$ is .

³The roles of the auxiliary ports of the λ -cell have been permuted from what is usually done. The goal is just to avoid complicated wirings in the reduction.

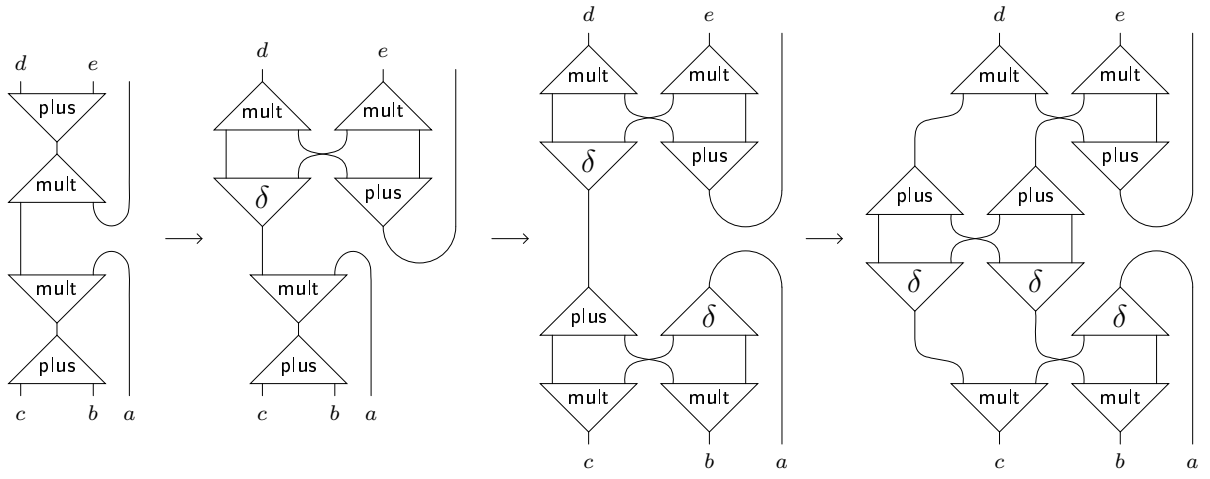


Figure 1: $a(b+c) \times (d+e)$ expanded into $(ab+ac)d + (ab+ac)e$

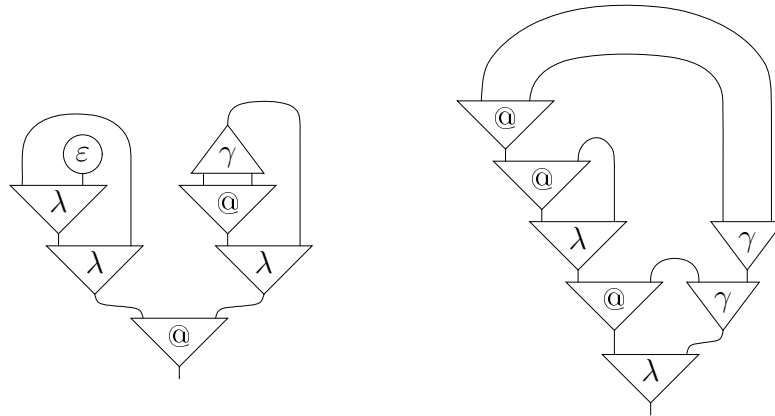
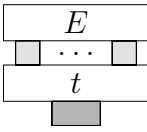



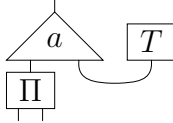
Figure 2: $(\lambda x \lambda y x) \lambda x (x)x$ and $\lambda x (\lambda y ((x)x)y)x$

- The translation of the closure (t, E) is .

Actually, a closure has several translations depending on the number of times it can be used. Consequently, an environment can also have several translations. This is the main point where the naive Krivine machine differs from our implementation. Whereas transition *push* leads to the duplication of the environment in the naive Krivine machine, it corresponds to “splitting” the environment in our implementation so that the function and the argument are connected only to part of the environment necessary for the evaluation of their respective free variables. Finally, the translations of a stack Π and a configuration \mathcal{C} are respectively pictured as follows.

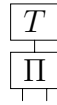


We introduce a binary symbol a to build stacks of closures.

- Translation of the empty stack is .
- Translation of stack $T\Pi$ is .

Remark 4.1 *We could also use two different symbols (for example cons and nil) with arities two and zero. The reason for which we use a unique symbol a is just to minimize the number of rules.*

Finally, the translation of a configuration $\mathcal{C} = (T, \Pi)$ is obtained by connecting the closure T to the top of Π .



Let us remark that the translations of T and Π are reduced; there is no cut in such nets. Therefore a configuration contains exactly one cut; this cut comes from the connection of an a -cell (the top of the stack) and a cell that occurs in a closure. The notations and translations of the Krivine machine components (Closures, Environments, Stacks and Configurations) are summed up in figure 5.

4.2 Duplication and erasing of a closure

During the transitions of the naive Krivine machine, some closures must be duplicated or erased. The translation of a closure is a package built with @-, λ -, γ -, ε - and \circ -cells. We have seen in section 2 how trees can be duplicated or erased. We can introduce a binary symbol δ and use the same schema for packages with a supplementary rule for $\delta\delta$ as shown in figure 6.

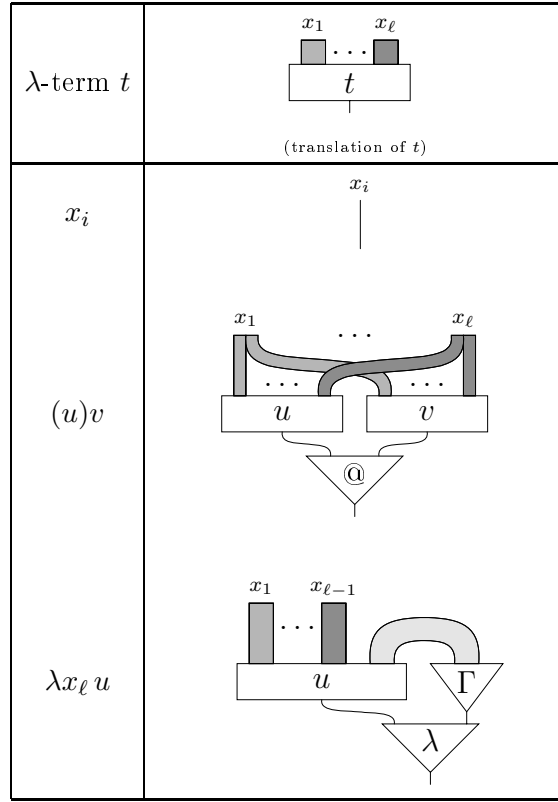


Figure 3: *Translation of a λ -term for the Krivine machine*

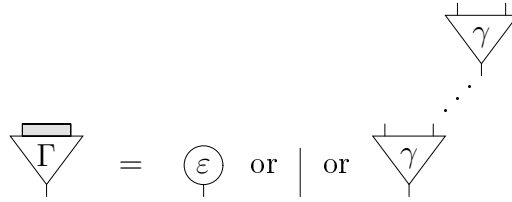

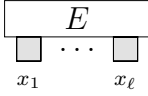




Figure 4: *Generalized Sharing*

| Component | Notation |
|-----------------------------|---|
| Closure T |  |
| Environment E |  |
| Stack Π |  |
| Configuration \mathcal{C} |  |

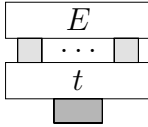
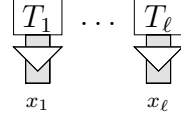

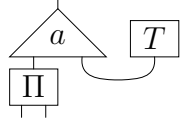

| | Component | Translation |
|---------------|--------------------------------------|---|
| Closure | $*$ | <i>empty net</i> |
| | (t, E) |  |
| Environment | $\langle T_1, \dots, T_\ell \rangle$ |  |
| Stack | 1 |  |
| | $T \Pi$ |  |
| Configuration | (T, Π) |  |

Figure 5: *Components of the Krivine machine*

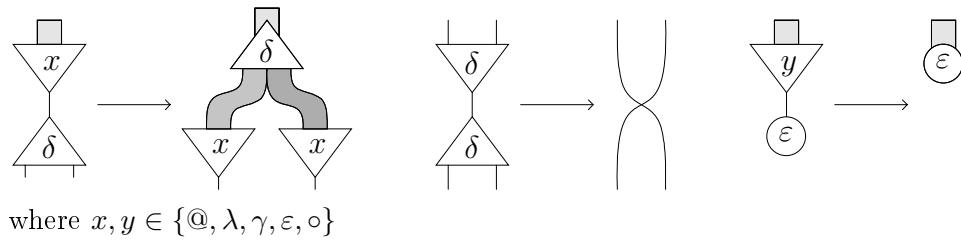


Figure 6: *Duplication/erasing rules for closures and λ -terms*

4.3 Transition rules

Transitions are enabled by an interaction between the top of the stack (*i.e* an a -cell) and a closure (*i.e* a $@$, λ or \circ -cell). The $a@$ and $a\circ$ rules respectively correspond to transitions *push* and *eval*.



Let us remark that those transitions are performed with only one interaction. On the other hand transition *pop* corresponds to $a\lambda$ and $\circ\gamma$ rules.



This transition is performed by several interactions: γ -cells are transformed into δ -cells (this is the real administrative part) and then, the closure on the top of the stack is duplicated by δ -cells.

4.4 Comments

The system above only implements weak head reduction, so it is clearly less ambitious than optimal reduction for instance. On the other hand, it is one of the smallest systems encoding β -reduction. For example, we do not have to introduce extra symbols and their corresponding rules such as *crochet* or *croissant* that are used to manage sharing. However we could perhaps improve the efficiency by using a weak reduction strategy as described by Jorge Sousa Pinto in [18]; this way we could avoid some unnecessary reduction steps. For example, a closure consists of several copies of the same (closed) λ -term; using a weak reduction strategy would help producing only copies that are really needed. Moreover, standard implementations of Krivine machine where environments are shared need a garbage collector since their size grows during the execution. This is not the case of our implementation which includes “memory management”.

Therefore we can consider our translation as a minimal graphical syntax where β -reduction is decomposed in several local reductions. In other words, it is both a didactic tool giving another point of view on λ -terms and a basic implementation that could be probably reused to implement other reduction strategies and λ -calculi.

4.5 Proof

We prove that our translation into interaction nets is an implementation of the Krivine machine. In other words, we prove that it does everything the specification describes (this is the difficult part) and then that it does nothing else (this is mainly a consequence of strong confluence).

4.5.1 Completeness

First of all, let us formulate what we call completeness.

Proposition 4.1 *For any pair of configurations \mathcal{C} and \mathcal{C}' such that $\mathcal{C} \rightarrow \mathcal{C}'$,*

$$\boxed{\mathcal{C}} \xrightarrow{*} \boxed{\mathcal{C}'}$$

So we have to prove that the three transitions are correctly simulated. Case *push* is proved using rule *a@* and the following lemma:

Lemma 4.1 *For any closure T and environment E we have the following decomposition:*

$$\boxed{T} = \boxed{T} \boxed{T} \quad \text{and} \quad \boxed{E} = \boxed{E} \boxed{E}$$

Proof. By induction on T .

Case *eval* is proved using rule *ao* and the translation of an environment. Case *pop* is a bit more complicated since it is simulated with the *aλ*-rule and the administrative rules, in particular *duplication/erasing* rules. To begin with, let us prove the following lemma:

Lemma 4.2 (Duplication/Erasing of a closure) *For any closure T ,*

$$\boxed{T} \triangle \delta \xrightarrow{*} \boxed{T} \boxed{T} \quad \boxed{T} \circ \varepsilon \xrightarrow{*}$$

Proof. A closure T can be decomposed into a tree τ connected to a wiring ω in the following way:

$$\boxed{T} = \boxed{\omega} \triangle \tau$$

So we have to prove a duplication and erasing property for those two components.

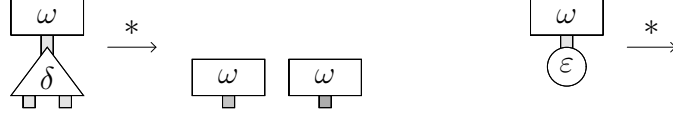
Remark 4.2 *There are no δ -cells in the translation of a closure. This is an important restriction concerning duplication in general [14].*

Lemma 4.3 (Duplication/Erasing of a tree) *For any tree τ built with @-, λ-, γ-, ε- and o-cells,*

$$\triangle \tau \xrightarrow{*} \triangle \delta \triangle \tau \triangle \tau \quad \triangle \tau \circ \varepsilon \xrightarrow{*} \circ \varepsilon$$

Proof. The proof is analogue to the proof of lemma 2.1.

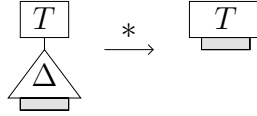
Lemma 4.4 (Duplication/Erasing of a wiring) *For any wiring ω ,*



Proof. By induction on the number of wires using $\delta\delta$ and $\varepsilon\varepsilon$ rules.

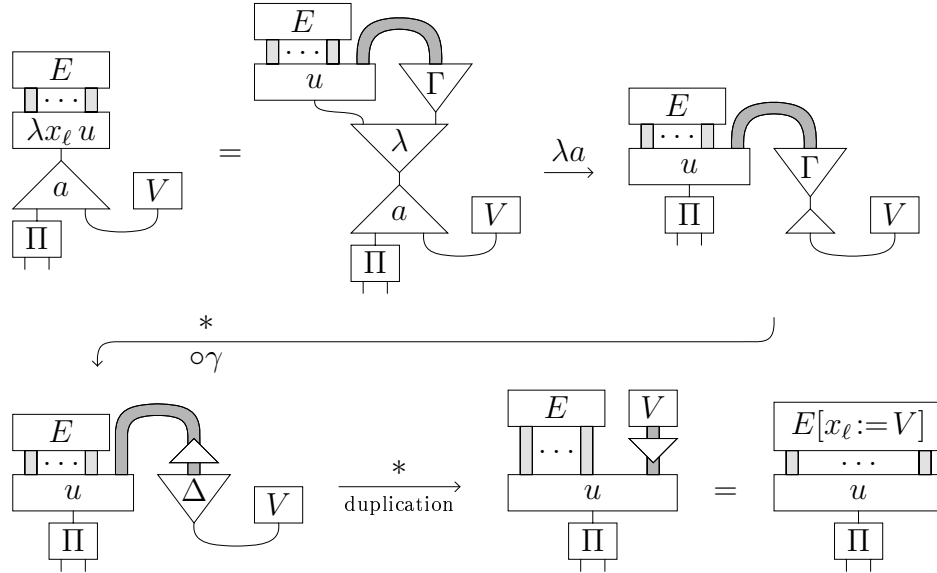
Let us give a more general statement of lemma 4.2 since several copies of a closure (maybe zero) are likely to be created during reduction.

Lemma 4.5 (Generalized duplication of a closure) *For any closure T and tree Δ built with δ - and ε -cells,*



Proof. By induction on Δ using lemma 4.2 and lemma 4.1.

Now, we can prove the *pop* case as follows:



where Δ is a generalized sharing with γ -cells replaced by δ -cells.

4.5.2 Correctness

Notation. We write $\xrightarrow{\mathcal{A}}$ the *administrative* reduction relation of the system without the transition rules *i.e* without $a\circ$ -, $a@$ - and $a\lambda$ -rules and define the set \mathcal{N} of nets that eventually correspond to a configuration of the Krivine machine.

$$\mathcal{N} = \left\{ \boxed{\nu} / \exists \mathcal{C} \boxed{\nu} \xrightarrow{\mathcal{A}^*} \boxed{\mathcal{C}} \right\}$$

Remark 4.3 *The translation of a configuration is irreducible by $\xrightarrow{\mathcal{A}}$ so by confluence the configuration \mathcal{C} is unique and we have defined a surjection φ from \mathcal{N} to the set of configurations of the Krivine machine.*

Remark 4.4 *There is exactly one cut ($a@$, $a\lambda$ or $a\circ$) in the translation of a configuration.*

Remember (see remark 4.1) we use a single symbol (a) to build stacks. Consequently, a cut between the a -cell that *terminates* the stack of closures and a λ -cell must not be eliminated since it appears in (the translation of) a configuration where no transition can take place. In other words, the two free ports of a net ν corresponding to a configuration (that is in \mathcal{N}) are connected to the auxiliary ports of an a -cell; we shall say that ν is *a-terminated*.

Remark 4.5 *Basically, the result of a weak reduction is a λ -term t and a variable free in t . Indeed, this is what we obtain if a configuration is completely reduced.*

Lemma 4.6 *For any configuration \mathcal{C} and a-terminated net ν such that $\boxed{\mathcal{C}} \longrightarrow \boxed{\nu}$, we have $\nu \in \mathcal{N}$ and $\mathcal{C} \rightarrow \varphi(\nu)$.*

Proof. The translation of \mathcal{C} can be reduced without eliminating the “terminal” a -cell so a transition can take place in \mathcal{C} . A transition is simulated with an a -reduction ($a@$, $a\circ$ or $a\lambda$) followed by some administrative reductions. Consequently, using remark 4.4, we have the following reduction.

$$\boxed{\mathcal{C}} \xrightarrow{*} \boxed{\nu} \xrightarrow{\mathcal{A}^*} \boxed{\mathcal{C}'}$$

where $\mathcal{C} \rightarrow \mathcal{C}'$

Proposition 4.2 *For any configuration \mathcal{C} and a-terminated net ν such that $\boxed{\mathcal{C}} \xrightarrow{*} \boxed{\nu}$, we have $\nu \in \mathcal{N}$ and $\mathcal{C} \xrightarrow{*} \varphi(\nu)$.*

Proof. By induction on the length of the reduction using lemma 4.6.

5 Extension to a λ -calculus normalizer

Krivine machine implements weak head reduction so the previous system is also an implementation of weak head reduction. We show how to implement a λ -calculus normalizer starting from this. We start by redefining the previous system without any reference to the Krivine machine; this is just another presentation of the system. Then we show how it can be extended to implement left reduction.

5.1 Coded and Decoded translations of a λ -term

Let us make some general observations about the Krivine machine and its translation into interaction nets. The translation of a λ -term (or closure) is a *reduced net* so there is no interaction in it. Indeed this translation has been specifically designed to be duplicated or erased by a δ - or ε -cell. A transition of the Krivine machine (or weak head reduction of a λ -term) results from an interaction between a closure and the top of a stack of closures. That is an interaction between an a -cell and a $@$ -, λ - or \circ -cell coming from a closure.

The previous observations lead us to introduce two different translations of λ -terms: a coded one that can be duplicated and decoded one where β -reduction can take place. This presentation is detailed in figures 8 and 9. Moreover, let us remark that we can associate a λ -term to the configuration of a Krivine machine by expanding all substitutions; the precise definition of such translation is given in appendix A. This way, we can reformulate the translation of a configuration into the decoded translation of the corresponding λ -term. In fact those translations differ only for the application case. In the coded case we use an @-cell with the principal port connected to the root of the λ -term whereas in the decoded case we use an a -cell (called *applicator*) with the principal port connected to the function. Of course rules are unchanged: we just give another presentation of the translation replacing the components of the Krivine machine (closure, environment, stack, configuration) by a coded and decoded translation of λ -terms.

5.2 Implementation of a λ -calculus normalizer

We give here a quick overview of a λ -calculus normalizer starting from the ideas used for the Krivine machine and the remarks of the previous section. A detailed description of this system with a detailed proof as well as a comparison with Girard's translation in proof nets can be found in [12].

5.2.1 Translations

A first step is to modify the decoded translation to implement head reduction. We have seen that β -reduction takes place in a decoded translation whereas there is no interaction in a coded translation. Consequently, the decoded translation of a λ -abstraction must be modified so that β -reduction can occur under a λ -abstraction. To make clear the difference between a coded and decoded translation, we introduce another symbol of arity two ℓ (called *abstractor*) for the decoded abstraction. The first auxiliary port of the abstractor must be connected to the decoded translation of the body of abstraction. The second one is connected to the root of a tree built with δ -cells (called *duplicators*). The coded translation of an abstraction is slightly modified by marking bound variables with \circ -cells. We shall see the technical role of those cells in the rules. The last change is concerning the translation of a decoded variable *i.e* the head variable. Since it is likely to be substituted during head reduction by a (coded) λ -term we have to mark its occurrence with a unary cell \circ (called *decoder*) that decode λ -terms coming from substitution of the head variable. Those translations are summed up in figure 10.

5.2.2 Rules

As previously for the Krivine machine we can split the rules into two groups: the duplication/erasing rules and the *decoding* rules. The first one is left unchanged and the second one gathers rules between the decoder and cells used to built a coded translation. The decoding rules faithfully follow the definition of the decoded translation; they are summed up in figure 11. Let us remark that β -reduction corresponds to an interaction between an applicator and an abstractor so the $a\lambda$ -rule is replaced by the $a\ell$ -rule given in figure 12.

Finally, we implement left reduction by iterating head reduction on the arguments u_1, \dots, u_n of a head normal form $\lambda x_1 \dots \lambda x_k (x_i) u_1 \dots u_n$. We have seen that head reduction corresponds to decoding so we have to decode the arguments of a head normal form and then re-encode to

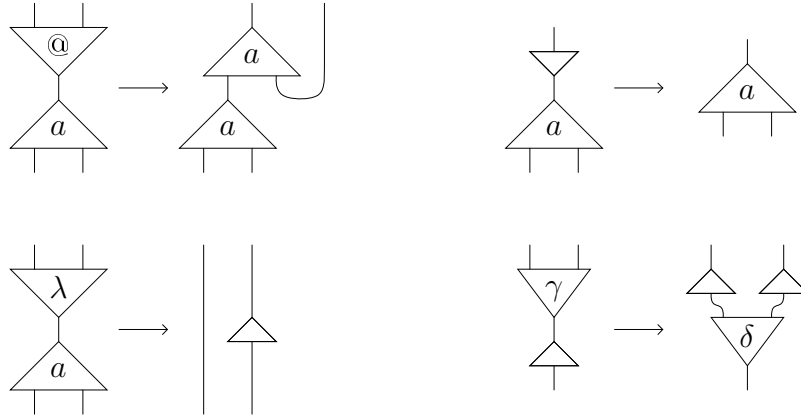


Figure 7: *Transitions of the Krivine machine*

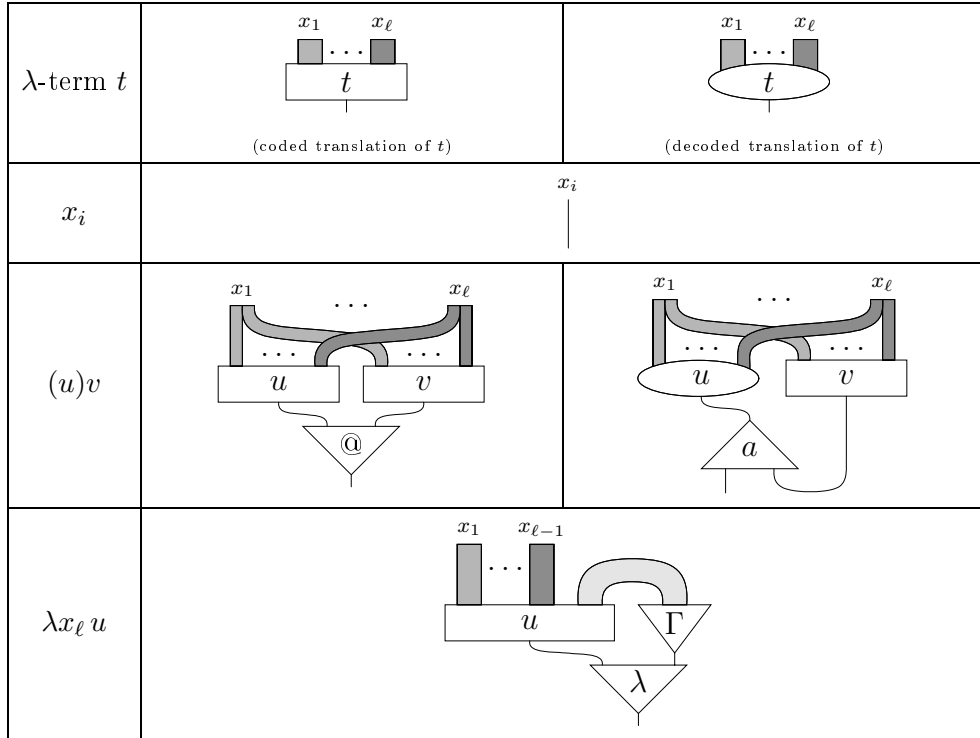


Figure 8: *Translations for weak head reduction*

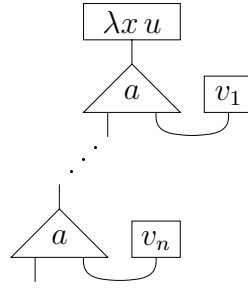


Figure 9: *Decoded translation of a closed λ -term $(\lambda x u)v_1 \dots v_n$ for weak head reduction*

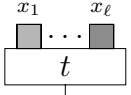
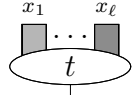


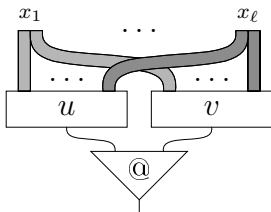
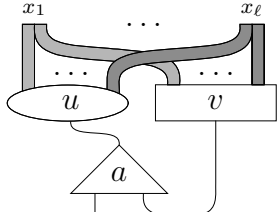
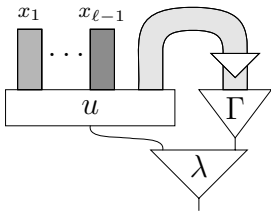
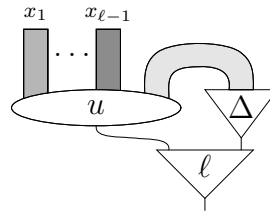
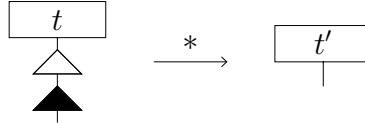
| λ -term t |  (coded translation of t) |  (decoded translation of t) |
|---------------------|--|--|
| x_i |  |  |
| $(u)v$ |  |  |
| $\lambda x_\ell u$ |  |  |

Figure 10: *Translations for the normalizer*

preserve the decoded translation. So we introduce a unary cell \bullet (called *encoder*) to perform this task. The encoding rules are summed up in figure 13.

To conclude, we can sum up the process of left reduction in two stages: decoding (*i.e* head reduction) and encoding (*i.e* left reduction). This comes from the fact that to perform left reduction we can first compute the head normal form. This is formulated in the following proposition.

Proposition 5.1 (Left reduction) *For any pair of λ -terms t and t' such that t' is the normal form of t ,*



Proof. The detailed proof can be found in [12].

6 Conclusion

This implementation of the Krivine machine can be considered as a basic graphical tool that lies between the textual λ -calculus syntax (where duplication and the use of alpha-conversion are hidden) and concrete (optimal) implementations burdened by the administrative rules. This is also a didactic tool since the graphical interpreter in² for interaction nets [13] can be used to visualize λ -terms and their reduction. In the previous section, we give an example of application by giving the implementation of left reduction. Another example is to extend the translation to the λ_c -calculus (with Felleisen *CC* operator) where the whole stack may be duplicated during reduction. We believe that those ideas can be used for other λ -calculi and reduction strategies. However, except for the standard strategies (head and left reduction) and small variations of the λ -calculus, this way has not been investigated yet.

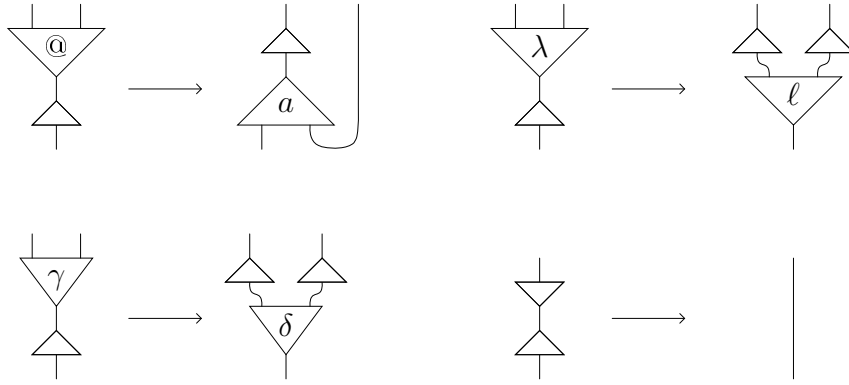


Figure 11: *Decoding rules*

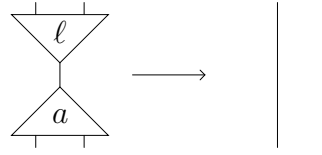


Figure 12: β -reduction rule

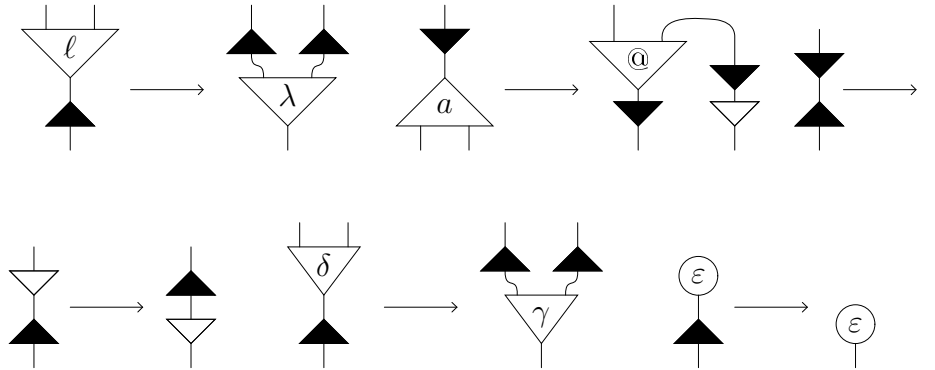


Figure 13: *Encoding rules*

A Correctness of the Naive Krivine machine

We prove the correctness of the Krivine machine and use the definitions introduced in section 3. We first define the translation of a configuration into a (closed) λ -term and then prove that weak head reduction is correctly simulated.

A.1 Translation of a configuration into a λ -term

The translation of a closure is obtained simply by substituting all free occurrences of variables by their corresponding values stored in the environment.

Definition A.1 (Translation of closures) *For any closure T , we define the λ -term associated to T , noted \overline{T} as follows:*

- $\overline{*} = *$
- $\overline{(t, \langle \overline{T_1}, \dots, \overline{T_\ell} \rangle)} = t[\overline{T_1}/x_1] \dots [\overline{T_\ell}/x_\ell] = t[\overline{T_1}/x_1, \dots, \overline{T_\ell}/x_\ell]$

Remark A.1 *The translation of a closure is a closed λ -term (we consider that $*$ is closed). So the order in which substitutions are performed to compute \overline{T} is not important and this definition is indeed correct (see [6]).*

Remark A.2 *Let a closure $T = (t, E)$. If t is a λ -abstraction (resp. an application), so is \overline{T} . The converse is false; for example, if $t = x_1$ and $\overline{E(x_1)}$ is a λ -abstraction, \overline{T} is a λ -abstraction.*

Remark A.3 *By definition of substitution, $\overline{((u)v, E)} = (\overline{(u, E)})\overline{(v, E)}$*

Now we can define the translation of a configuration. All closures are translated and the resulting λ -terms are put together in several (left associative) applications. In other words those λ -terms are the arguments of the *spinal branch* (left branch) of the λ -term corresponding to the configuration.

Definition A.2 (Translation of a configuration) *Let $\mathcal{C} = (T, P)$ a configuration where $P = T_1 \dots T_n$. The translation of \mathcal{C} , noted $\overline{\mathcal{C}}$ is $(\overline{T})\overline{T_1} \dots \overline{T_n}$.*

A.2 Weak head reduction

Let us first recall the definition of weak head reduction (noted $>$) by the following two inference rules:

$$\frac{}{(\lambda x_i u)v > u[v/x_i]} \beta \quad \frac{u > u'}{(u)v > (u')v} a$$

We first prove three simple lemmas and then give a concise proof of the correctness in proposition A.1. Lemma A.1 ensures that we can obtain a configuration $((t, E), P)$ where t is an application or abstraction by applying multiple *eval* transitions.

Lemma A.1 (substitution) *Let a configuration $\mathcal{C} = (R, P)$. There exists a closure S such that $(R, P) \xrightarrow{*} (S, P)$, $S \neq (x_i, E)$ and $\overline{R} = \overline{S}$.*

Proof. By induction on closure R :

- If $R \neq (x_i, E)$, we take $S = R$.

- Otherwise, $R = (x_i, E)$. Thus $\mathcal{C} \rightarrow (E(x_i), P)$ and we apply the induction hypothesis on $(E(x_i), P)$.

Lemma A.2 states that we obtain the same transitions if we add a closure at the bottom of the stack in the initial configuration. This ensures that we can perform all the transitions with a non empty stack.

Lemma A.2 (non empty stack) *Let configurations (U, P) and (U', P') and closure V .*

$$(U, P) \xrightarrow{*} (U', P') \iff \begin{cases} (U, PV) \xrightarrow{*} (U', P'V) \\ \text{The stack is never empty during the transition} \end{cases}$$

Proof. By induction on the length of the reductions.

Lemma A.3 ensures that for any configuration corresponding to an application $(u)v$ we can eventually reach a configuration corresponding to the same λ -term where the argument v is at the bottom of the stack.

Lemma A.3 (push) *Let a configuration $\mathcal{C} = (R, P)$ and some λ -terms u and v such that $\overline{\mathcal{C}} = (u)v$. There are a configuration $\mathcal{D} = (S, Q)$ and a closure V such that: $\mathcal{C} \xrightarrow{*} (S, QV)$, $\overline{(S, Q)} = u$ and $\overline{V} = v$.*

Proof.

- If $P = 1$, by lemma A.1 (substitution) and remarks A.2 and A.3, $(R, 1) \xrightarrow{*} (((u_0)v_0, E), 1)$ where u_0 and v_0 are λ -terms and E an environment such that $u = \overline{(u_0, E)}$ and $v = \overline{(v_0, E)}$. We take $\mathcal{D} = ((u_0, E), 1)$ and $V = (v_0, E)$.
- Otherwise, $P = T_1 \dots T_n T_{n+1}$. Thus $\overline{\mathcal{C}} = (\overline{R})\overline{T_1} \dots \overline{T_n} \overline{T_{n+1}}$. So we have $\overline{T_{n+1}} = v$. We take $\mathcal{D} = (R, T_1 \dots T_n)$ and $V = T_{n+1}$.

Now we can prove that weak head reduction on closed λ -terms is correctly simulated by the Krivine Machine. Remark (case β) that this proof is correct mainly because closures are closed λ -terms. So free occurrences of variables cannot be captured when performing substitution.

Proposition A.1 (completeness) *Let two closed λ -terms t and t' such that $t > t'$ and a configuration \mathcal{C} such that $\overline{\mathcal{C}} = t$. There is a configuration \mathcal{C}' such that $\mathcal{C} \xrightarrow{*} \mathcal{C}'$ and $\overline{\mathcal{C}'} = t'$.*

Proof. By induction on the length of the deduction of $t > t'$. Let us consider the last rule that has been used:

a $t = (u)v$ and $t' = (u')v$.

By lemma A.3 (push), there are a configuration (T, P) and a closure V such that $\overline{(T, P)} = u$, $\overline{V} = v$ and $\mathcal{C} \xrightarrow{*} (T, PV)$. By induction, there is a configuration (T', P') such that $(T, P) \xrightarrow{*} (T', P')$ and $\overline{(T', P')} = u'$. By lemma A.2 (non empty stack), $(T, PV) \xrightarrow{*} (T', P'V)$. Thus we take $\mathcal{C}' = (T', P'V)$.

β For example, $t = (\lambda x_1 u)v$ and $t' = u[v/x_1]$.

By lemma A.3 (push), there is a configuration (R, P) and a closure V such that $\mathcal{C} \xrightarrow{*} (R, PV)$, $\overline{(R, P)} = \lambda x_1 u$ and $\overline{V} = v$. $P = 1$ (otherwise $\overline{(R, P)}$ would be an application). By lemma A.1 and remark A.2, there is a closure S such that $(R, V) \xrightarrow{*} (S, V)$, $\overline{R} = \overline{S}$ and

$S = (\lambda x_1 w, E)$ where $E = \langle T_1, \dots, T_\ell \rangle$ is an environment. We take $\mathcal{C}' = ((w, E[x_1 := V]), 1)$. We indeed have $\mathcal{C} \rightarrow \mathcal{C}'$.

Let us prove that $\overline{\mathcal{C}'} = u[v/x_1]$.

$$\overline{S} = (\lambda x_1 w)[\overline{T_1}/x_1, \dots, \overline{T_\ell}/x_\ell] = \lambda x_1 w[\overline{T_2}/x_2, \dots, \overline{T_\ell}/x_\ell] = \lambda x_1 u.$$

Thus $w[\overline{T_2}/x_2, \dots, \overline{T_\ell}/x_\ell] = u$.

$$\overline{\mathcal{C}'} = w[\overline{V}/x_1, \overline{T_2}/x_2, \dots, \overline{T_\ell}/x_\ell] = w[\overline{T_2}/x_2, \dots, \overline{T_\ell}/x_\ell][\overline{V}/x_1] = u[\overline{V}/x_1].$$

Of course, we can prove that the Krivine Machine implements only weak head reduction.

Proposition A.2 (correctness) *For any configurations \mathcal{C} and \mathcal{C}' such that $\mathcal{C} \rightarrow \mathcal{C}'$, we have $\overline{\mathcal{C}} = \overline{\mathcal{C}'}$ or $\overline{\mathcal{C}} > \overline{\mathcal{C}'}$*

Proof. It follows from the following remarks. First, the Krivine Machine is deterministic: for any configuration, one (at most) transition can be applied. Second, only *pop* transitions change the corresponding λ -term and it cannot be applied on configurations corresponding to weak head normal forms.

References

- [1] Andrea Asperti and Stefano Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
- [2] Vincent Danos and Laurent Regnier. Reversible, Irreversible and Optimal λ -machines. *Theoretical Computer Science*, 227, 1999.
- [3] Jean-Yves Girard. Linear logic. *Theoretical Computer Science* 50, 50(1):1–102, 1987.
- [4] Georges Gonthier, Martin Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages (POPL'92)*, ACM Press., pages 15–26, 1992.
- [5] Jean-Louis Krivine. Un interpréteur pour le λ -calcul. Technical report, Notes de cours de DEA, Université de Paris 7, 1985.
- [6] Jean-Louis Krivine. *Lambda-calculus, types and models*. Ellis Horwood Series in Computers and their Applications., 1993. Transl. from the French by Rene Cori. (English).
- [7] Yves Lafont. Interaction nets. In *proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, Orlando (Fla., USA)*, pages 95–108, 1990.
- [8] Yves Lafont. From proof-nets to interaction nets. In *Advances in Linear Logic*, London Mathematical Society Lecture Note Series 222. Cambridge University Press, 1995.
- [9] Yves Lafont. Interaction combinators. *Information and Computation*, 137(1):69–101, 1997.
- [10] John Lamping. An algorithm for optimal lambda-calculus reduction. In *proceedings of the 17th Annual ACM Symposium on Principles of Programming Languages, Orlando (Fla., USA)*, 1990.
- [11] Frédéric Lang. *Modèles de la β -réduction pour les implantations*. PhD thesis, Ecole normale supérieure de Lyon, 1998.
- [12] Sylvain Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structures in Computer Science*, 12(6), December 2002.
- [13] Sylvain Lippi. in²: a graphical interpreter for the interaction nets. In *Proceedings of Rewriting Techniques and Applications (RTA '02)*. Springer Verlag, 2002.
- [14] Sylvain Lippi. *Théorie et pratique des réseaux d'interaction*. PhD thesis, Université de la méditerranée, 2002.
- [15] Ian Mackie. *The geometry of implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, 1994.
- [16] Ian Mackie. Yale: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*. ACM Press, 1998.

- [17] Ian Mackie and Jorge Sousa Pinto. Encoding linear logic with interaction combinators. *Information and Computation*, 176:153–186, 2002.
- [18] Jorge Sousa Pinto. Weak reduction and garbage collection in interaction nets. In *Proceedings of the 3rd Int'l Workshop on Reduction Strategies in Rewriting and Programming*. In B. Gramlich and S. Lucas, editors, 2003.