ROMA TRE
UNIVERSITÀ DEGLI STUDI

FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI
GRADUATION THESIS IN MATHEMATICS

# An Implementation of Lambda Calculus on Graphic Cards — Abstract

*Author:*
Giulio Pellitta
289286

*Supervisor:*
Prof. Marco Pedicini

Academic year: 2009/2010

MSC AMS: 03B70, 03F05, 03B40, 68N18, 68Y05.
KEYWORDS: lambda calculus, optimal reduction, linear logic, geometry of interaction, parallel and distributed computation.

# 1 Lambda Calculus

Lambda Calculus is a formal language created by Alonzo Church in 1936. It is equivalent to other computational models such as Turing's machine and recursive functions. For a long time it had only a theoretical interest, then with the formulation of the Curry-Howard isomorphism and Levy-optimality its computational side clearly surfaced.

**Definition 1.** *Let $V = \{x_i | i \in \mathbb{N}\}$ be a denumerable set of variables. We define the set of $\lambda$-terms as the set $L$ of finite sequences of symbols "(", ")", "$\lambda$" and variables $x_i$ obtained applying the following rules a finite number of times*

- *if $x \in V$ then $x \in L$;*
- *if $t, u \in L$ then $(u)t \in L$;*
- *if $x \in V, u \in L$ then $\lambda x\, u \in L$.*

**Definition 2** (Free variables)**.** *We define the set of* free occurrences *of a variable $x$ in $t$:*

- *if $t = y$, $x$ occurs free in $t$ iff $x = y$;*
- *if $t = (u)v$, the free occurrences of $x$ are those in $u$ and $v$;*
- *if $t = \lambda y\, u$ then the free occurrences of $x$ are those in $u$ if $x \neq y$ and none otherwise.*

*We denote $FV(t)$ the set of* free variables *of $t$, i.e. those variables that have at least one free occurrence in $t$. Given a set of terms $\{t_1, \ldots, t_k\}$ we define $FV(\{t_1, \ldots, t_k\}) := \cup_i FV(t_i)$.*

**Definition 3** (Bound variables)**.** *We define $BV(t)$, the set of* bound variables *of $t$:*

- *if $t = x$, $BV(t) = \emptyset$;*
- *if $t = (u)v$, $BV(t) = BV(u) \cup BV(v)$;*
- *if $t = \lambda x\, u$, $BV(t) = BV(u) \cup \{x\}$.*

*A term containing only bound variables is said to be* closed *(and* open *otherwise).*

## 1.1 $\alpha$-equivalence

If two terms differ only in the name of the bound variables (such as $\lambda x\, x$ and $\lambda y\, y$) they actually represent the same thing. So $\alpha$-equivalence is introduced to identify such terms. Let $u, t_1, \ldots, t_k \in L$ and $x_1, \ldots, x_k \in V$ s.t. $\forall 1 \leq i \neq j \leq k : x_i \neq x_j$; we define $u\langle t_1/x_1, \ldots, t_k/x_k\rangle$ as the result of the substitution of $t_i$ in place of every free occurrence of $x_i$ in $u$ ($1 \leq i \leq k$). We define it by induction on $u$ according to the following rules:

- $u = x \in V \Rightarrow u\langle t_1/x_1, \ldots, t_k/x_k\rangle = \begin{cases} t_i & \text{if exists } i \text{ s.t. } x = x_i, \\ & i \in \{1, \ldots, k\} \\ x & \text{if for all } i\ x \neq x_i, \\ & i \in \{1, \ldots, k\}; \end{cases}$
- $u = (w)v \Rightarrow u\langle t_1/x_1, \ldots, t_k/x_k\rangle =$
  $(w\langle t_1/x_1, \ldots, t_k/x_k\rangle)v\langle t_1/x_1, \ldots, t_k/x_k\rangle$;

- $u = \lambda x\, v \Rightarrow u\langle t_1/x_1, \ldots, t_k/x_k\rangle =$
$$\begin{cases} \lambda x\, v\langle t_1/x_1, \ldots, t_{i-1}/x_{i-1}, t_{i+1}/x_{i+1}, \ldots, t_k/x_k\rangle & \text{if exists } i \text{ s.t. } x = x_i, \\ & i \in \{1, \ldots, k\} \\ \lambda x\, v\langle t_1/x_1, \ldots, t_k/x_k\rangle & \text{if for all } i\ x \neq x_i, \\ & i \in \{1, \ldots, k\}. \end{cases}$$

We call such substitution "simple substitution", to tell it apart from the on we define later. The order in which the substitutions are executed is irrelevant.

**Definition 4** ($\alpha$-equivalence)**.** *We define on the set $L$ of $\lambda$-terms the following equivalence relation $\equiv$:*

- *if $u \in V$ then $u \equiv u' \Leftrightarrow u = u'$;*
- *if $u = (w)v$ then $u \equiv u' \Leftrightarrow u' = (w')v', v \equiv v', w \equiv w'$;*
- *if $u = \lambda x\, v$ then $u \equiv u' \Leftrightarrow u' = \lambda x'\, v'$, with $v\langle y/x\rangle \equiv v'\langle y/x'\rangle$ for each $y \in V$ except a finite number.*

**Definition 5.** *Let $u, t_1, \ldots, t_k \in \Lambda$, $x_1, \ldots, x_k \in V$. We define the term $u[t_1/x_1, \ldots, t_k/x_k] := u'\langle t_1/x_1, \ldots, t_k/x_k\rangle$ where $u' \equiv u$ is such that $BV(u') \cap FV(\{t_1, \ldots, t_k\}) = \emptyset$.*

## 1.2 $\beta$-equivalence

Turing's machines operates with a transitions table that defines how to go from one configuration to the next, finally reaching the result. Lambda calculus has a simple transition mechanism called $\beta$-reduction. A term $t = (\lambda x\, u)v$ (an abstraction followed by an application) is called a *redex*: a $\beta$-reduction step consists in replacing the term with $t' = u[v/x]$.

**Definition 6.** *A term of the form $(\lambda x\, u)t$ is called a* redex*, $u[t/x]$ is called its* contractum*. We define a binary relation $\beta_0$ on $\Lambda$; we have $t\beta_0 t'$ if $t'$ is obtained by contracting a redex (or by a $\beta$-*reduction[1]*) in $t$.*

- *if $t \in V$, $t\beta_0 t'$ is false for any $t'$;*
- *if $t = \lambda x\, u$, then $t\beta_0 t'$ iff $t' = \lambda x\, u'$, with $u\beta_0 u'$;*
- *if $t = (u)v$, then $t\beta_0 t'$ iff*
  - *either $t' = (u)v'$ (resp. $t' = (u')v$) with $v\beta_0 v'$ (resp. $u\beta_0 u'$),*
  - *or $u = \lambda x\, w$ and $t' = w[v/x]$.*

**Definition 7.** *The transitive closure of $\beta_0$ is denoted $\beta$:*

$$t\beta t' \Leftrightarrow \exists n \in \mathbb{N}, t = t_0, t_1, \ldots, t_{n-1}, t_n = t' \in \Lambda,$$

*where $t_i\beta_0 t_{i+1}$ for all $i$ s.t. $0 \leq i < n$.*

[1] Contracting a single redex is often called *$\beta$-conversion*, while contracting any number of redexes is called *$\beta$-reduction*.

With each step, we try to simplify a term into an *explicit* form: a term that contains no redexes and can no longer be simpified (such a term is called *normal*). Not every term has a normal form; there are even *normalizable* terms such that a particular reduction chain can go on forever without reaching the normal form (these terms are normalizable but not *strongly* normalizable). However, we do know that if the normal form exists it is unique.

### 1.2.1 Confluence: Church-Rosser

In general a term contains several redexes, so the computation is not deterministic. However, the calculus is confluent: if two terms $u$ and $w$ are obtained from $t$ through a series of $\beta$-reductions, then there exists a term $w$ such that both $u$ and $v$ can be reduced to it.

**Theorem 1** (Church-Rosser)**.** *The $\beta$-conversion has the Church-Rosser property.*

Simplifying a single redex is called *contraction* ($t \to_{\beta_0} t'$), while the simplification of any number of redexes is called *reduction* ($t = t_0 \to_{\beta_0} t_1 \to_{\beta_0} \dots \to_{\beta_0} t_{n-1} \to_{\beta_0} t_n = t'$ can be written equivalently $t \to_\beta t'$ if we do not need to specify all the intermediate steps).

**Definition 8.** *The* $\beta$-equivalence *(denoted by $\simeq_\beta$) is defined as the least equivalence relation which contains $\beta_0$ (or equivalently $\beta$, its transitive closure).*

$$t \simeq_\beta t' \Leftrightarrow \textit{ exists } n \in \mathbb{N}, (t = t_0), t_1, \dots, t_{n-1}, (t_n = t'),$$
$$\textit{such that } t_i \beta_0 t_{i+1} \textit{ or } t_{i+1} \beta_0 t_i \forall 0 \leq i < n.$$

## 1.3 Representation of recursive functions

As we said, lambda calculus is a *formal* language: there are no functions or integers, not as we know them. Still, there are terms that can represent — in a *precise* sense — a given function. Similarly, integers (as well as booleans and other data types) can be encoded as particular terms which have the "right" operational behaviour.

### 1.3.1 Church integers

Let $(t)^k u$ denote $\underbrace{(t)\dots(t)}_{k} u$ if $k > 0$ and $u$ if $k = 0$. We define the term $\underline{k} = \lambda f\ \lambda x\ (f)^k x$; $\underline{k}$ is called the integer $k$ of $\lambda$-calculus (or Church integer). The term represents a generic function $f$ being applied $k$ times on some term $x$.
Note that $(f)^a (f)^b = (f)^{a+b}$, where $f$ is a generic function and $a, b$ are (non-negative) integers.

### 1.3.2 Function representation

**Definition 9.** *Let $\varphi : \mathbb{N}^n \to \mathbb{N}$ be a partial function. Given the $\lambda$-term $\Phi$, we say* $\Phi$ represent *(resp.* strongly represent*) $\varphi$ if, for any $k_1, \dots, k_n \in \mathbb{N}$:*

- *if $\varphi(k_1, \dots, k_n)$ is undefined, then $(\Phi)\underline{k_1} \dots \underline{k_n}$ is not normalizable (resp. not solvable);*
- *if $\varphi(k_1, \dots, k_n) = k$, then $(\Phi)\underline{k_1} \dots \underline{k_n}$ is $\beta$-equivalent to $\underline{k}$.*

*If $\varphi$ is a total function, the two notions coincide.*

**Theorem 2.** *Every recursive partial function from $\mathbb{N}^k$ to $\mathbb{N}$ is (strongly) representable by a term of $\lambda$-calculus.*

# 2 Optimal reduction

In spite of the progress made, for years functional languages had only limited use because of the difficulty of designing an efficient implementation. Lévy defined optimality as avoiding duplication of work (i.e., avoiding the duplication of redexes), but did not provide any practical way to do optimal reductions. Ten years later Lamping designed the first optimal algorithm that avoided duplication using an explicit mechanism for sharing of subexpressions.

## 2.1 Reduction strategies

A redex is to the *left* of another redex if its lambda abstractor appears further to the left. Now let us consider two possible reduction strategies, corresponding to opposite viewpoints:

i. *leftmost innermost* — the leftmost redex not containing any other redex is contracted;

ii. *leftmost outermost* — the leftmost redex not contained in any other redex is contracted.

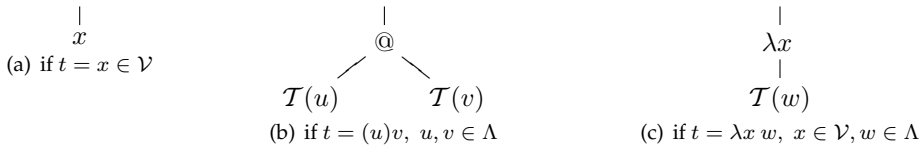
In the first case, the arguments are evaluated before the function, while in the latter the order is reversed. Which strategies permits to reach the normal form in a minimal number of $\beta$-conversions?

## 2.2 Sharing of expressions

A good reduction strategy should be able to avoid duplication any redexes, even a *virtual redex* (i.e., a sub-term which is not yet redex, but that will be after a substitution). In general this is not possible because there are terms for which any reduction strategy duplicate some redex (in general it is not even possible to predict which strategy causes the least duplication of work).

$\lambda$-terms have also a graphical representation. It is useful to study (virtual) redexes and is the first step for leaving the world of usual $\lambda$-terms and start thinking in term of graphs and paths.

**Definition 10.** *Given a $\lambda$-term $t$, we define its* abstract syntax tree *$\mathcal{T}(t)$ as:*




(a) if $t = x \in \mathcal{V}$ (b) if $t = (u)v,\ u, v \in \Lambda$ (c) if $t = \lambda x\, w,\ x \in \mathcal{V}, w \in \Lambda$

### 2.2.1 Wadsworth's algorithm

In his PhD [Wad71], Wadsworth described a graph reduction technique[2] that used directed acyclic graphs similar to syntax trees to represent $\lambda$-terms. The problem of sharing is addressed by the use of pointers: identical subexpression are represented with a single piece of graph, so the reduction of $(\lambda x\ M)N$ requires to link $N$ to every occurrence of $x$ in $M$.[3]

**Example 1.** *Let $t = (\lambda x\ (x)x)(\lambda x\ y)I$ and $t' = (y)y$ its normal form. The reduction is shown in Figure 1: notice that the initial graph is a normal syntax tree, while the final graph is not.*

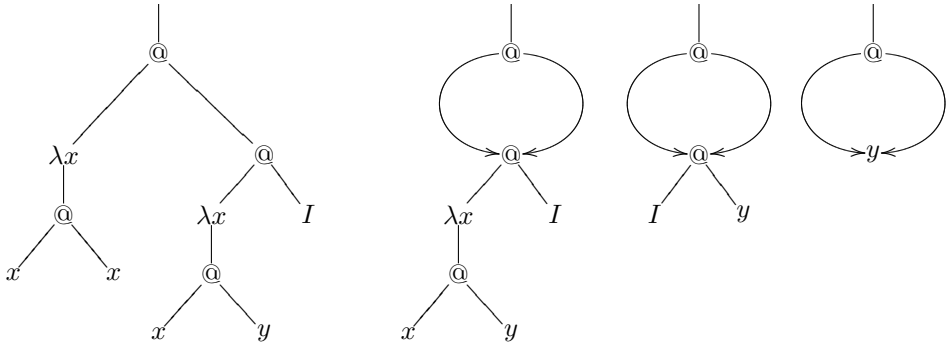



Figure 1: Reduction of $(\lambda x\ x)(\lambda x\ y)I$.

## 2.3 Optimality

Lévy generalized the concept of reduction to include *parallel reductions* (i.e., in which several redexes are contracted simultaneously). Lévy showed with an example [Lév78, pag. 199] that in general parallel reductions are essential to avoid duplications. Since parallel reductions are the basis of optimal reduction, he said we can not assume that each simultaneous reduction has the same cost. Although he did not know of a structure better than Wadsworth's to back his reasonings, he suggested representing several redexes with the same object (this is the case in Lamping's sharing graphs).

## 2.4 Lamping's algorithm

As an introduction to the actual algorithm, we present here the main facts on sharing graphs and the reduction rules of a simplified version. Finally we also briefly discuss the full algorithm.

[2] Cfr. also [Lév78, pp. 195–198] and [AG98, pp. 20–22].

[3] This way, any simplification inside $N$ corresponds to multiple simplifications to each copy in the corresponding $\lambda$-term.

### 2.4.1 Sharing graphs

Lamping technique employs an explicit node for sharing, called *fan*. The node has three ports, one *principal port* (or *main port*) and two *auxiliary ports* (or *secondary ports*) denoted $\circ$ and $*$: the idea is that when a path starting from the root of the term meets a fan it enters from the principal port and exists by one of the secondary ports, each corresponding to a different copy of a shared sub-term. Contrarily to Wadsworth's case, sharing of expressions with the same "structure" but different sub-expressions (such as $((M)N)(M)N'$) is allowed (a fact essential for *optimal* sharing): this is obtained by the use of two paired fans, the first (called *fan-in*) is used to denote sharing, the second (called *fan-out*) is used to denote unsharing. Informally, a fan-in (resp. fan-out) marks the start (resp. the end) of a shared sub-expression.

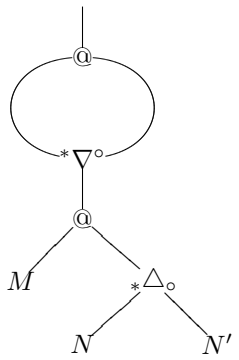



Figure 2: The sharing graph of $((M)N)(M)N'$.


Figure 3: Principal ports.

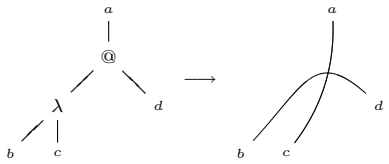
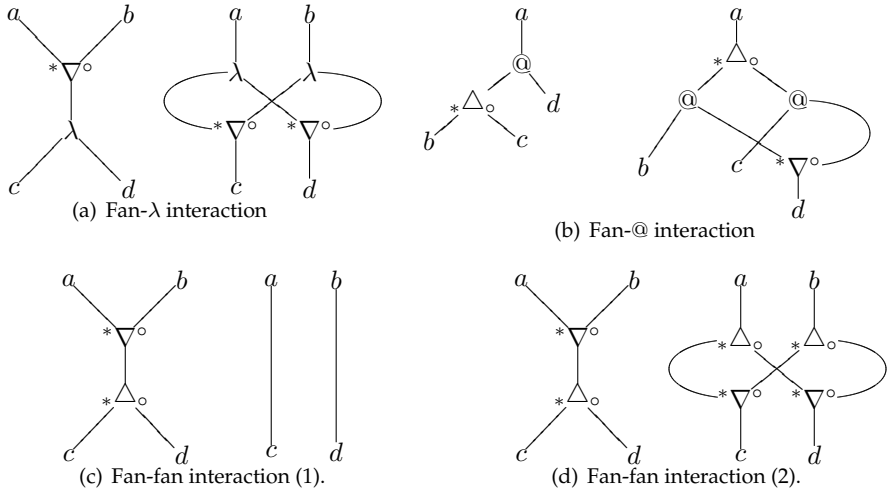



Figure 4: $\beta$-rule




(a) Fan-$\lambda$ interaction

(b) Fan-@ interaction

(c) Fan-fan interaction (1).

(d) Fan-fan interaction (2).

Figure 5: Fan-interaction rules (the rewriting is from left to right).

**Reduction rules** Another difference with Wadsworth's graphs is that each variable is linked to the corresponding $\lambda$ node. The fan-interaction rules presented here are optimal (in the sense that they do not duplicate redexes of the same family), and they are interaction rules (meaning two nodes can interact only if they are linked by their principal ports).

### 2.4.2 Full Algorithm

We have omitted how to determine which rule should be used when two fans meet. This is the problem of pairing fans (if two fans match the fans cancel each other out, if they do not the fans duplicate each other). To solve this problem it is necessary to introduce two types of control nodes for managing levels of sharing.

**Control nodes** The control nodes in Figure 6 are called croissant and bracket. The role of a croissant (resp. bracket) node is to open (resp. temporarily close) a level.


Figure 6: Control nodes.

**Contexts** A context is a stacks of levels: it is used to describe paths in a graph keeping track of the auxiliary ports ($*$ or $\circ$) traveled at each fan (it is possible to describe the syntax tree of a lambda by the sets of its maximal paths — those starting at the root and ending at the nodes of the tree — similarly to how a proof-net may be described by its execution). Since a fan may be crossed multiple times, the information must be organized properly.
We may plug a context into the hole $[\cdot]$

$$A^n[C] = \langle\langle \dots \langle C, a_{n-1}\rangle, \dots, a_1\rangle, a_0\rangle$$

where $C$ is any context (i.e., $A^n[C] = A$ iff $C$ is the subcontext of $A$ at level $n$). With the notations we introduced, Figure 7 shows how a context is modified upon entering a fan from an auxiliary door. If a croissant (resp. bracket) is traversed, contexts are modified as

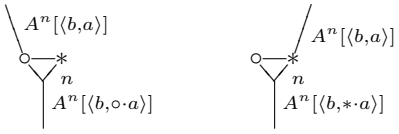



Figure 7: Context transformation (1).

in Figure 8 When a croissant is met a new level is opened, so all existing information is


Figure 8: Context transformation (2).

pushed at a higher level and the level $n$ is initialized to the empty list $\square$. When a bracket is traversed, it is necessary to store all branching information recorded for level $n$ in such a way to retrieve it later when the matching bracket is found along the path. There is no need to record any information when an application (resp. abstraction) node is met.

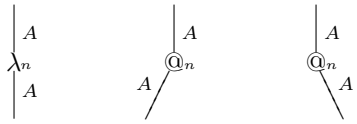



Figure 9: Context transformation (3).

**Correctness**

**Definition 11.** *A* proper path *in a sharing graph $G$ is a path such that:*

1. *every edge of the path is labeled with a context;*

2. *consecutive pairs of edges satisfy one of the constraints in Figures 7, 8, 9.*

3. *the path does not contain bounces.*

The concept of proper path is the equivalent of *straight path* in Geometry of Interaction. It leads to the following result:

**Theorem 3.** *For any sharing graph reduction $[M] \to G$, there exists a corresponding $\lambda$-term reduction $M \to_\beta N$ such that $G$ matches $N$.*

The previous result is described in Figure 10.

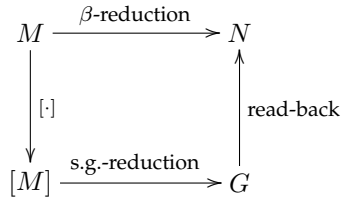



Figure 10: Correctness of the sharing graph reduction.

# 3 Linear Logic

In Intuitionistic Logic proofs may not have several conclusions and we lose the symmetry of the Classical Logic sequent. Linear Logic is an answer to both these problem: it has the symmetry of Classical Logic and the constructiveness of Intuitionistic Logic.
The word linear means that each formula may only be used once. Unless that formula has the form $?A$ or $!A$, where $?$ and $!$ denote two special unary connectives called *exponentials*.

$$(!A)^\perp = ?A^\perp \qquad\qquad (?A)^\perp = !A^\perp.$$

Here is how structural rules become with the introduction of exponentials.

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\,! \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\,W? \qquad \frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\,C? \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\,D?$$

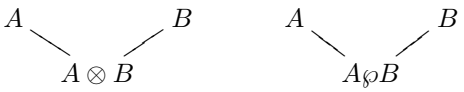
## 3.1 Proof-nets

As previously stated, in Linear Logic it is possible to have proofs with more than one conclusion. It is possible to represent them graphically (analogously to intuitionistic proofs in natural deduction, but without the one-conclusion-only limitation) with special graphs called proof-nets. We examine them in the framework of MLL and MELL (resp. multiplicative and multiplicative-exponential linear logic).

### 3.1.1 MLL

**Definition 12** (Symbol)**.** *A* symbol *(or* link[4]*) is a triple* $(l, a, p)$ *where* $l$ *is the* label *of the symbol,* $a$ *is a finite sequence of* auxiliary ports *or* premises *and* $p$ *is a finite sequence of* principal ports *or* conclusions*. The number of premises (resp. conclusions) is the arity (resp. co-arity) of the symbol. Multiplicative symbols are* axiom, cut, tensor *and* par *and they are shown in Figures 11(a) and 11(b) along with their labels, arity and co-arity.*


(a) Identity group: Ax/Cut




(b) Logical group: $\otimes/\wp$

Figure 11: MLL rules

### 3.1.2 MELL

To extend MLL by including the exponential fragment of linear logic, it is necessary to introduce the concept of *proof-box*. A proof-box (or simply *box*) contains a complete proof-net and is employed when exponential links are involved, since it marks a subnet that may be erased or duplicated. Since boxes introduce sequentiality (in contrast with the parallel nature of proof-nets), one must try to use them as little as possible: although avoiding the ! case (promotion link) is not possible [Gir87, sec 2.6], a reformulation of proof-nets may allow some improvements of the syntax.

**Definition 13.** *A MELL net* $\mathcal{N}$ *is said to be correct if*

- *every box contains a correct sub-net, and*
- *every switch graph* $\mathcal{N}^s$ *derived from* $\mathcal{N}$ *is acyclic and connected.*

### 3.1.3 Cut-elimination

It is possible to define rules for proof-normalization directly in the framework of proof-nets. A step of normalization usually consists of "rewiring", unless proof-box are involved. Encoding terms as proof-nets, cut-elimination can also be applied to lambda calculus.
Cut-elimination is only the trace of a deeper procedure: we need an operational semantics of linear logic to model execution at a lower level, simple enough to be mechanized someway (on an actual, concrete machine). This semantics was discovered by Girard [Gir89b] [Gir89a] and we explain it in the next section (along with some implementation techniques).

[4] A link is a node, not an edge.

# 4 Geometry of Interaction

We are interested in some way to obtain the normal form of a term rather than implementing $\beta$-reduction directly. $\lambda$-terms may be represented as special graphs called *directed virtual nets*. The normal form can be computed considering maximal paths starting from the root of the graph whose weight is not zero; the paths whose weight is non-zero are said to be *regular*. On the other hand, the paths that survive reduction are said to be *persistent*. Those two notions have been proved to be equivalent [DR95], so we actually compute the weight of the paths by performing reductions.[5] Indeed, Geometry of Interaction is essentially an algebraic tool for computing the weights of persistent paths. Its name reflects the fact that it uses geometric (or algebraic) tools to deal with (logical) computations: in other words, it offers a logical foundation for graph-rewriting techniques.
We start within a proof-nets framework, but then we move onto the more general directed virtual nets and we finally proceed discussing the techniques to implement.

**Definition 14** (Straight path)**.** *A path $\gamma = t_1 \dots t_n$ is said to be* straight *if it is:*

**non-twisting** *if the goal of $t_i$ and the source of $t_{i+1}$ are two premise ports of a symbol $X$ then $X$ is a cut symbol (changing direction is only allowed at* Cut *or* Ax *symbols, otherwise one always moves upward or always downward);*

**non-bouncing** *if for any $1 \leq i < n$ we have $t_i \neq t_{i+1}^*$ (two consecutive edge-traversals can't be identical except for the direction they're traversed).*

Proof nets can be labeled with elements from Girard's algebra $L^*$ (with morphism $!(\cdot)$ and inversion $(\cdot)^*$), which is characterized by the following equations.

$$x^* y = \delta_{xy}, \quad x, y = p, q, w_i, \tag{1}$$

$$!(u) w_i = w_i !^{e_i}(u), \tag{2}$$

Orienting these equation from left to right, we get a rewriting system. An element of $L^*$ can be written in the form $a^* b$ (known as *stable form*) iff it is not zero. A proof-net $\mathcal{N}$ is characterized by the following invariant, called *Girard's Execution Formula*, where the sum is extended to all maximal paths and $\Pi^\bullet$ and $\sigma$ are two matrices (whose entries are operators on the Hilbert Space $l^2$) representing $\mathcal{N}$.

**Definition 15.**

$$\mathcal{E}x(\Pi^\bullet, \sigma) = (1 - \sigma^2) \left( \Pi^\bullet \sum_{k=0}^{\infty} (\sigma \Pi^\bullet)^k \right) (1 - \sigma^2) \tag{3}$$

To compute the normal form of a proof-net one can calculate its execution, which is the sum of all regular maximal straight paths (or equivalently of all maximal straight paths that survive reduction).
We already said that a path is regular iff it is persistent, so if we consider regular path we can work directly on the weighted graphs (called virtual nets) which can be obtained from a proof-net or a lambda-term. The concepts of (straight) paths and execution are still the key.

[5] There are even more equivalent notions, cfr. [ALDR94] for a recap.

## 4.1 Virtual Reduction

Virtual Reduction is a local graph rewriting introduced in [DR93] in which we compose two consecutive edges in the resulting path (hence *shortening* the path). A VR step is described in the following figure, where $[x] = 1 - xx^*$ is called the *filter* of $x$.

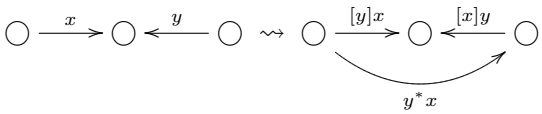



Figure 12: A step of VR.

**Definition 16** (Straight path)**.** *A path is* straight *if it contains no subpath of the form $\phi\phi^*$.*

**Definition 17.** *A weighted directed graph is said to be* split *if any three coincident edges $\phi_1, \phi_2, \phi_3$ are such that $\langle\phi_1\rangle\langle\phi_2\rangle\langle\phi_3\rangle = 0$; it is said to be* square-free *if for any straight path $\phi$, $\phi\phi = 0$.*

Informally, splitness prevents any interference between compositions while square-freedom prevents the existence of self-recomposable cycles[6].

**Definition 18.** *A weighted directed graph is a* virtual net *if it is split and square-free.*

If $V$ is a virtual net and $V'$ is obtained by $V$ with a step of VR it is still a virtual net and $\mathcal{E}x(V) = \mathcal{E}x(V')$. VR can be applied to lambda calculus encoding lambda terms as virtual nets. Since by definition VR creates weight of increasing order (such as $[x[y]], z[x[y]]$ and so on), Directed Virtual Reduction is introduced.

## 4.2 Directed Virtual Reduction

**Notation.** *Set $[b_1, \dots, b_n] := 1 - b_1 b_1^* - \cdots - b_n b_n^*$. The filter is an idempotent iff $\langle b_i {b_i}^*\rangle \langle b_j {b_j}^*\rangle = 0$ for $1 \leq i \neq j \leq n$ (i.e. the $b_i$'s are pairwise orthogonal). Let $\alpha = [b_1, \dots, b_n]a$: we denote $\alpha^+$ the weight without filter, that is $a$.*

**Definition 19.** *A* directed virtual net *$V$ is an acyclic[7] virtual net such that for each edge $\alpha$:*

**A.** *$\alpha = [b_1, \dots, b_n]a$, where $a, b_1, \dots, b_n \in L^*$ are positive.*

**B.** *for any $i \neq j$ and $\beta_1, \beta_2$ counter-edges of $\alpha$ along $\tau_1, \tau_2$:*

$$\langle b_i\rangle\langle b_j\rangle = 0 \qquad 0^R(\alpha; b_i; b_j)$$
$$\langle b_i\rangle\langle \tau^*\beta^+\rangle = 0 \qquad 1^R(\alpha; b_i; \tau^*\beta)$$
$$\langle \tau_1^*\beta_1^+\rangle\langle \tau_2^*\beta_2^+\rangle = 0 \qquad 2^R(\alpha; \tau_1^*\beta_1; \tau_2^*\beta_2)$$

Contrarily to VR, in directed virtual reduction the weight of the composition (in stable form) is written on two new edges (one for the positive part, the other for the negative part).

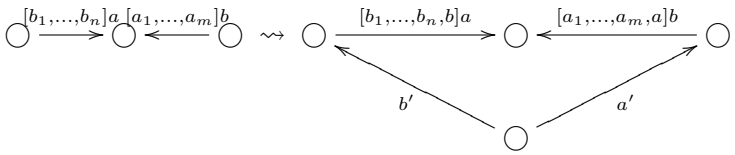



Figure 13: A step of DVR.

The directed virtual reduction of a directed virtual nets yields a directed virtual nets. DVR is a particular case of VR, so it preserves the execution of the net. Now all we need is a practical strategy to implement DVR. We show two of them.

## 4.3 Combustion

Let's call *full* directed virtual net a directed virtual net in which each edge is either *ghost*, i.e. has no counter-edge, or is weighted by a positive monomial (typically, a proof-net/a sharing graph is translated into a full directed virtual net). From now on the valence of a node is the number of non ghost edges exiting this node. Note that if a node has valence zero, then all his outgoing edges are ghost thus the node will never receive any residuals from them.

**Proposition 1.** *The* combustion strategy *chooses a node of valence 0 and performs all the possible compositions. If $R'$ is obtained from $R$ by the combustion strategy and if $R$ is full then so is $R'$.*

The idea is that once a node of the net receives all edges, we can perform all steps of DVR in one macro-step (in particular, there is no more need to keep filters).

## 4.4 Half-Combustion

We now define the Half Combustion Strategy (HC) that like combustion does not require to keep filters and, in addition, allows the composition to be performed even on nodes having valence greater than zero, thus allowing high degree of parallelism. HC relies on the following notion of *semifull* directed virtual net which is a generalization of the notion of full directed virtual net.

**Definition 20.** *Let us call semifull directed virtual net a directed virtual net in which each edge either is weighted by a positive monomial (i.e. its weight has no filter) or all its coincident counter-edges are weighted by a positive monomial (i.e. it can be composed exclusively with edges having a positive weight).*

**Definition 21.** *Given a composable edge $\alpha$ with positive weight in a semifull directed virtual net $R$, we have to consider two cases:*

[6] Cfr. [Pin01, p. 118]
[7] No *directed* cycle.

1. *if $\alpha$ has no non-positive coincident counter-edge and (at least) a positive one $\beta$, then the half combustion strategy (HC) performs the composition of $\beta$ with $\alpha$ and possibly with every non-positive edge composable with $\beta$;*

2. *if the set $\{\beta_1, \ldots, \beta_n\}$ of non-positive edges composable with $\alpha$ is non-empty then HC performs all the possible compositions of $\alpha$ with the $\beta_i$s.*

**Remark 1.** *HC can be performed discarding the filters. Let us explain how. A mark is associated to each edge,* incoming *or* combusted*: when an edge is created, it is marked as incoming; after it has been composed with every other coincident edge, it is marked as combusted. If two edges belong to the set of the combusted edges of a node, then they are never composable. Note that an edge may have positive weight and be combusted (if it is not composable with any coincident edge).*

# 5 Implementation on GPU

PELCR is a software for lambda calculus reductions composed of several modules. One of these, called `symbolic.c`, is responsible for the compositions of weights of $L^*$ and related tasks. We gave a preliminary implementation in order to execute PELCR using parallelism provided by GPUs, that we shortly report below.

## 5.1 GPGPU

General Purpose GPUs support general-purpose high-level languages, so the programmer does not have to be concerned with hardware details. This devices are suitable for vectorial task, i.e., all programs which benefits from large-scale data-parallelism can be efficiently executed on GPUs, as explained in [NVI09, Section 1.1]:

> [...]the GPU is specialized for compute-intensive, highly parallel computation — exactly what graphics rendering is about — and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations — the same program is executed on many data elements in parallel — with high arithmetic intensity — the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

### 5.1.1 CUDA

C for CUDA extends C by allowing the programmer to write C functions called *kernels* that can be executed $N$ times in parallel by $N$ CUDA threads. A kernel is defined by the keyword `__global__`, that denotes a function called by the host and executed on the device.

The threads that runs on the GPU are grouped in *blocks* that are themselves grouped in a *grid*. Blocks can be 1D, 2D or 3D, and they all have the same sizes, while the grid

can only be 1D or 2D. Threads within a block can be synchronized calling the function `__syncthreads()`, which acts as a barrier at which all threads must wait before any can proceed (global synchronization can be obtained using different kernels). Block and grid size must be specified at kernel launch. All threads in a block reside on the same processor and they all share a common memory called *shared memory* designed for fast read-and-write access. Besides shared memory, a thread has access to its own local memory, to *global memory* (which is shared by all threads, but not very fast) and a *constant memory* (a special read-only memory that is accessible to all threads and is faster than global memory). There is also a special *texture memory* for graphics-specific purposes

A CUDA GPU consists of an array of multi-threaded Streaming Multiprocessors (SMs). When a block terminates, new blocks are automatically loaded; the number of blocks can greatly exceed the number of blocks that can be executed concurrently by all processor; the number of threads in a single block can also be greater than the maximum number of concurrent threads a multiprocessor can handle. This allows the device to better schedule the various blocks and threads. A Multiprocessor contains eight Scalar Processor (SP) cores, two special functions units for trascendentals, a multi-threaded instruction unit, and an on-chip memory. The execution is scheduled in hardware with zero overhead and synchronizations is done with a single hardware instruction, so very fine-grained parallelism is possible.

# 6 Implementation

First we describe a new coding of the weights, then we give a high-level description of the kernel showing how DVR with HC can be implemented.

## 6.1 Weight representation

The idea is to use `unsigned int` to represent integers and symbols (`'!','w','p','q'`): integers will be represented in the obvious way (more or less), while symbols will be represented using a symbol code.

**Example 2.** *Let us consider* $X = w(1,2)!^2(qpp!(p))$*, how is it encoded? First we separate the symbolic and the numeric parts:*

$$\texttt{"w(1,2)!5qpp!1p"} \mapsto \texttt{("w!qpp!p", [(1,2)][5][1])} = (X_s, X_n)$$

*Then we encode the symbolic part with the following binary symbolic code:*

$$c_s: \begin{array}{|c|c|}\hline a_i & c_s(a_i) \\ \hline ! & 00 \\ w & 01 \\ p & 10 \\ q & 11 \\ \hline \end{array} \tag{4}$$

$$c_s^+(w!qpp!p) = [01001110100010000000000000000000]$$

*As for the numeric part, we use a special encoding to memorize the index and the lift of an exponential into the same* `unsigned`*, while the level of a bang is encoded normally:*

$$c_n : \begin{array}{|c|c|}\hline a_i & c_w(a_i) \\ \hline (i,l) & [i|l] \\ \hline a_i & c_!(a_i) \\ \hline k & k \\ \hline \end{array} \tag{5}$$

$$\begin{aligned} c_n^+([(1,2)][5][1]) = [1|2][5][1] = \\ [00000000000000010000000000000010] \\ [00000000000000000000000000000101] \\ [00000000000000000000000000000001]. \end{aligned}$$

*$X$ can be finally encoded as $\{c_s^+(X_s), c_n^+(X_n)\}$.*

## 6.2 Kernel

To implement DVR with half-combustion, we consider the strategy we describe below. First, let us make a few considerations:

1. since the kernel grid is composed of blocks of the same size, there should be some aggregation mechanism analogous to the one already used for the application messages in PELCR (to avoid wasting resources on the GPU);
2. the sizes of different (semi)nodes might differ a lot, so there should be several kernel invocation (each call, perhaps on different streams, should process the nodes which have a number of combusted edges inferior to a certain threshold);
3. since only one CPU process can execute a kernel on the GPU at one time, we should determine an appropriate policy (the starting process controls the GPU throughout the execution; or when the kernel terminates, the control of the GPU passes to the process with the maximum number of resident nodes; or the GPU is used periodically by all process; ...);
4. the execution should be organized in such a way the various weights stay on the GPU for as long as possible (until the read-back needs to be performed, if possible);
5. the (format of the) output of a kernel call should be as close as possible to the (format of the) input of a future invocation, to minimize additional work (the aggregation strategy needs to accumulate the weights of incoming edges as future input).

**Execution strategy** Each threads block corresponds to a node: each thread in a block calculates the product of a left incoming edge and a right combusted edge, then adds the left incoming edge to the set of left combusted edges and synchronizes with the threads in the block. Then that thread in a block calculates the product of a right incoming edge and a left combusted edge (this includes edges just created), then adds the right incoming edge to the set of right combusted edges.

# References


[AG98] 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.

[ALDR94] Andrea Asperti, Cosimo Laneve, Vincent Danos, and Laurent Regnier. Paths in the lambda-calculus — three years of communications without understanding, 1994.

[DR93] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of girard's execution formula). In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science (LICS 1993)*, pages 296–306. IEEE Computer Society Press, June 1993.

[DR95] Vincent Danos and Laurent Regnier. Proof-nets and Hilbert space. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 307–328. Cambridge University Press, 1995.

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[Gir89a] Jean-Yves Girard. Geometry of interaction. I. Interpretation of system **F**. In *Logic Colloquium '88 (Padova, 1988)*, volume 127 of *Stud. Logic Found. Math.*, pages 221–260. North-Holland, Amsterdam, 1989.

[Gir89b] Jean-Yves Girard. Towards a geometry of interaction. In *Categories in computer science and logic (Boulder, CO, 1987)*, volume 92 of *Contemp. Math.*, pages 69–108. Amer. Math. Soc., Providence, RI, 1989.

[Lév78] Jean-Jacques Lévy. Réductions correctes et optimales dans le lambda calcul, 1978.

[NVI09] NVIDIA. CUDA programming guide 2.3, 2009.

[Pin01] Jorge Sousa Pinto. *Implantation Parallèle avec la Logique Linéaire*. PhD thesis, École Polytechnique, 2001.

[Wad71] P. Christopher Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, Oxford, 1971.