LIGHT LINEAR LOGIC

Jean-Yves Girard
Institut de Mathématiques de Luminy, UPR 9016 – **CNRS**
163, Avenue de Luminy, Case 930, F-13288 Marseille Cedex 09

girard@iml.univ-mrs.fr

Abstract

The abuse of structural rules may have damaging complexity effects.

1 INTRODUCTION : A LOGIC OF POLYTIME ?

We are seeking a « logic of polytime ». Not yet one more axiomatization, but an intrinsically polytime system. Our methodological bias will be to consider that the expressive power of a system is the complexity of its cut-elimination procedure, and we therefore seek a system with a polytime complexity for cut-elimination (to be precise : besides the *size* of the proof, there will be an auxiliary parameter, the *depth*, controlling the degree of the polynomial). This cannot be achieved within classical or intuitionistic logics, because of structural rules, especially contraction : this is why the complexity of cut-elimination in all extant logical systems (including the standard version of linear logic which controls structural rules without forbidding them) is catastrophic, elementary (towers of exponentials) or worse. Light Linear Logic is a purely logical system with a more careful handling of structural rules : this system is strong enough to represent all polytime functions, but cut-elimination is (locally) polytime. With **LLL** our control over the complexity of cut-elimination improves a lot.
But this is not the only potentiality of **LLL** : why not transforming it into a system of mathematics, and try to formalize « polytime mathematics » in the same way as Heyting arithmetic formalizes constructive mathematics ? The possibility is clearly open, since **LLL** admits extensions into a naive set-theory, with full comprehension, still with a polytime cut-elimination. This system admits full induction on data types, which shows that, within **LLL**, induction is compatible with low complexity...

1.1 Background

1.1.1 Complexity of normalization

Our goal is to find a logical system in which the I/O dependencies are given by polytime functions. We shall try a proof-theoretic approach, namely to make sure that cut-elimination is polytime. In fact we shall concentrate on the following question : which *logical* system(s) induce normalizations of a given complexity (polytime or not) ?

There is an answer, namely **MALLq** (multiplicative-additive-quantifiers (of any order) linear logic) : the small normalization theorem of [3] assigns a size bound to proofs ; this size shrinks during *lazy* cut-elimination, hence induces linear time functions. A crucial technological point is that the notion of cut-degree disappears, i.e. the procedure is not dependent on the fact that cuts are replaced with simpler ones. The operation has perfectly succeeded, but the patient died (of starvation) : this system is desperately inexpressive.

Linear logic wisely has another stock of connectives, namely *exponentials* which should compensate for this limitation, by restoring the necessary amount of structural manipulations (mainly the contraction rule). Now the patient is still dying, but of overfeeding : the complexity is no longer bounded by any reasonable measure, since usual logic (classical or intuitionistic) imbeds. The question is therefore to find more reasonable connectives, sitting in between **MALLq** and **LL**. These connectives are the *light* exponentials.

The first attempt dates back from 1987 (joint work with A. Scedrov & P. Scott, [2]) and is based on the idea of replacing $!A$ (which usually means A *ad libitum*) by $(1 \,\&\, A) \otimes \ldots \otimes (1 \,\&\, A)$, i.e. essentially by a finite tensor power of A, $!_nA$. It is also immediate to see that the rules of weakening, dereliction, contraction and promotion are still valid w.r.t. bounded exponentials : the bounds are respectively given by $0, 1, +, .$, i.e. the maintenance is polynomial. This very good starting point leads to **BLL** (Bounded Linear Logic) ; **BLL** has a lot of qualities (it exactly corresponds to polytime etc.), but it has a major drawback : it mentions the polynomial bounds which should remain hidden [1]. By the way observe that **BLL** is far from giving good bounds : the main property of exponentials is the isomorphism between $!A \otimes !B$ and $!(A \,\&\, B)$, but **BLL** yields the bounds $!_{n+m}(A \,\&\, B) \multimap !_nA \otimes !_mB$ and $!_nA \otimes !_nB \multimap !_n(A \,\&\, B)$, which induce by composition $!_{2n}(A \,\&\, B) \multimap !_n(A \,\&\, B)$, not quite an isomorphism.

Since this first attempt, many other restrictions have been tried by Danos, Joinet, Lafont, Schellinx and myself, without obtaining truly convincing results [2].

Other connections between polytime complexity and normalization have been made in recent years, typically works of Leivant, Leivant-Marion [10, 11], and Hillebrand-Kanellakis-Mairson, [6]. These approaches stay inside typed λ-calculi, i.e. systems which are by no standards polytime (the complexity is at least elementary), but they individualize certain interesting situations where the complexity is exactly polytime (this is based on the fact that in traditional situations,

---

[1] These bounds do not refer to time, but to size.

[2] *A posteriori* it is clear that all these attempts failed because they included the principle [V], see 1.1.3.

the complexity is determined by the cut-formulas : the basic idea is to restrict one to cuts of a certain form to achieve complexity effects). The obvious advantage of these approaches lies is the use of traditional systems (or at least systems not too far from that). But these systems can hardly claim to bring some insight as to the logical nature of polytime, since as soon as we iterate their logical primitives, the complexity explodes, in other terms the logical primitives (basically intuitionistic implication) make mistakes w.r.t. complexity. To take an analogy : classical (Peano) arithmetic is indeed constructive for Π^0_2 sentences, where it coincides with Heyting arithmetic ; for more complex formulas, it is constructively wrong. There is therefore still a want for a system which is intrinsically polytime, in the way that Heyting arithmetic is intrinsically constructive. An answer is **LLL**.

1.1.2 LLL and naive set-theory

We are seeking a logical system with a light complexity. This basically means that the cut-elimination bounds will not depend on the cut-formulas, i.e. will not rely on the replacement of a cut by a simpler one. Then such a system will accommodate naive set-theory. This is simply because naive set-theory has a normalization procedure (the one described by Prawitz in the 60's) which will terminate in such a framework. Typically this works for **MALLq** (this is indeed an old remark of Grishin, [5]), since the naive comprehension scheme does not prevent normalization from shrinking !

Our crucial test for selecting the right rules, will be to check whether or not naive set-theory becomes inconsistent with the proposed set of rules for exponentials. Typically, naive set-theory enables us to get fixpoints of any logical operation (like naive function theory, i.e. λ-calculus), and it suffices to check the impossibility of getting a contradiction from fixpoints. The best candidate is the one arising from Russell's paradox, i.e. $A \simeq !A^\perp$. For those who find this methodology surprising, we can phrase it differently : inconsistency provides a non-terminating cut-elimination, and non-termination can be seen as the worse possible complexity.

We shall therefore tailor our light exponentials w.r.t. naive set-theory, but keep only the second order propositional logic arising from this study. It would be possible to do much more : one can add the logical rules of naive set-theory to **LLL**, and this provides a very powerful system. In this system extensionality fails (as already observed by Grishin), but Leibniz equality can do wonders. Integers in unary (or binary) representation can be defined, and full induction therefore works. In other terms, one can get a pure logical system without any proper axiom, which contains both a light set-theory and a light arithmetic.

1.1.3 Dissection of exponentials

Exponentials are used to “classicize” **LL**. This involves a lot of micro properties, that we can individualize below :

- [I] : $!(A \,\&\, B) \multimap !A \otimes !B$ (and $!\top \multimap 1$)
- [II] : $!A \otimes !B \multimap !(A \,\&\, B)$ (and $1 \multimap !\top$)

- [III] : from $A \multimap B$ derive $!A \multimap !B$
- [IV] : $!A \multimap ?A$
- [V] : $!A \otimes !B \multimap !(A \otimes B)$ (and $!1$)
- [VI] : $!A \multimap A$
- [VII] : $!A \multimap !!A$

The first two principles express the usual isomorphism which is responsible for the name "exponential" : principle [I] expresses contraction and principle [II] expresses weakening.
Principle [III] expresses functoriality of the exponentials and is absolutely basic.
Principle [IV] is a weak form of dereliction (i.e. principle [VI]).
These four principles will constitute the basis of **LLL**.
Principle [V] enables one to give a multilinear version of functoriality (from $\Gamma \vdash B$ derive $!\Gamma \vdash !B$), and will not be accepted in **LLL**, although it is also compatible with naive set-theory [3].
In the presence of the fixpoint $A \simeq !A^{\perp}$, it is possible to derive the sequent $\vdash$ (so no cut-elimination !), in two ways

- from [I] + [III] + [VI]
- from [I] + [III] + [IV] + [VII]

In both cases one first proves $\vdash ?A, ?A$ from the fixpoint principle $\vdash A, ?A$:

- dereliction [VI] yields $\vdash ?A, ?A$
- [III] yields $\vdash !A, ??A$, then [IV] yields $\vdash ?A, ??A$ and [VII] removes the extra "?"

From $\vdash ?A, ?A$ contraction [I] yields $\vdash ?A$, and by fixpoint one gets $\vdash A^{\perp}$, which by promotion yields in turn $\vdash !A^{\perp}$. We end with a cut between $\vdash ?A$ and $\vdash !A^{\perp}$. Therefore principles [VI] and [VII] are definitely excluded.
The failure of dereliction is the reason for the introduction of the weaker principle [IV]. Unfortunately it turns out that this principle is too weak in terms of expressive power, and this is the reason why an additional modality is introduced.

1.1.4 The three modalities

In **LLL** there are indeed three modalities $!, \S, ?$. $\S$ (*neutral*) is a new intermediate modality. $\S$ is self-dual, i.e. $(\S A)^{\perp}$ is $\S A^{\perp}$, and its intuitive meaning is the (common) unary case of ! and ?. The principles of **LLL** are :

- [I], [II], [III] (written in terms of $!, ?$)
- [VIII] : from $A \multimap B$ derive $\S A \multimap \S B$
- [IX] : $!A \multimap \S A$

---

[3] It yields another logic with an elementary complexity for cut-elimination, **ELL**.

- [X] : $\S A \otimes \S B \multimap \S(A \otimes B)$ (and $\S 1$)

[VIII] is just usual functoriality, and [X] enables one to get a n-ary version ; [IX] is a compensation for the want of dereliction...observe that it implies by duality $\S A \vdash ?A$ and is therefore an improved version of [IV].
These principles can be organized along a sequent calculus which enjoys a cut-elimination with polynomial bounds, as we shall see below.

1.2 Expressive power of LLL

LLL can be seen as a system of set-theory (or arithmetic). It can also be seen as a system of typed λ-calculus. We have to explain how to encode data and polytime algorithms.

1.2.1 Integers

Remember that complexity depends on the representation of data : typically integers can be given in tally representation or in binary representation, with an exponential reduction of their size. This is why we introduce two types, **int** and **bint** for integers.
Tally integers can be given the type $\mathbf{int} = \forall X.!(X \multimap X) \multimap \S(X \multimap X)$, where X is a second order variable. The traditional type $\forall X.!(X \multimap X) \multimap (X \multimap X)$ cannot be used for want of dereliction, and the immediate substitute for it, $\forall X.!(X \multimap X) \multimap !(X \multimap X)$ cannot be used either, since the principle $!(A \multimap A); !(A \multimap A) \vdash !(A \multimap A)$ is not part of **LLL**. Observe that *addition* can be given the type $\mathbf{int}; \mathbf{int} \vdash \mathbf{int}$ and *multiplication* can be given the type $\mathbf{int}; !\mathbf{int} \vdash \S\mathbf{int}$. In fact any polynomial P in n variables can be given [4] a type $\mathbf{int} \otimes \ldots \otimes \mathbf{int} \multimap \S^k\mathbf{int}$, where k is an integer depending on the degree of P. Typically, x^2 can be given the type $\mathbf{int} \multimap \S\S\mathbf{int}$.
Binary integers (lists of 0 and 1) can be given the type $\mathbf{bint} = \forall X.!(X \multimap X) \otimes !(X \multimap X) \multimap \S(X \multimap X)$. There is a canonical map which consists in replacing a binary list with a unary one, i.e. $\sharp(x)$ is the length of x in tally representation. The type of this map is $\mathbf{bint} \multimap \mathbf{int}$.

1.2.2 Turing machines

Let us fix the alphabet and the set of states of our Turing machines. In order to represent our machines, all we now have to do is to find a type **Tur** (see 2.5.4), in such a way that configurations (tape + state) of such a machine are exactly the objects of type **Tur**. **Tur** must also be such that the instructions of a Turing machine induce objects of type **Tur** $\multimap$ **Tur**. Several possibilities are at hand, but the simplest is for sure to use the fixpoint facility coming from the naive comprehension axiom. (The fixpoint of the operator $\Phi[p]$ is obviously $t \in t$, with $t = \{x \mid \Phi[x \in x]\}$).

1.2.3 Polytime functions

Let us now take a polytime program from binary integers to binary integers, with runtime P. We can consider the function of type $\mathbf{bint} \multimap \S\mathbf{Tur}$ which yields the

---

[4] Up to minor details

input configuration of the machine, as well as the function of type $\mathbf{bint} \multimap (\S)^k\mathbf{int}$ which yields the number of steps ; if our program is represented by φ of type $\mathbf{Tur} \multimap \mathbf{Tur}$, then we eventually get an object of type $\mathbf{bint} \multimap (\S)^{k+2}\mathbf{Tur}$ which yields, in function of the binary input, the output tape.
This representation has no pretension to elegance : its only virtue is to show the expressive power of **LLL**. Since **LLL** is a real logical system in which it is impossible to be worse than polytime, smarter representations must be found.

1.3 Cut-elimination

The sequent calculus naturally associated with **LLL** is a double-layer version, i.e. with additive and multiplicative disjunctions. This is not very friendly [5], but after all sequent calculus is not the only proof-theoretic syntax, and one can use more sophisticated technologies, typically proof-nets. The proof-net technology has made essential progress in the recent years, and it is now possible to represent the full sequent calculus, in terms of proof-nets with boxes. Boxes are only needed for exponentials : the promotion rule [III] induces a box as well as the dereliction (or rather its weaker version) (combination of [VIII],[IX],[X]). The main parameters of a proof are

- its *depth*, which is the nesting number of the exponential boxes ;
- its *size* which counts the number of links ;
- its partial sizes, i.e. the size of the part of the net which is at a certain depth.

Cut-elimination works as follows : it is a lazy one, i.e. no cut is eliminated « inside a &-box » [6], which is performed layer after layer, first starting with depth 0. After eliminating the cuts of depth 0, the sizes (which were $s_0, s_1, s_2, \ldots$) become at most $s_0, s_0s_1, s_0s_2, \ldots$ From this it is immediate that after eliminating all cuts, the final size is roughly s^{2^d}, where s, d are the original size and depth. What makes the argument work is that the light rules are of constant depth, i.e. that no change (increase or decrease) may happen during cut-elimination. By the way, these bounds are the simplest refutation for additional principles such as $[XI] : \S(A \oplus B) \vdash \S A \oplus \S B$: a polytime boolean function can be given a type $\mathbf{bint} \vdash \S^k(1 \oplus 1)$, and by $[XI]$ the type $\mathbf{bint} \vdash \S^k 1 \oplus \S^k 1$; but in that case the output is given by the $\oplus$-rule used (left or right), and since $\oplus$ is not nested, we get this rule after a purely external normalization whose runtime is linear.
With proof-nets, it is easy to see that the bound immediately yields a $s^{2^{d+2}}$ time bound for usual I/O, like binary strings. Moreover a binary string is represented by a proof-net of depth 1, hence the application $f(s)$ of a given function f of type $\mathbf{bint} \multimap (\S)^k\mathbf{Tur}$ to a binary string s will have the same depth as f, i.e. the computation will run in a time which is polynomial in the size of s.

---

[5] The proof of cut-elimination with the polynomial bounds is not manageable with sequent calculus.

[6] Technically this is the notion of ready cut coming from [4]... strictly speaking there are no longer additive boxes.

1.4 Added in print

Three years later (October 1997), the paper seems to be essentially the individuation of two weird proof-theoretical worlds, corresponding to **LLL** and the sketched **ELL** : we now know how to logically control complexity of cut-elimination. On the other hand neither exist as logical worlds, for want of definite interpretations ; in particular, certain principles which are proof-theoretically admissible (i.e. which are compatible with our complexity-theoretic constraints) can be accepted or refused depending on personal taste : typically § will have a tendency to lose its self-duality, promotion with empty context can be added to **LLL** with immediate simplifications... one must also mention Asperti's [1] variant of intuitionistic affine **LLL**, with immediate simplifications too. In other terms **LLL** is a theme with possible variations, and not a well-defined logical system ; this unpleasant situation will only be fixed when a convincing semantics will be produced ; in this respect, the recent work of Kanovich, Okada and Scedrov [8] is a partial fulfilment of this goal (by the way, it concludes to a non self-dual §).

2 THE SYNTAX OF LLL

Constructive logic is basically propositional ; this is why we focus on (second-order) propositional **LLL**. However the system is quite flexible and accepts quantifiers of any order, including set-quantifiers.

2.1 The formulas of LLL

LLL has the same connectives as usual linear logic but for the exponentials : there is an extra (self-dual) modality, § (*neutral*) is added [7].

Definition 1

Literals (T) *and formulas* (F) *are defined as follows*

$T = \alpha, \beta, \gamma \ldots, \alpha^\perp, \beta^\perp, \gamma^\perp \ldots$

$F = T, 1, \perp, 0, \top, !F, \S F, ?F, F \otimes F, F \parr F, F \,\&\, F, F \oplus F, \forall \alpha F, \exists \alpha F$

Definition 2

(Linear) negation is a defined connective :

- $(\alpha)^\perp = \alpha^\perp, (\alpha^\perp)^\perp = \alpha$
- $1^\perp = \perp, \perp^\perp = 1$
- $0^\perp = \top, \top^\perp = 0$
- $(!A)^\perp = ?A^\perp, (\S A)^\perp = \S A^\perp, (?A)^\perp = !A^\perp$
- $(A \otimes B)^\perp = A^\perp \parr B^\perp, (A \parr B)^\perp = A^\perp \otimes B^\perp$
- $(A \,\&\, B)^\perp = A^\perp \oplus B^\perp, (A \oplus B)^\perp = A^\perp \,\&\, B^\perp$

---

[7] We have been tempted to replace $!, \S, ?$ by the musical symbols $\sharp, \natural, \flat$

- $(\forall \alpha A)^\perp = \exists \alpha A^\perp, (\exists \alpha A)^\perp = \forall \alpha A^\perp$

Linear implication is a defined connective :
$A \multimap B = A^\perp \parr B$

2.2 The sequents of LLL

Definition 3

- *A* discharged *formula is an expression* $[A]$, *where* A *is a formula ;*
- *A* block $\mathbf{A}$ *is a sequence* $A_1, \ldots, A_n$ *of formulas, or a single discharged formula* $[A]$ *; the standard case is that of a block of length* 1, *for which we use the notation* A *or* $[A]$ *;*
- *A* sequent *is an expression* $\vdash \mathbf{A_1}; \ldots; \mathbf{A_n}$, *where* $\mathbf{A_1}, \ldots, \mathbf{A_n}$ *are blocks. The standard case is that of a sequence of (undischarged) formulas ; even more standard is the case when the sequence consists of exactly one formula.*

Remark. —

- A *block* $A_1, \ldots, A_n$ is hypocrisy for the formula $A_1 \oplus \ldots \oplus A_n$;
- A discharged formula $[A]$ is hypocrisy for $?A$;
- If $\mathbf{A_1}, \ldots, \mathbf{A_n}$ are hypocrisy for formulas $A_1, \ldots, A_n$, then the sequent $\vdash \mathbf{A_1}; \ldots; \mathbf{A_n}$ is hypocrisy for the formula $A_1 \parr \ldots \parr A_n$.

2.3 The sequent calculus of LLL

Identity / Negation

$$\frac{}{\vdash A; A^\perp}\ (\mathit{identity}) \qquad\qquad \frac{\vdash \Gamma; A \quad \vdash A^\perp; \Delta}{\vdash \Gamma; \Delta}\ (\mathit{cut})$$

Structure

$$\frac{\vdash \Gamma; \mathbf{A}; \mathbf{B}; \Delta}{\vdash \Gamma; \mathbf{B}; \mathbf{A}; \Delta}\ (\textit{M-exchange}) \qquad\qquad \frac{\vdash \Gamma; \mathbf{A}, C, D, \mathbf{B}}{\vdash \Gamma; \mathbf{A}, D, C, \mathbf{B}}\ (\textit{A-exchange})$$

$$\frac{\vdash \Gamma}{\vdash \Gamma; [A]}\ (\textit{M-weakening}) \qquad\qquad \frac{\vdash \Gamma; \mathbf{A}}{\vdash \Gamma; \mathbf{A}, B}\ (\textit{A-weakening})$$

$$\frac{\vdash \Gamma; [A]; [A]}{\vdash \Gamma; [A]}\ (\textit{M-contraction}) \qquad\qquad \frac{\vdash \Gamma; \mathbf{A}, B, B}{\vdash \Gamma; \mathbf{A}, B}\ (\textit{A-contraction})$$

Logic

$$\frac{}{\vdash 1}\ (\mathit{one}) \qquad\qquad \frac{\vdash \Gamma}{\vdash \Gamma; \bot}\ (\mathit{false})$$

$$\frac{\vdash \Gamma; A \quad \vdash B; \Delta}{\vdash \Gamma; A \otimes B; \Delta}\ (\textit{times}) \qquad \frac{\vdash \Gamma; A; B}{\vdash \Gamma; A \parr B}\ (\textit{par})$$

$$\frac{}{\vdash \Gamma; \top}\ (\textit{true}) \qquad (\textit{no rule for zero})$$

$$\frac{\vdash \Gamma; A \quad \vdash \Gamma; B}{\vdash \Gamma; A \,\&\, B}\ (\textit{with}) \qquad \frac{\vdash \Gamma; A}{\vdash \Gamma; A \oplus B}\ (\textit{left plus}) \qquad \frac{\vdash \Gamma; B}{\vdash \Gamma; A \oplus B}\ (\textit{right plus})$$

$$\frac{\vdash B_1, \ldots, B_n; A}{\vdash [B_1]; \ldots; [B_n]; !A}\ (\textit{of course}) \qquad \frac{\vdash \Gamma; [A]}{\vdash \Gamma; ?A}\ (\textit{why not})$$

$$\frac{\vdash B_1 \mid \ldots \mid B_n; A_1; \ldots; A_m}{\vdash [B_1]; \ldots; [B_n]; \S A_1; \ldots; \S A_m}\ (\textit{neutral : } B_1, \ldots, B_n \textit{ are formulas separated by commas or semicolons})$$

$$\frac{\vdash \Gamma; A}{\vdash \Gamma; \forall\alpha\, A}\ (\textit{for all : } \alpha \textit{ is not free in } \Gamma) \qquad \frac{\vdash \Gamma; A[B/\alpha]}{\vdash \Gamma; \exists\alpha\, A}\ (\textit{there is})$$

2.4 Comments on the system

2.4.1 Discharging

Little attention should paid to discharged formulas ; $[A]$ may be replaced with $?A$ with practically no difference. The interest of this pedantism appears in the proof of cut-elimination through the translation into proof-nets : there the distinction between discharge and ? actually matters.

2.4.2 Additive blocks

The main idea behind **LLL** is to restrict promotion to a unary context, i.e. to $[III]$: from $\vdash A; B$ derive $\vdash ?A; !B$. But this is not enough to get $[II]$, e.g. $\vdash ?A^\perp; ?B^\perp; !(A \,\&\, B)$, and this is why additive blocks are introduced, with permission to treat them as a unique formula : $\vdash ?A^\perp; ?B^\perp; !(A \,\&\, B)$ can be obtained from $\vdash A^\perp, B^\perp; A \,\&\, B$. In other terms, the **LLL**-promotion rule is exactly $[II] + [III]$. A particular case is promotion with one empty block (a block may be empty), i.e. from $\vdash ; A$ derive $\vdash !A$, a rule difficult to apply, since $\vdash ; A$ has the same meaning as $\vdash 0; A$, i.e. $\top \multimap A$. This should not be confused with the (illegal) promotion without context « from $\vdash A$ derive $\vdash !A$ »[8].

---

[8] By the way the adoption of this rule (i.e. promotion with at most one block in the context) would simplify many minor details in encoding polytime, and moreover this rule would not destroy the polytime complexity of cut-elimination : one more possible « fine tuning » of rules.

We eventually stumble on this double-layer sequents, with two series of structural rules, M-rules and A-rules (M for multiplicative, A for additive). Promotion replaces commas with semi-colons, with is the intended meaning of exponentiation, i.e. the replacement of additives (",") with multiplicatives (";"). By the way, some people would prefer to keep "," for the multiplicative comma... the problem being that ";" looks "stronger" than ","... I don't know.

2.4.3 Neutral

The rule contains « neo-dereliction », i.e. the principle $!A \multimap \S A$, as well as the principle $\S A \otimes \S B \multimap \S(A \otimes B)$. Observe that the punctuation between the A_i in our rule is a semi-colon, and it cannot be replaced with a comma without damaging the full architecture. The rule also expresses the fact that $!A \multimap ?A$. But what is problematic is the self-duality of §, which is compatible with our results, but seems to be logically wrong ; a non self-dual § would have a rule like

$$\frac{\vdash B_1 \mid \ldots \mid B_n ; A_1 ; \ldots ; A_m ; A}{\vdash [B_1] ; \ldots ; [B_n] ; \S^{\perp} A_1 ; \ldots ; \S^{\perp} A_m ; \S A}$$

2.5 The expressive power of LLL

Our goal here is to prove that polytime functions can be represented in **LLL**. This can be established by various means. We adopt the simplest (maybe not the most elegant) solution, namely to encode polytime Turing machines in an intuitionistic version of **LLL**, **ILLL**. There will be a forgetful function of **ILLL** into system $\mathbb{F}$ (with conjunction), hence the ultimate interpretation will be in system $\mathbb{F}$, in which the representation of data, algorithms, is quite familiar...

2.5.1 The system ILLL

The language of **ILLL** is based on the connectives $\otimes, \&, \multimap, !, \S$, and second-order quantification. The sequents of **ILLL** are of the form $\mathbf{A_1} ; \ldots ; \mathbf{A_n} \vdash B$, where $\mathbf{A_1}, \ldots ; \mathbf{A_n}$ are blocks, and B is a formula. The formulas $1^k = !^k 1$ are allowed in the blocks, although they are not part of the language of **ILLL**[9]. The rules of **ILLL** are those that remain correct when we translate $\mathbf{A_1} ; \ldots ; \mathbf{A_n} \vdash B$ as $\vdash \mathbf{A_1}^{\perp} ; \ldots ; \mathbf{A_n}^{\perp} ; B$.
The forgetful functor (*erasure*) from **ILLL** into **LL** (second-order propositional intuitionistic linear logic, based on implication, conjunction and universal quantification) is defined as follows :

- To a formula A of **ILLL** we associate A^-, a formula of **LL**, as follows : ! and § are erased, $\otimes$ and & are replaced with $\wedge$, $\multimap$ is replaced with $\Rightarrow$, variables and quantifiers are unchanged ;
- To a sequent S of **ILLL** we associate a sequent S^- of **LL** :
  - In S remove all discharges $[\cdot]$;

---

[9] This is due to the technical restrictions on the !-rule ; should **ILLL** be developed for its own sake, these special formulas should be replaced with markers...

- In S replace all semicolons by commas ;
- In S replace all formulas of **LLL** by their erasure ;
- In S remove all formulas 1^k.

Typically the erasure of $A, B, !!1; [1]; [C]; D, E \vdash F$ will be interpreted as $A^-, B^-, C^-, D^-, E^- \vdash F^-$.

▶ To a proof $\mathbf{\Pi}$ of S we associate a proof $\mathbf{\Pi}^-$ of S^-; since **LL**-sequent-calculus can be interpreted in natural deduction, i.e. system $\mathbb{F}$v, we eventually get a term of system $\mathbb{F}$. Typically a proof of $A, B, !!1; [1]; [C]; D, E \vdash F$ will induce a term t, depending on variables v, w, x, y, z :

$$v : A^-, w : B^-, x : C^-, y : D^-, z : E^- \vdash t : F^-$$

The interpretation is straightforward ; observe that the erasure of 1^k is unproblematic, since 1^k indeed deals with weakening, i.e. dummy variables.

The basic idea behind the erasure is that (intuitionistic) linear logic (light or not) can be viewed as a more refined way to speak of implication, conjunction, erasing reuse. These refinements are not taken into account in intuitionistic logic, and the forgetful functor collapses the two conjunctions, ignores exponentials (and therefore destroys 1^k, a very subtle handling of weakening). This is reflected in the translation of the formulas and also of the sequents, where the additive, multiplicative and exponential layers (represented by “, ”, “; ”, $[\cdots]$ are collapsed into a comma. When we shall represent data and algorithms in **ILLL**, we shall implicitly refer to their forgetful image in $\mathbb{F}$. It goes without saying that the notion of reduction to be defined in **LLL** is compatible with the notion of reduction in second-order intuitionistic sequent calculus, and therefore (through the translation from sequent calculus to natural deduction) in system $\mathbb{F}$, so that the only consideration of the forgetful images matters. In what follows the most important functions are represented in details ; we assume that the reader is most familiar with system $\mathbb{F}$, the Curry-Howard isomorphism which identifies natural deduction with typed λ-terms, and therefore that he has no problem to synthesize the λ-term associated with a proof in second-order intuitionistic sequent calculus.

2.5.2 Representation of tally integers

Integers We define the type **int** of tally integers by

$$\mathbf{int} = \forall\alpha.\ !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$$

The tally integer $\overline{n}$ is obtained as follows :

$$\begin{array}{c}
\alpha \vdash \alpha \quad \alpha \vdash \alpha \\ \hline
\alpha \multimap \alpha ; \alpha \vdash \alpha \quad \alpha \vdash \alpha \\ \hline
\alpha \multimap \alpha ; \alpha \multimap \alpha ; \alpha \vdash \alpha \\
\vdots \\
\alpha \multimap \alpha ; \ldots ; \alpha \multimap \alpha ; \alpha \vdash \alpha \\ \hline
\alpha \multimap \alpha ; \ldots ; \alpha \multimap \alpha \vdash \alpha \multimap \alpha \\ \hline
[\alpha \multimap \alpha] ; \ldots ; [\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha) \\ \hline\hline
[\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha) \\ \hline
!(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha) \\ \hline
\vdash !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha) \\ \hline
\vdash \forall\alpha .\ !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)
\end{array}$$

It is immediate that $\mathbf{int}^- = \forall\alpha.\ (\alpha \Rightarrow \alpha) \Rightarrow (\alpha \Rightarrow \alpha)$, and that $(\overline{n})^- = \Lambda\alpha.\lambda x^{\alpha\Rightarrow\alpha}.\lambda y^{\alpha}.x(x \ldots (x(y)) \ldots)$.

Addition
Addition is the proof $+$ of $\mathbf{int};\mathbf{int} \vdash \mathbf{int}$ obtained as follows :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha \vdash \alpha \quad \alpha \vdash \alpha}{\alpha \multimap \alpha ; \alpha \vdash \alpha} \quad \alpha \vdash \alpha}{\alpha \multimap \alpha ; \alpha \multimap \alpha ; \alpha \vdash \alpha}}{\alpha \multimap \alpha ; \alpha \multimap \alpha \vdash \alpha \multimap \alpha}}{\S(\alpha \multimap \alpha) ; \S(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)} \quad \dfrac{\alpha \multimap \alpha \vdash \alpha \multimap \alpha}{[\alpha \multimap \alpha] \vdash !(\alpha \multimap \alpha)}}{\mathbf{int}_\alpha ; \S(\alpha \multimap \alpha) ; [\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha)} \quad \dfrac{\alpha \multimap \alpha \vdash \alpha \multimap \alpha}{[\alpha \multimap \alpha] \vdash !(\alpha \multimap \alpha)}}{\mathbf{int}_\alpha ; \mathbf{int}_\alpha ; [\alpha \multimap \alpha] ; [\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha)}}{\mathbf{int}_\alpha ; \mathbf{int}_\alpha ; [\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha)}}{\mathbf{int}_\alpha ; \mathbf{int}_\alpha \vdash \mathbf{int}_\alpha}}{\mathbf{int} ; \mathbf{int}_\alpha \vdash \mathbf{int}_\alpha}}{\mathbf{int} ; \mathbf{int} \vdash \mathbf{int}_\alpha}}{\mathbf{int} ; \mathbf{int} \vdash \mathbf{int}}$$

with $\mathbf{int}_\alpha = !(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha)$. It is immediate that the erasure of $+$ is the usual representation of addition in $\mathbb{F}$.

Multiplication
Multiplication is a proof $\times$ of $!\mathbf{int};\mathbf{int} \vdash \S\mathbf{int}$:

$$\dfrac{\dfrac{\dfrac{\mathbf{int}\vdash\mathbf{int} \qquad \begin{matrix}\vdots\,\overline{0}\\ \vdash\mathbf{int}\end{matrix}}{\mathbf{int}\multimap\mathbf{int}\vdash\mathbf{int}}}{\S(\mathbf{int}\multimap\mathbf{int})\vdash\S\mathbf{int}} \qquad \dfrac{\dfrac{\begin{matrix}\vdots\,+\\ \mathbf{int};\mathbf{int}\vdash\mathbf{int}\end{matrix}}{\mathbf{int}\vdash\mathbf{int}\multimap\mathbf{int}}}{[\mathbf{int}]\vdash!(\mathbf{int}\multimap\mathbf{int})}}{\dfrac{[\mathbf{int}];\mathbf{int}_{\mathbf{int}}\vdash\S\mathbf{int}}{\dfrac{[\mathbf{int}];\mathbf{int}\vdash\S\mathbf{int}}{!\mathbf{int};\mathbf{int}\vdash\S\mathbf{int}}}}$$

It is immediate that the erasure of $\times$ is the usual representation of multiplication in $\mathbb{F}$.

Iteration
The principle of *iteration* is derivable : if Γ is a block and not discharged, then from a proof of $\Gamma \vdash A \multimap A$ and a proof of $\Delta \vdash \S A$, one can derive $[\Gamma];\Delta;\mathbf{int} \vdash \S A$:

$$\dfrac{\dfrac{\Gamma\vdash A\multimap A}{[\Gamma]\vdash!(A\multimap A)} \qquad \dfrac{\Delta\vdash\S A \qquad \dfrac{\dfrac{\dfrac{}{A\vdash A} \quad \dfrac{}{A\vdash A}}{A;A\multimap A\vdash A}}{\S A;\S(A\multimap A)\vdash\S A}}{\Delta;\S(A\multimap A)\vdash\S A}}{\dfrac{[\Gamma];\Delta;\mathbf{int}_A\vdash\S A}{[\Gamma];\Delta;\mathbf{int}\vdash\S A}}$$

It is immediate that the erasure of iteration is the usual representation of iteration in $\mathbb{F}$; however, very few actual iterations of $\mathbb{F}$ can be obtained this way.
The types $\mathbf{list}^{\mathbf{k}}$, to be defined below, have similar primitives, including a notion of iteration.

Coercions
Observe that any sequent $1^{k_1};\ldots;1^{k_p};\Gamma \vdash A$ can be replaced with $1^k;\Gamma \vdash A$, provided $k \geqslant k_1,\ldots,k_p$. We shall content ourselves with a weaker typing of integers, namely $1^p \vdash \S^q\mathbf{int}$ (in general $p = q$, but we do not mind about the actual value of p. A n-ary function from integers to integers will be given a type $1^p;\mathbf{int};\ldots;\mathbf{int} \vdash \S^q\mathbf{int}$.
The successor function is naturally typed $\mathbf{int} \vdash \mathbf{int}$, which can be replaced with $!^k\mathbf{int} \vdash !^k\mathbf{int}$, and therefore by $1 \vdash !^k\mathbf{int} \multimap !^k\mathbf{int}$. The integer $\overline{0}$ can be given the type $1 \vdash \mathbf{int}$, hence the type $1^k \vdash !^k\mathbf{int}$, and also the type
$1^{k+1} \vdash \S!^k\mathbf{int}$. We are in position to apply iteration and we get a function which

is typed $[1]; 1^{k+1}; \mathbf{int} \vdash \S!^k\mathbf{int}$, which can be replaced with $1^{k+1}; \mathbf{int} \vdash \S!^k\mathbf{int}$. This function is essentially the identity on integers, but it changes the type, and we call it a *coercion*.
In a similar way, we can define coercions of type $1^{p+q}; \mathbf{int} \vdash \S^p!^q\mathbf{int}$, when $p \neq 0$. An immediate consequence is that the multiplication can be given a more even type : replace $!\mathbf{int}; \mathbf{int} \vdash \S\mathbf{int}$ with $\S!\mathbf{int}; \S\mathbf{int} \vdash \S^2\mathbf{int}$, and compose with the coercions $1^2; \mathbf{int} \vdash \S!\mathbf{int}$ and $!1; \mathbf{int} \vdash \S\mathbf{int}$, in order to get
$1^2; !1; \mathbf{int}; \mathbf{int} \vdash \S^2\mathbf{int}$, which can be simplified into $1^2; \mathbf{int}; \mathbf{int} \vdash \S^2\mathbf{int}$.
It is then easy to see that, if $f(x_1, \ldots, x_n)$ and $g(y, y_1, \ldots, y_m)$ have been attributed types $1^p; \mathbf{int}; \ldots; \mathbf{int} \vdash \S^p\mathbf{int}$ and $1^q; \mathbf{int}; \ldots; \mathbf{int} \vdash \S^q\mathbf{int}$, then the function $g(f(x_1, \ldots, x_n), y_1, \ldots, y_m)$ can be given a similar type, namely
$1^{p+q}; \mathbf{int}; \ldots; \mathbf{int} \vdash \S^{p+q}\mathbf{int}$. In other terms, all polynomials in which each variable occurs exactly once can be typed.

Weakening and contraction
In order to get all polynomials, we must be able to represent dummy dependencies, and repetition of variables, i.e. weakening and contraction for $\mathbf{int}$.
Weakening can be defined as a sum with a function which is identically 0 :

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha \multimap \alpha \vdash \alpha \multimap \alpha}{\S(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)}}{\S(\alpha \multimap \alpha); !(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)} \quad \dfrac{\dfrac{\alpha \vdash \alpha}{\vdash \alpha \multimap \alpha}}{[1] \vdash !(\alpha \multimap \alpha)}}{[1]; \mathbf{int}_\alpha; !(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)}}{[1]; \mathbf{int}_\alpha \vdash \mathbf{int}_\alpha}$$

etc.
Contraction is obtained by composition with a diagonal map, i.e. the function $diag(n) = < n, n >$. For this observe that the successor induces a map of type $\mathbf{int} \otimes \mathbf{int} \vdash \mathbf{int} \otimes \mathbf{int}$ (corresponding to the function
$f(< n, n >) = < n+1, n+1 >$) which can be retyped $1; \mathbf{int} \otimes \mathbf{int} \vdash \mathbf{int} \otimes \mathbf{int}$, and 0 induces an object of type $\mathbf{int} \otimes \mathbf{int}$. By iteration, we get a function of type $[1]; \mathbf{int} \vdash \S\mathbf{int} \otimes \mathbf{int}$, corresponding to the map $f(n) = < n, n >$. Now, if we compose with the diagonal map, it is clear that we can identify variables (in general the integer k will increase).
So all polynomials can be given a type $1^k; \mathbf{int}; \ldots; \mathbf{int} \vdash \S^k\mathbf{int}$, and we can even fix the value of k when the degree is known.
Similar weakening and contraction maps are available for the types $\mathbf{list^k}$ to be defined below, in particular for $\mathbf{bint}$.

The predecessor
Last, but not least, we must type the predecessor, i.e. the function *pred* such

that $pred(0)=0, pred(n+1)=n$. The predecessor gets the type $!1;\mathbf{int}\vdash\mathbf{int}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha\vdash\alpha}{\alpha^\dagger\vdash\alpha}\,\&_1\vdash \qquad \dfrac{\alpha\vdash\alpha\quad\alpha\vdash\alpha}{\alpha\vdash\alpha^\dagger}}{\alpha^\dagger\multimap\alpha^\dagger;\alpha\vdash\alpha}}{\alpha^\dagger\multimap\alpha^\dagger\vdash\alpha\multimap\alpha}}{\S(\alpha^\dagger\multimap\alpha^\dagger)\vdash\S(\alpha\multimap\alpha)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha\vdash\alpha}{1;\alpha\vdash\alpha}}{\overline{1,\alpha\multimap\alpha;\alpha^\dagger\vdash\alpha}}\,\&_2\vdash \quad \dfrac{\dfrac{\alpha\vdash\alpha\quad\alpha\vdash\alpha}{\alpha\multimap\alpha;\alpha\vdash\alpha}}{\overline{1,\alpha\multimap\alpha;\alpha^\dagger\vdash\alpha}}\,\&_2\vdash}{1,\alpha\multimap\alpha;\alpha^\dagger\vdash\alpha^\dagger}}{1,\alpha\multimap\alpha\vdash\alpha^\dagger\multimap\alpha^\dagger}}{[1];[\alpha\multimap\alpha]\vdash!(\alpha^\dagger\multimap\alpha^\dagger)}}{\overline{!1;!(\alpha\multimap\alpha)\vdash!(\alpha^\dagger\multimap\alpha^\dagger)}}}{\dfrac{!1;\mathbf{int}_{\alpha^\dagger};!(\alpha\multimap\alpha)\vdash\S(\alpha\multimap\alpha)}{!1;\mathbf{int}_{\alpha^\dagger}\vdash\mathbf{int}_\alpha}}$$

etc., with $\alpha^\dagger=\alpha\mathbin{\&}\alpha$. The basic idea is to iterate, instead of f of type $\alpha\multimap\alpha$, the function f' in $\alpha^\dagger\vdash\alpha^\dagger$ such that $f'(x)=<x,f(x)>$ [10]. Eventually the first projection of the result is kept. Similar functions for $\mathbf{list^k}$ can be defined.

2.5.3 Some data types

Familiar data types as well as the basic operations on them can be represented in **LLL**. We shall only need a type with n elements $\mathbf{bool^n}$ and the type of lists of tokens taken among m tokens, $\mathbf{list^m}$.

Booleans

The type $\mathbf{bool^k}$ is defined as $\forall\alpha.\ \S(\alpha\mathbin{\&}\ldots\alpha\multimap\alpha)$; there are k occurrences of α to the left, and we agree on some (irrelevant) bracketing convention. In particular, $\mathbf{bool^2}$ (written more simply $\mathbf{bool}$) is $\forall\alpha.\ \S(\alpha\mathbin{\&}\alpha\multimap\alpha)$. Its erasure $\forall\alpha.\ \alpha\wedge\alpha\Rightarrow\alpha$ is one of the standard representations of booleans in $\mathbb{F}$. We can define proofs $\mathbf{b}_1,\ldots,\mathbf{b}_k$ of $\mathbf{bool}$, by starting with one of the k canonical proofs of $\alpha\mathbin{\&}\ldots\alpha\vdash\alpha$, and ending with $\multimap$, $\S$ and $\forall$-rules. Typically the boolean « false », $\mathbf{b}_2$ is :

$$\dfrac{\dfrac{\dfrac{\dfrac{\alpha\vdash\alpha}{\alpha\mathbin{\&}\alpha\vdash\alpha}\,\&_2\vdash}{\vdash\alpha\mathbin{\&}\alpha\multimap\alpha}}{\vdash\S(\alpha\mathbin{\&}\alpha\multimap\alpha)}}{\vdash\mathbf{bool}}$$

whose erasure is the standard term $\Lambda\alpha\ \ \lambda x^{\alpha\&\alpha}\ \ \pi_2(x)$, which represents « false » in $\mathbb{F}$.

If then else

We can give a type $1^1;\mathbf{bool^k};A;\ldots;A\vdash A$ to the k-ary version of « if... then

---

[10] f' is not quite linear in f, which is reflected by the 1 in the sequent $1,\alpha\multimap\alpha\vdash\alpha^\dagger\multimap\alpha^\dagger$

... else ... » , when A is a data type (it works when A is a boolean type or a type of lists). We give an example when A is **int** and $k = 2$, i.e. we try to "type" the function $f(true, n, m) = n, f(false, n, m) = m$.

$$
\begin{prooftree}
\AxiomC{$\alpha \vdash \alpha$}
\AxiomC{$\alpha \vdash \alpha$}
\AxiomC{$\alpha \vdash \alpha$}
\BinaryInfC{$\alpha; \alpha \multimap \alpha \vdash \alpha$}
\UnaryInfC{$\alpha; \alpha \multimap \alpha; 1 \vdash \alpha$}
\AxiomC{$\alpha \vdash \alpha$}
\AxiomC{$\alpha \vdash \alpha$}
\BinaryInfC{$\alpha; \alpha \multimap \alpha \vdash \alpha$}
\UnaryInfC{$\alpha; 1; \alpha \multimap \alpha \vdash \alpha$}
\doubleLine
\BinaryInfC{$\alpha; 1, \alpha \multimap \alpha; 1, \alpha \multimap \alpha \vdash \alpha \,\&\, \alpha$}
\BinaryInfC{$\alpha \,\&\, \alpha; 1, \alpha \multimap \alpha; 1, \alpha \multimap \alpha; \alpha \vdash \alpha$}
\UnaryInfC{$\alpha \,\&\, \alpha; 1, \alpha \multimap \alpha; 1, \alpha \multimap \alpha \vdash \alpha \multimap \alpha$}
\doubleLine
\UnaryInfC{$[1]; \mathbf{bool}_\alpha; \S(\alpha \multimap \alpha); \S(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)$}
\AxiomC{$\alpha \multimap \alpha \vdash \alpha \multimap \alpha$}
\UnaryInfC{$[\alpha \multimap \alpha] \vdash !(\alpha \multimap \alpha)$}
\BinaryInfC{$[1]; \mathbf{bool}_\alpha; \mathbf{int}_\alpha; \S(\alpha \multimap \alpha); [\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha)$}
\AxiomC{$\alpha \multimap \alpha \vdash \alpha \multimap \alpha$}
\UnaryInfC{$[\alpha \multimap \alpha] \vdash !(\alpha \multimap \alpha)$}
\doubleLine
\BinaryInfC{$[1]; \mathbf{bool}_\alpha; \mathbf{int}_\alpha; \mathbf{int}_\alpha; [\alpha \multimap \alpha] \vdash \S(\alpha \multimap \alpha)$}
\UnaryInfC{$[1]; \mathbf{bool}_\alpha; \mathbf{int}_\alpha; \mathbf{int}_\alpha; !(\alpha \multimap \alpha) \vdash \S(\alpha \multimap \alpha)$}
\UnaryInfC{$[1]; \mathbf{bool}_\alpha; \mathbf{int}_\alpha; \mathbf{int}_\alpha \vdash \mathbf{int}_\alpha$}
\end{prooftree}
$$

etc., with $\mathbf{bool}_\alpha = \S(\alpha \,\&\, \alpha \multimap \alpha)$. Weakening and contraction on $\mathbf{bool^k}$ can be defined in terms of generalized « if... then ... else ... ».

Lists

We define $\mathbf{list^k}$ to be $\forall\alpha.\ (!(\alpha \multimap \alpha) \multimap (\ldots \multimap !(\alpha \multimap \alpha)\ldots)) \multimap \S(\alpha \multimap \alpha)$, with k occurrences of $!(\alpha \multimap \alpha)$ to the left. So $\mathbf{list^1}$ is just **int**, and $\mathbf{list^2}$ is abbreviated into **bint** (binary integers).

We discuss the type **bint**, but our discussion applies to any type $\mathbf{list^k}$. First we observe that the empty list **emptylist** and more generally any finite list of digits 0 and 1 can be encoded by a proof of **bint**. This is more or less obvious, since $\mathbf{bint}^-$ is the usual $\mathbb{F}$-translation of binary lists. Concatenation of lists, $\frown$, can be represented by a proof of $\mathbf{bint}; \mathbf{bint} \vdash \mathbf{bint}$, which is basically a binary version of $+$, and that we therefore skip. In particular the two successor functions $\cdot \frown 0$ and $\cdot \frown 1$ can both be given the type $\mathbf{bint} \vdash \mathbf{bint}$.

An important function is the « kind of list », of type $!1; \mathbf{bint} \vdash \mathbf{bool^3}$. On the empty list it yields the value $\mathbf{b}_1$, on a list ending with 0, it yields the value $\mathbf{b}_2$, and on a list ending with 1, it yields the value $\mathbf{b}_3$: let $\alpha^* = (\alpha \,\&\, \alpha) \,\&\, \alpha$, and introduce three proofs Π_f, Π_g and Π_h of $\alpha^* \vdash \alpha^*$ respectively corresponding to the functions $f(x) = << \pi_1(\pi_1(x)), \pi_1(\pi_1(x)) >, \pi_1(\pi_1(x)) >$,
$g(x) = << \pi_2(\pi_1(x)), \pi_2(\pi_1(x)) >, \pi_2(\pi_1(x)) >$,

$h(x) =<< \pi_2(x), \pi_2(x) >, \pi_2(x) >$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\alpha \vdash \alpha \qquad \begin{array}{c}\vdots\,\Pi_f\\ \alpha^* \vdash \alpha^*.\end{array}}{\alpha^* \multimap \alpha \vdash \alpha^* \multimap \alpha}}{\S(\alpha^* \multimap \alpha^*) \vdash \S(\alpha^* \multimap \alpha)} \quad \dfrac{\begin{array}{c}\vdots\,\Pi_h\\ \alpha^* \vdash \alpha^*.\end{array}}{\overline{[1] \vdash !(\alpha^* \multimap \alpha^*)}}}{[1]; !(\alpha^* \multimap \alpha^*) \multimap \S(\alpha^* \multimap \alpha^*) \vdash \S(\alpha^* \multimap \alpha)} \quad \dfrac{\begin{array}{c}\vdots\,\Pi_g\\ \alpha^* \vdash \alpha^*.\end{array}}{\overline{[1] \vdash !(\alpha^* \multimap \alpha^*)}}}{\dfrac{[1]; \mathbf{bint}_{\alpha^*} \vdash \S(\alpha^* \multimap \alpha)}{\overline{!1; \mathbf{bint} \vdash \S(\alpha^* \multimap \alpha)}}}}{!1; \mathbf{bint} \vdash \mathbf{bool}}$$

with $\mathbf{bint}_\alpha =!(\alpha \multimap \alpha) \multimap (!(\alpha \multimap \alpha) \multimap \S(\alpha \multimap \alpha))$.
Among the functions connected with $\mathbf{list^k}$ are all the functions $\mathbf{list^f}$, of type $\mathbf{list^k} \vdash \mathbf{list^{k'}}$, induced by a map f from $\{1, \ldots, k\}$ to $\{1, \ldots, k'\}$. They are easily defined, mainly by structural manipulations. Three important examples :

- The (unique) function $\mathbf{list^f}$ from **bint** to **int** identifies the two digits, and produces a tally integer : it will be used for the length of the input of a Turing machine ;
- When $k \geqslant 2$, the function from **bint** to $\mathbf{list^k}$ that identifies a binary integer with a k-list : it will be used for the input tape of a Turing machine ;
- When $k \geqslant 2$, the function from $\mathbf{list^k}$ to **bint** that replaces any digit distinct from $0, 1$ with 0 : it will be used for the output of a Turing machine.

2.5.4 Polytime functions

Turing machines
Consider a (deterministic) Turing machine using p symbols and with q states. The current configuration can be represented by three data :

- a list dealing with the leftmost part of the tape (up to the position of the head)
- a list dealing with the right part of the tape, in reverse order
- the current state

The type $\mathbf{Tur^{p,q}} = \mathbf{list^p} \otimes \mathbf{list^p} \otimes \mathbf{bool^q}$ can therefore be used to represent any configuration of the machine. The instructions of the machine depend on reading the last symbol of one list, (including testing whether or not a list is empty), and also depend on the current state. From what precedes, it is possible (by eventually adding new instructions so that the machine can never stop), to represent a Turing machine by a proof of $!1; \mathbf{Tur^{p,q}} \vdash \mathbf{Tur^{p,q}}$: just use successors, predecessors, « kind of list », and generalized « if … then … else… ».

Inputs and outputs
We assume that our inputs are binary integers, i.e. that the digits 0 and 1 belong to the p legal symbols of the tape. The input (initial configuration) can therefore be expressed by means of a map of type $\mathbf{bint} \vdash \mathbf{Tur^{p,q}}$ which maps a binary list s into the 3-tuple $< s, \mathbf{emptylist}, \sigma >$, where σ is the initial state. When the expected runtime of the machine is over, we may also decide to read off the output, i.e. we need to represent the map $f(< s, t, \tau >) = s'$, where s' is obtained from s by replacing any symbol distinct from $0, 1$ by 0. Such a function is easily obtained by means of a function $\mathbf{list^f}$ and of the weakening facilities on our data types.

Run of a Turing machine
Assume that we are given a time θ (represented by a tally integer of type **int**), an initial input s of type **bint**, and a Turing machine of type $!1; \mathbf{Tur^{p,q}} \vdash \mathbf{Tur^{p,q}}$. Then running the machine for θ steps from the initial input, can be represented by means of an iteration. As a function of θ, s it may receive the type $1^2; \mathbf{int}; \S\mathbf{bint} \vdash \S\mathbf{Tur^{p,q}}$ and therefore (using the coercion map $!1; \mathbf{bint} \vdash \S\mathbf{bint}$) also the type $1^2; \mathbf{int}; \mathbf{bint} \vdash \S\mathbf{Tur^{p,q}}$. The result at time θ (if we stop the machine after θ steps) can be written as a function of θ, s of type $1^2; \mathbf{int}; \mathbf{bint} \vdash \S\mathbf{bint}$.

Polytime machines
A polytime machine is a machine with a polynomial clock, which stops after $P(\sharp(s))$ steps, where $\sharp(s)$ is the size of the input, and P is a given polynomial ; when $P(\sharp(s))$ steps have been executed, then we print out the result. Now observe that P can be given a type $1^k; \mathbf{int} \vdash \S^k\mathbf{int}$, and using the (unique) map $\mathbf{list^f}$ from **bint** into **int**, the function $P'(s) = P(\sharp(s))$ can be given the type $1^k; \mathbf{bint} \vdash \S^k\mathbf{int}$. By composition with the runtime function, we get the type $1^{k+2}; \mathbf{bint}; \S^k\mathbf{bint} \vdash \S^{k+1}\mathbf{bint}$ to represent the function $\varphi(s, s')$ which is the result of the computation after $P'(\sharp(s))$ steps with the input s'. Using the contraction facility on **bint** we can make $s = s'$ and replace this type with $1^{k+3}; \mathbf{bint} \vdash \S^{k+2}\mathbf{bint}$. If we insist on having the same integer on both sides, we can, using the coercion of type $1^{k+3}; \S^{k+2}\mathbf{bint} \vdash \S^{k+3}\mathbf{bint}$, replace this type with $1^{k+3}; \mathbf{bint} \vdash \S^{k+3}\mathbf{bint}$.

2.5.5 The representation theorem

Theorem 1

Any polytime function from binary lists to binary lists can be represented in **LLL** *as a proof of a formula* $1^k; \mathbf{bint} \vdash \S^k\mathbf{bint}$

PROOF. — This is obvious from what precedes. The algorithm can be executed in $\mathbb{F}$, but also as a proof-net, in which case the output is a proof-net with conclusions $\S^k\mathbf{bint}; \bot^k$, and the $\bot^k$, which eventually comes from 0-ary $\bot$-links, can be ignored. □

2.5.6 The user's viewpoint

Let us admit that this works, without being especially friendly. Since this paper is concerned with showing that **LLL** is intrinsically polytime, this subsection was concerned with a rather marginal question : to show that it was strictly polytime, i.e. that any polytime function could be typed inside the system. So we didn't care much about the potential users of such a system. Surely this practical aspect should be developed, under the form of a typed λ-calculus, analogous to system $\mathbb{F}$. The best would be a system of pure λ-calculus with typing declarations in **ILLL**.

But this is not the only possibility : a less conservative option would be to exploit the classical symmetries of **LLL**, which have a lot of interesting consequences (for instance, using the fact that $\alpha \multimap \alpha$ is isomorphic to $\alpha^\perp \multimap \alpha^\perp$, which induces an isomorphism between $\mathbf{bint}_{\alpha^\perp}$ and $\mathbf{bint}_\alpha$, we get the proof :

$$\dfrac{\dfrac{\mathbf{bint}_{\alpha^\perp} \vdash \mathbf{bint}_\alpha}{\mathbf{bint} \vdash \mathbf{bint}_\alpha}}{\mathbf{bint} \vdash \mathbf{bint}}$$

whose action is to reverse a list).

3 PROOF-NETS FOR LLL

Cut-elimination in sequent calculus is unmanageable -especially in presence of additive features- : too many permutations of rules occur, and the counting of these permutations blurs the actual complexity of the process. This is why we choose to use *proof-nets* to prove the main theorem of this paper. Our basic reference will be [4] where the proof-net technology is expounded. We shall therefore content ourselves with modifying the definitions of [4] so as to take care of the specificities of **LLL**. We adopt the definitions and conventions of this paper, in particular we shall very often speak of formulas to mean "occurrences of formulas". We shall ignore the additive constants $\top$ and 0 on the double grounds that they play little role and that they can be handled anyway by means of second-order definitions in case we badly insist to keep them. This will save a lot of inessential details.

3.1 Proof-nets with multiplicative/additive conclusions

We first liberalize the condition about the weights of conclusions in definition 3 of [4]. Let $\Gamma = [\Delta]; \mathbf{A_1}; \ldots; \mathbf{A_n}$ be a sequent. Then a proof-structure will be declared to have the conclusion Γ when its conclusions are the formulas (discharged or not) listed in Γ and furthermore, for each $\mathbf{A_i}$ the sum of the weights of the formulas of $\mathbf{A_i}$ is equal to 1. This is equivalent to saying that, after applying *ad hoc* $\oplus$-links to the formulas of $\mathbf{A_i}$, then we obtain a proof-structure in the sense of section 3 of [4].

We consider the following exponentials links :

- The ?-link, with n unordered premises, which are all occurrences of the same discharged formula $[A]$, and with conclusion $?A$;
- The !-box, which is a generalized axiom whose (unordered) conclusions are $[A_1], \ldots ; [A_n]; !B$. This link is called a box because in order to use it, one has to give a proof-net Θ whose conclusions are $A_1, \ldots, A_n; B$; our conventions about proof-structures imply that n is nonzero. A pictural representation of a box is precisely a… box whose contents is Θ and below which the conclusions of the link are written ;
- The §-box, which is a generalized axiom whose (unordered) conclusions are $[A_1]; \ldots ; [A_n]; \S A_{n+1}; \ldots ; \S A_{n+m}$. This link is called a box because in order to use it, one has to give a proof-net Θ whose conclusion is a sequent $\Gamma; \Delta$ without discharged formulas, and such that the formulas occurring in Γ are exactly $A_1, \ldots, A_n$, and Δ is $A_{n+1}; \ldots ; A_{n+m}$. A typical example is that of a proof-net with conclusions $A, B; C; D, E, F; G; H$, which can be used to form a §-box with conclusions $[A]; [B]; [C]; [D]; [E]; [F]; \S G; \S H$, but also a §-box with conclusions $[A]; [B]; \S C; [D]; [E]; [F]; \S G; \S H$.

Weights are subject to the usual conditions ; moreover

- A discharged formula is the conclusion of exactly one link, i.e. one box ;
- If L is a ?-link with premises $[A_1], \ldots, [A_n]$ (occurrences of the same discharged formula), then $w(L) \geqslant w([A_i])$ for $i = 1, \ldots, n$; remember that a default jump, i.e. a formula B such that $w(B) \geqslant w(L)$ must be provided with the link.

The condition for being a proof-net is defined in the obvious way : once a valuation ψ has been selected, one builds a graph whose vertices are those formulas A such that $\psi(w(A)) = 1$. The edges are selected as in [4] ; moreover

- For any ?-link, one draws an edge between the conclusion of the link and any premise of the link which a vertex of the graph, or with the default jump B (this is crucial in case no premise of the link is a vertex of the graph) ;
- For any box with conclusions $A_1, \ldots, A_n$, one draws an edge between A_1 and A_2, A_2 and A_3,… A_{n-1} and A_n. The choice of edges depends on an ordering of the conclusions of the box, but any other ordering would produce an equivalent graph.

Observe that since boxes are built from proof-nets, our condition indeed means that a proof-structure is a proof-net iff it is a proof-net when we consider its boxes as proper axioms, and if the contents of its boxes are in turn proof-nets, etc.

3.2 Sequentialization for LLL

We must first define what it means for a proof in sequent calculus to be a *sequentialization* of a proof-net. This is done without problem, following the lines of [4]. We only need to be careful about the structural maintenance : typically certain formulas of $\vdash \Gamma$ are not present in the proof-net, because they would receive the weight 0. This is the case inside blocks, and for discharged formulas. We can state the :

Theorem 2

Proof-nets are sequentializable, i.e. every proof-net is the sequentialization of at least one sequent calculus proof.

PROOF. — By induction on the *depth*, i.e. the maximum nesting of boxes. If we assume that the inside of all boxes is sequentializable, since the rules for the formation of boxes are the same as the rules for ! and §, then we are left with the problem of sequentializing a usual proof-net with boxes, a question solved in [4], section 3. □

3.3 Cut-elimination for LLL

Since this proof is rather delicate, we suggest to first understand it in the case without additives. Hence there is no notion of weight, the !-boxes have exactly two conclusions, all cuts are ready, and all exponential cuts are special. Moreover the notion of proof-net in this case is akin to the more familiar multiplicative case.

3.3.1 The size and depth of a proof-net

Definition 4

The size $\sharp(L)$ *of a link* L *is defined by :*

- *if* L *is an identity link,* $\sharp(L) = 2$ *;*
- *if* L *is a cut-link,* $\sharp(L) = 0$ *;*
- *if* L *is an exponential box constructed from a proof-net with conclusion* Γ*, then* $\sharp(L) = 1 + s$*, where* s *is the number of semi-colons in* Γ *;*
- *otherwise* $\sharp(L) = 1$.

The size $\sharp(\Theta)$ *of a proof-net* Θ *is the sum of the sizes of the links occurring in it, including what (hereditarily) occurs inside the boxes.*

Definition 5

The depth $\partial(\Theta)$ *of a proof-net* Θ *is the maximum nesting number of boxes in* Θ*. The* depth *of a formula* A *(denoted* ∂A *or* $\partial A/\Theta$*) is the number of boxes containing it : typically, if* Θ *consists of a sole box* $\mathcal{B}$ *made from a proof-net* Θ'*, then the conclusions of* $\mathcal{B}$ *have depth* 0*, whereas the depth of a formula* A *of* Θ' *is given by* $\partial A/\Theta = \partial A/\Theta' + 1$*. One similarly defines the notion of* depth *of a link : typically in the case just considered, the box gets the depth* 0*, whereas* $\partial L/\Theta = \partial L/\Theta' + 1$ *for all links* L *occurring inside* $\mathcal{B}$*. Finally we define the* partial size*, also called d-size,* $\sharp_d(\Theta)$ *to be the sum of the sizes of links of depth* d *in* Θ*, so that* $\sharp(\Theta) = \sharp_1(\Theta) + \ldots + \sharp_n(\Theta)$*, where* n *is the depth of* Θ.

These definitions have been chosen because of their relevance to cut-elimination. But what about the relevance of our size w.r.t. the actual size of a proof-net ?

- The size of a link is almost the number of its conclusions. In Θ, define a function f as follows : if A is not discharged, let$f(A)$ be any link with conclusion A, if $[A]$ is discharged, then it is the conclusion of a box, and A occurs (undischarged) inside the box, and we set $f([A]) = f(A)$. It is easy to see that L occurs in the range of f at most twice the size of L, hence the number of formulas in Θ is bounded by $2\sharp(L)$.

- Cut-links do not contribute to the size ; however if A is the premise of such a link, then A is the conclusion of another link, and it is easy to see that the number of cuts cannot exceed the size.

- The actual size of a net as a graph is therefore linear in the official size, Good News ! However we are not done since the net also involves the boolean weights. But, as observed in [4], these weight can be replaced with a structure of a coherent space between the links and therefore the size of the missing structure is quadratic in the official size of the net.

- Finally the size does not take into account the actual sizes of formulas. Here very little can be done, especially in presence of quantifiers. The most natural viewpoint is to see the formulas as comments, which are erased at runtime, in the same way that the actual execution of a typed λ-term is the execution of its erasure, i.e. of the underlying pure λ-term. In other terms, we work with a kind of *interaction net* à la Lafont, see [9].

This should be enough to convince one that the polynomial bounds obtained below actually induce a polytime algorithm. Concretely, as explained in [4], the substitutions occurring during the additive steps are delayed and those occurring during the quantifier steps are not performed (they can be stored in some auxiliary memory). If the final result should be without additives, then the additive substitutions can be done at the end, producing a cleansing of the graph (all weights become 0 or 1). If the final result is also free from any kind of existential quantifiers, then the formulas can be synthesized in an obvious way, and we have no use for our stack of substitutions.

3.3.2 Cut-elimination : the general pattern

We shall define a *lazy* cut-elimination which terminates in polytime. The result of the procedure (which is Church-Rosser) is cut-free only in certain cases, but this is enough for us.

Let us call a cut *exponential* when the cut-formulas begin with exponentials and both premises are conclusions of exponential links. For non-exponential ready links, the paper [4] defines a linear time cut-elimination procedure : each step of this *basic* procedure strictly shrinks the size of the proof-net (and this remains true with our specific measurement of size).The pattern is as follows :

- In a preliminary round we apply the basic procedure at depth 0, which induces a shrinking of the proof-net at depth 0, the other sizes staying the same ; then the real things begin
- In a first round we work at depths 0 and 1 ; at depth 1 only the basic procedure is allowed, whereas only certain exponential cuts are removed at depth 0. If the original partial sizes were $s_0, \ldots, s_d$, then the new sizes after completing the procedure, will not exceed $s_0, s_0 s_1, \ldots, s_0 s_d$ (and the depth does not increase).
- In a second round we apply a similar procedure at depths 1 and 2 ; this procedure fires no new reduction at depth 0, so that after completing this second round, our partial sizes will not exceed $s_0, s_0 s_1, s_0^2 s_1 s_2, \ldots, s_0^2 s_1 s_d$, and the depth still not increases.
- The d^{th} round occurs at depths $d-1$ and d. When it is completed, nothing more can be done (in the lazy case, we shall be cut-free). The depth of the proof-net is still at most d (it can diminish in the very unlikely situation of erasure of a deeply nested box), and the sizes are now at most $s_0, s_0 s_1, s_0^2 s_1 s_2, s_0^4 s_1^2 s_2 s_3, \ldots, s_0^{2^d} s_1^{2^{d-1}} \ldots s_{d-2}^2 s_{d-1} s_d$. The final size is therefore bounded by s^{2^d}.

It will be easy to see that s^{2^d} actually counts the number of steps, if s is the size and d is the depth : we are therefore polystep in s (when d is fixed, which corresponds to practice). Since the steps are not too big, the actual runtime is polynomial in the number of steps, and the complexity of cut-elimination, for a given depth d will therefore be polytime.

3.3.3 Elimination of exponential cuts

Definition 6

The actual weight *of a discharged formula* $[A]$ *is the weight of the conclusion* A *of the proof-net inside the box. An exponential cut is* special *if it is a ready one and in case one of the premises of the cut is the conclusion of a* ?-*link, then this link is either* 0-*ary or one of its premises has actual weight* 1.

We now explain how to eliminate special cuts : this is the *special procedure*

- §-reduction : take a ready cut between $\S A$ and $\S A^\perp$, both A and $A^\perp$ are conclusions of §-boxes whose contents are proof-nets with respective conclusions $\Gamma; A$ and $A^\perp; \Delta$: in this case we first perform a cut on A between the two proof-nets, yielding a proof-net with conclusion $\Gamma; \Delta$, then we form a §-box with this proof-net.
- Weakening reduction : take a special cut between $!A$ and $?A^\perp$, where $?A^\perp$ is the conclusion of a 0-ary link : in this case we remove the box with conclusion $!A$. This involves the destruction of the conclusions $[B_i]$ of this box, but this only amounts to reducing the arity of some ?-links.

- Contraction reduction : take a special cut between $!A$ and $?A^{\perp}$, where $?A^{\perp}$ is the conclusion of a ?-link with a premise $[{A_i}^{\perp}]$ of actual weight 1. Then $[{A_i}^{\perp}]$ is in turn the conclusion of a box $\mathcal{B}$. $\mathcal{B}$ is made from a proof-net Ξ whose conclusions are ${A_i}^{\perp}; \Delta$, whereas the box $\mathcal{A}$ with $!A$ among its conclusions is made from a proof-net Θ whose conclusions are $\Gamma; A$. By means of a cut between A and ${A_i}^{\perp}$, we can produce a new proof-net Π. Π can be used to produce a new box $\mathcal{C}$ whose conclusions are the same as those of $\mathcal{B}$, except that $[{A_i}^{\perp}]$ is replaced with $[\Gamma]$. In this case we replace $\mathcal{B}$ with $\mathcal{C}$. Observe that new occurrences of $[\Gamma]$ are created, hence the arity of some ?-link will increase.

What about the size during this procedure ? Let us assume that our special cut is of depth 0, and that our original sizes are $s_0, s_1, \ldots, s_d$;

- §-reduction : the size obviously decreases by 2, since three links (two boxes and a cut) counting for $1 + n + 1 + m$ are replaced with two links (one box and a cut), counting for $1 + (n - 1) + (m - 1) + 1$. A new estimate for the partial sizes is $s_0 - 2, s_1, \ldots, s_d$;
- Weakening reduction : the size strictly shrinks, and $s_0 - 1, s_1, \ldots, s_d$ is a very pessimistic majorization of the size of the result ;
- Contraction reduction : at depth 0 the size stays the same, since $\mathcal{C}$ has the same size as $\mathcal{B}$ (this is because there is no semicolon in Γ, so that $\Gamma; \Delta$ has the same number of semi-colons as ${A_i}^{\perp}; \Delta$). But otherwise it increases : more precisely, if the partial sizes of the proof-net Θ are $t_0, t_1, \ldots, t_{d-1}$, then the partial sizes of our new proof-net are exactly $s_0, s_1 + t_0, s_2 + t_1, \ldots, s_d + t_{d-1}$.

In the first round we systematically perform the basic procedure at depth 1 together with the special procedure at depth 0. The point of the basic procedure is that it induces changes of weights inside the boxes, and therefore some conclusions of the proof-nets inside a box receive a new weight 0, in which case some conclusion $[A]$ of the box disappears. This does not affect the size of the box (the number of semicolons stays the same) and since such a conclusion was the premise of some ?-link, this only induces a change of arity of the ?-link. Of course some conclusion of box may get the actual weight 1, which can fire a contraction reduction, etc. By the way no basic reduction at depth 0 can be fired during the first round, and this is why we may assume that they have been done during a preliminary round. Later on, in the second round, no basic reduction at depth 0 or 1 will occur etc.

3.3.4 Bounding the sizes

Bounding the size essentially amounts to considering the first round. We therefore assume that the basic procedure has been completed at depth 0. We also make a simplifying hypothesis, namely that no non-trivial weight remains at depth 0 : this will be the case when we normalize proofs of *lazy sequents*, see below [11].

---

[11] If non-trivial weights appear at depth 0, we can apply the constructions of this subsection to the part of the proof-net which is of size 1

We introduce a *precedence* relation between discharged formulas : $[A] <_1 [B]$ when $[B]$ is conclusion of a !-box $\mathcal{B}$ and the other conclusion of the box $!A^\perp$ is the premise of an exponential cut whose other premise $?A$ is the conclusion of a ?-link, with $[A]$ among its premises. By the correctness criterion, the transitive closure $<$ of precedence is a partial order. We can therefore consider the forest $\mathbb{F}$ of finite sequences $([A_0], \ldots, [A_n])$ of discharged formulas, such that $[A_0]$ is minimal w.r.t. $<$ and $([A_0] <_1 \ldots <_1 [A_n])$. A *discharged block* is a set of discharged formulas $[\mathbf{A}]$ which occur among the conclusion of some exponential box of depth 0, made from a proof-net with conclusion Γ, and such that $[\mathbf{A}]$ is a block of Γ. A *coherent subforest* in $\mathbb{F}$ is a subforest μ of $\mathbb{F}$ such that whenever two sequences $[S], [A], [S']$ and $[S], [B], [S'']$ belong to μ, then either $[A]$ and $[B]$ are the same or they belong to distinct discharged blocks. Given μ, we can define the *multiplicators* $\mu(\mathcal{B})$ for any box $\mathcal{B}$ with discharged conclusions to be the number of sequences in μ such that the last element of the sequence is a conclusion of B ; if $\mathcal{B}$ is a §-box whose conclusions are all of the form $\S A$, let $\mu(\mathcal{B}) = 1$. The *potential sizes* of Θ are defined as follows : for each depth $i \neq 0$, we can write $s_i = \sum s_i^{\mathcal{B}}$, where $\mathcal{B}$ varies through boxes of depth 0 ($s_i^{\mathcal{B}}$ is just the contribution of the proof-net inside $\mathcal{B}$ to i^{th} size). We define $S_0^\mu = s_0, S_1^\mu = \sum \mu(\mathcal{B}) s_1^{\mathcal{B}}, \ldots, S_d^\mu = \sum \mu(\mathcal{B}) s_d^{\mathcal{B}}$.

Proposition 1

Assume that Θ reduces to Π during the first round. Then the potential sizes of Π are not greater than the potential sizes of Θ.

PROOF. — We already know that the size does not increase at depth 0 ; let us check the property at any other depth, typically depth 1, and in the only problematic case, namely the contraction reduction. We start with boxes $\mathcal{A}$ and $\mathcal{B}$ in Θ to produce a box $\mathcal{C}$ which replaces $\mathcal{B}$. If a_1, b_1 are the respective contributions of $\mathcal{A}$ and $\mathcal{B}$ to the 1-size of Θ, then then contribute as $\mu_\Theta(\mathcal{A})a_1 + \mu_\Theta(\mathcal{B})b_1$ to the potential 1-size of Θ, whereas in Π the boxes $\mathcal{A}$ and $\mathcal{B}$ contribute to the potential size as $\nu_\Pi(\mathcal{A})a_1 + \nu_\Pi(\mathcal{C})(a_1 + b_1)$. But since $\mathcal{C}$ is obtained by merging $\mathcal{B}$ with a copy of $\mathcal{A}$, it is easy to construct, given ν a μ such that $\mu(\mathcal{B}) = \nu(\mathcal{C})$ and $\mu(\mathcal{A}) = \nu(\mathcal{C}) + \nu(\mathcal{A})$. This proves the claim. $\square$

By the way observe that any maximal μ will yield $\mu(\mathcal{A}) \geqslant 1$, hence the potential sizes easily exceeds the sizes ; on the other hand observe that coherent subforests are not too big, since they cannot branch at all : this is due to the peculiarities of the ! boxes. Moreover, thanks to acyclicity, the same discharged formula cannot occur twice in the same branch : in other terms $\mu(\mathcal{A})$ cannot exceed the number of roots of μ [12], which is bounded by the number of discharged blocks, and this number is in turn bounded by s_0. This is why the first round yields the bounds $s_0, s_0 s_1, \ldots, s_0 s_d$.

3.3.5 Bounding the runtime

We show below that the runtime is of degree 3 in the size, which will yield polytime complexity of degree 2^{d+2} for our algorithm. It suffices to compute the

---

[12] This is the only point where **ELL** diverges from **LLL** : in **ELL** coherent subforests do branch !

complexity of the first round :

- ▸ The number s_0 dominates both the number of steps of the preliminary round and the number of special steps which are not contraction reductions ; the number $s_0 s_1$ dominates the number of basic steps in the first round. The number of contraction reductions performed during the first round is smaller than the maximum size of a coherent subforest of $\mathbb{F}$, and is therefore less than b_0^2, where b_0 is the number of discharged blocks of depth 0, which is turn is bounded by s_0^2. The number of steps during the first round is therefore easily bounded by $(s_0 + s_1)^2$;
- ▸ However, the number of steps is not the runtime : some steps, typically contraction reductions involve a duplication of the structure, which means that each step can cost at most the *actual size* of the proof-net. We already observed that the actual size is quadratic in the size (which is bounded by $s_0 s$) hence we arrive at a total of $(s_0 s)^2 s^2$ for the first round.

Without being very cautious, we can bound the total runtime by something like $3s^{2^d}$, which is enough for our purpose.
This does not mean that the algorithm cannot be improved. The decomposition in rounds is rather artificial etc. But we are not looking for efficient implementation, just for a proof-system which is intrinsically polytime, and that's it.

3.3.6 Lazy sequents

A formula is said to be *lazy* when it contains neither the symbol & nor higher order existential quantification. A sequent is said to be *lazy* when all the formulas occurring in it are lazy.

Proposition 2

Let Θ be a proof-net without non-exponential ready cut of depth 0, and assume that Θ has a ready cut at depth 0. Then one of the conclusions of Θ is a non-lazy formula of weight 1.

PROOF. — since eigenweights can only be used at a given depth, some eigenweight is used at depth 0, and we can look for a &-link L such that the empire of its conclusion is maximal w.r.t. any valuation. Then the downmost conclusion below this link cannot be the premise of a cut (in which case we can show, as in [4], that the cut would be ready), hence it must be a conclusion, and its weight is bigger that the weight of L, so it is equal to 1. ☐
As a corollary, after the preliminary round, a proof-net whose conclusion is lazy has no non-trivial weight at depth 0.

Proposition 3

After the first round, the proof of a lazy sequent has no cut of depth 0.

PROOF. — assume that the first round is completed, and consider the forest $\mathbb{F}$; if there is still a cut of depth 0, then there is a sequence $([A_0], [A_1])$ in $\mathbb{F}$, and $[A_0]$ is the conclusion of a box $\mathcal{B}$. Since a conclusion of $\mathcal{B}$ is the hereditary premise of an exponential cut and the contraction reduction does not apply, then this conclusion must have a non-trivial weight. Now the proof-net Π, which is in $\mathcal{B}$ has a conclusion with a non-trivial weight, and since the basic procedure has been completed for Π, there is a conclusion C of Π which is non-lazy and of weight 1. This conclusion yields a conclusion of $\mathcal{B}$ and :

- either the conclusion is a formula $\S C$; since this formula is non-lazy, it must be the premise of a cut. . . But the §-reduction would apply, a contradiction ;
- or this conclusion is a formula $!C$, which must also be the premise of a cut. In this case, observe that $C <_1 A_0$, a contradiction ;
- or this conclusion is the premise $[C]$ of a ?-link, which must in turn be the premise of a cut ; in this case $[C]$ is of actual weight 1, and the contraction elimination does apply, a contradiction.

Therefore $\mathbb{F}$ is trivial and Θ is cut-free at depth 0. ☐

Theorem 3

Cut-elimination converges to a (unique) normal form for proofs of lazy sequents ; furthermore, for bounded depth, the runtime is polynomial in the size of the net.

PROOF. — more or less obvious from what precedes. ☐

Observe that application of a function of type $1^k; \mathbf{bint} \vdash \S^k\mathbf{bint}$ to an argument falls into this case : a (cut-free) argument is of depth 1, hence the global depth is the depth of the proof representing the function. Observe that, unlike other approaches, like [10, 6], there is still a complexity bound for arbitrary functionals, something like n^{2^n}, since the size clearly exceeds the depth by 2.

A APPENDIX

A.1 Naive Set-Theory and LLL

We have so far only considered second order propositional **LLL**. But this is not the only possibility :

- We can consider first-order **LLL**, which is straightforward.

- We can also consider **LLL** with first and second order quantifications ; this system would be a natural candidate for a light second order arithmetic. By the way a light first order arithmetic could easily be extracted, but one would have to think twice in front of the difficulties inherent to equality, especially in terms of proof-nets (e.g. certain formulas like $0 \neq 1$ will be equivalent to $\top$, hence the case of $\top$ has first to be fixed) ;

- We can also consider quantifications of any order…

- And last but not least, we can consider Naive Set Theory, which encompasses all kinds of quantification.

In fact, Naive Set Theory has been the starting point of **LLL** : I was looking for a system in which the complexity could be expressed independently of the complexity of the cut-formulas. In particular it would also work for naive set-theory, since there is a well-known (non-terminating, for obvious reasons) cut-elimination procedure for it ; by the way, it had been observed long ago by Grishin [5] that, in the absence of contraction, cut-elimination works [13]. So I decided to translate Russell's paradox into linear logic with exponentials. Using fixpoint facilities (see below) one can produce a new constant A, which has the rules (unary links in terms of proof-nets) : from $?A$ deduce $A^\perp$, from $!A^\perp$ deduce A.

- There is a first possibility for deriving a paradox (here a proof-net with no conclusion), which is based on dereliction : the proof-net has depth 1, but the process of cut-elimination does not converge at depth 0, since the normalization of a cut with dereliction between $?A$ and $!A^\perp$ involves an "opening" of the box with conclusion $!A^\perp$: the contents of this box is "poured" into depth 0, so that the size s_0 no longer shrinks ;

- There is another possibility which does not use dereliction, but the principle $??A \multimap ?A$; in this case, the first round is easily completed, but the handling of the exponential cuts involves the creation of a deeper box, i.e. the size increases.

This is why we restricted to rules whose normalization involves no change of depth.

---

[13] This is not very helpful, since the system without exponentials is awfully inexpressive in terms of computational power

A.1.1 Expressive power of LLLs

Light Naive Set-Theory **LLLs** is defined exactly as **LLL**, but for the quantifiers and the terms :

Definition 7

Terms (T) and formulas (F) are defined as follows

$T = x, y, z, \ldots \; \{x \mid F\}$

$F = T \in U, T \notin U, 1, \bot, 0, \top, !F, \S F, ?F, F \otimes F, F \parr F, F \& F, F \oplus F, \forall x F, \exists x F$

Negation is defined as expected ; in particular, $(T \in U)^{\perp} = T \notin U$ and $(T \notin U)^{\perp} = T \in U$.
The logical rules are modified as follows :

$$\frac{\vdash \Gamma; A}{\vdash \Gamma; \forall x\, A} \; (\textit{for all : } \alpha \textit{ is not free in } \Gamma) \qquad \frac{\vdash \Gamma; A[T/x]}{\vdash \Gamma; \exists x\, A} \; (\textit{there is})$$

$$\frac{\vdash \Gamma; A[T/x]}{\vdash \Gamma; T \in \{x \mid A\}} \; (\in) \qquad \frac{\vdash \Gamma; A[T/x]^{\perp}}{\vdash \Gamma; T \notin \{x \mid A\}} \; (\notin)$$

The representation in terms of proof-nets is straightforward : the $\in$-rules induce two unary links, one with premise $A[T/x]$ and conclusion $T \in \{x \mid A\}$, the other with premise $(A[T/x])^{\perp}$ and conclusion $T \notin \{x \mid A\}$.

A.1.2 Equality

Definition 8

The Leibniz equality $t = u$ is defined by $\forall x (t \in x \multimap u \in x)$; $t \neq u$ is short for $t = u \multimap 0$, a strong form of negation.

Exercise. — Prove the following sequents :

- ▶ $t = u; A[t/x] \vdash A[u/x]$
- ▶ $t = u \vdash u = t$
- ▶ $t = u \vdash (u = v \multimap t = v)$
- ▶ $t = u \vdash 1$
- ▶ $t = u \vdash t = u \otimes t = u$

Definition 9

The singleton $\{t\}$ is defined as $\{x \mid x = t\}$; the pair $\{t, u\}$ is defined as $\{x \mid x = t \oplus x = u\}$; the ordered pair $< t, u >$ is defined as $\{\{t\}, \{t, u\}\}$.

Exercise. — Prove the following sequents

- ▶ $\{t\} = \{t'\} \vdash t = t'$
- ▶ $\{t, u\} = \{t', u'\} \vdash (t = t' \oplus t = u') \otimes (u = t' \oplus u = u')$

- $\{t,u\} = \{t',u'\} \vdash (t = t' \otimes u = u') \oplus (t = u' \otimes u = t') \oplus t = u$
- $\{t,u\} = \{t',u'\} \vdash (t = t' \otimes u = u') \oplus (t = u' \otimes u = t')$
- $\{t\} = \{t',u'\} \vdash t = t' \otimes t = u'$
- $< t,u >=< t',u' > \vdash t = t' \otimes u = u'$

Exercise. — Prove the formula $\{x \mid 0\} \neq \{t\}$; conclude that we can find terms $t_0, \ldots, t_n, \ldots$ such that $t_i \neq t_j$ is provable for $i \neq j$.

As a consequence of the exercise, it is possible to represent certain features of the usual equalitarian predicate calculus :

- we can represent a n-ary function letter f by assigning to it a specific term a_i coming from the previous exercise ; $ft_1 \ldots t_n$ will be represented as $< a_i, t_1, \ldots, t_n >$, using a $n+1$-ary pairing function. It follows from the previous exercises that usual equality axioms are satisfied together with $ft_1 \ldots t_n \neq gu_1 \ldots u_m$ (when f, g are distinct) and $ft_1 \ldots t_n = gu_1 \ldots u_n \multimap t_1 = u_1 \otimes \ldots \otimes t_n = u_n$
- we can also represent predicates by means of fixed variables (generic constants) and by means of the pairing function : $pt_1 \ldots t_n$ becomes $(t_1, \ldots, t_n) \in x$, where x is a variable assigned to p
- as a consequence, we have access to a representation of binary strings : for this we only need a constant ϵ and two unary successors S_0, S_1. Equality axioms, as well as inequalities $S_0 t \neq S_1 u, S_0 t \neq \epsilon, S_1 t \neq \epsilon$ are provable, as well as $S_i t = S_i u \multimap t = u$ for $i = 1, 2$.

A.1.3 Fixpoints

In order to formulate the fixpoint property, we introduce the following notation : the substitution of an abstraction term $\lambda x_1 \ldots x_n.B$ for a n-ary predicate symbol P in the formula A consists in replacing any atom $Pt_1 \ldots t_n$ of A by $B[t_1/x_1, \ldots, t_n/x_n]$.

Proposition 4

Let A be a formula in the language of **LLLs** *augmented by means of a n-ary predicate P, and let $x_1, \ldots, x_n$ be variables, so that we can write our formula $A[P, x_1, \ldots, x_n]$; then there is a formula B (depending on $x_1, \ldots, x_n$) such that the equivalence (i.e. both linear implications) between $A[\lambda x_1 \ldots x_n.B[x_1, \ldots, x_n]]$ and B is provable.*

Proof. — This is a straightforward imitation of Russell's paradox (already used in the fixpoint theorem of λ-calculus). For instance, let us assume that $n = 1$; then we can form $t := \{z \mid \exists x \exists y. z =< x,y > \otimes A[\lambda w. < w,y > \in y, x]\}$. Then $< x,t > \in t$ is provably equivalent with $A[\lambda w. < w,t > \in t, x]$ and we are done. □

As a consequence we get the possibility of defining various partial recursive functions. Typically, take for instance the exponential function (defined on binary strings, i.e. in number base 2) then we can get a two variable formula B such that $B[s,t]$ expresses that $t = 2^s$.

This is enough to convince one that **LLLs** bears all the features of a light arithmetic. In particular all numerical functions implicit in proofs made in this system will be polytime computable.

A.2 Elementary Linear Logic

Elementary Linear Logic arises as the alternative solution to the complexity problem at stake. It syntax does not contain § (or rather does not need it). The rule for ! is liberalized into :

$$\frac{\vdash B_1 \mid \ldots \mid B_n; A}{\vdash [B_1]; \ldots; [B_n]; !A} \quad (\textit{of course})$$

where the symbols $B_1, \ldots, B_n$ are separated by commas or semicolons.
As a consequence, the sequent $!A; !(A \multimap B) \vdash !B$ becomes provable (equivalently $!A; !B \vdash !(A \otimes B)$ is provable). Integers can now be represented by the type $\forall\alpha.\ !(\alpha \multimap \alpha) \multimap !(\alpha \multimap \alpha)$: the tally integers can be given this type, which was not the case for **LLL**. The representation results of **LLL** persist (replace § by ! everywhere). We can also get rid of the irritating markers 1^k in the representation theorem, since the rule for ! is now valid with an empty context. But new functions arise, namely exponentials. This is due the fact that multiplication can now be given the type $\mathbf{int}; \mathbf{int} \vdash \mathbf{int}$. If we feed the first argument with the integer $\overline{2}$, we can type duplication with $\mathbf{int} \vdash \mathbf{int}$, and as soon as duplication can be given a type $A \vdash A$, then we can iterate it, yielding a representation of the exponential function. The exponential can therefore be typed with $\mathbf{int} \vdash !\mathbf{int}$, and towers of exponentials with the type $\mathbf{int} \vdash !^k\mathbf{int}$. The same holds for other data types, and therefore we conclude that all elementary functions (i.e. functions whose runtime is bounded by a tower of exponentials) can be typed in **ELL**.
Is this optimal ? The proof of lazy normalization still works [14], but for the fact that coherent subforests are not so simple, since they may branch. The multiplication factor involved in the first round is no longer s_0 but depends exponentially on s_0, something like $s_0^{s_0}$. Completing the process will therefore cost a tower of exponentials, the height of the tower depending on the depth of the proof-net. Hence normalization is elementary in the size of the input, when the depth is given. This is analogous to the familiar bounds for predicate calculus/simply typed λ-calculus, but here the height of the tower does not depend on the cut-formula, but on more hidden parameter, the depth.
It is also possible to build a naive set-theory **ELLs**. Its expressive power is considerably bigger than before, since the exponential function plays a decisive role in mathematics. This induces a strange system which can both formalize a bunch of mathematics, and which admits definition by fixpoint. Such a system seems to be the optimal candidate for formalization of AI.

---

[14] However, as observed by Kanovich [7] the system must be modified in order to accomodate full (i.e. non-lazy) cut-elimination. Observe that the complexity of the full process is still elementary, since normalizing non-ready cuts increases the size by an exponential factor.

A.3 Questions

Semantics

What is the natural semantics for **LLL** ? The recent work of Kanovich, Okada and Scedrov [8] proposes a phase semantics for **LLL** ; unfortunately this does not address the question of a denotational semantics (i.e. some sort of coherent spaces) specific of **LLL** which is presumably the deepest problem connected with our new system : for the first time polynomial time appears as the result of the free application of logical principles which are in no means contrived, hence a denotational semantics of **LLL** would be a general semantics of polytime. This might be very rewarding : remember that polytime has been characterized in many ways, but always through *presentations* « A function is polytime iff it can be obtained by means of… », and nobody knows how to deal with a presentation. On the other hand a semantic characterization would insist on something like preservation properties etc. that a mathematician can more easily reason about.

The connective « § »

This strange connective has been introduced to compensate two things, namely the want of dereliction, but also the failure of the principle [V] : $!A \otimes !B \vdash !(A \otimes B)$, which is essential in the representation of data types. Surely $\S A \otimes \S B \vdash \S(A \otimes B)$ holds and $\S(A \oplus B) \vdash \S A \oplus \S B$ fails, but there are principles (typically the self-duality of the connective) that have been added on the sole grounds of their simplifying character. Later investigations (in particular semantical ones) could help to clarify this question [15]. In a similar way, the fact that !1 is not provable is backed by good taste (!1 looks like the 0-ary case of [V]), but by no deep intuition.

Completeness

In some sense **LLL** and **ELL** are complete, since the complexity bounds are here once for all. This is even more conspicuous with their naive set-theoretic extensions : what could be more powerful than unrestricted comprehension ? In some sense the theorems, the algorithms coming from these systems should be absolute. Is it possible to make sense of this informal remark ?

Execution

In our systems, the runtime is known in advance, depending only on the depth and size. We could seek an untyped calculus, with a notion of depth, and for each depth d a function $r_d(.)$ with the following property : after $r_d(\sharp(t))$ steps, then we reach either a normal form or a deadlock. There should be two solutions, corresponding to polytime and elementarity.

---

[15] The recent phase semantics [8] concludes to a non self-dual connective.

BIBLIOGRAPHY

[1] A. Asperti. **Special light linear logic**. Manuscript, http://hypatia.dcs.qmw.ac.uk/authors/A/AspertiA/SLLL.ps.gz, 1997.

[2] avec *A. Scedrov* et *P. Scott.* **Bounded linear logic : a modular approach to polynomial time computability : extended abstract**. In Buss and Scott, editors, *Feasible mathematics*, pages 195–209, Boston, 1990. *Birkhäuser.*

[3] J.-Y. Girard. **Linear logic**. *Theoretical Computer Science*, 50:1–102, 1987.

[4] J.-Y. Girard. **Proof-nets : the parallel syntax for proof-theory**. In Ursini and Agliano, editors, *Logic and Algebra*, New York, 1996. *Marcel Dekker.*

[5] V.N. Grishin. **Predicate and set-theoretic calculi based on logics without contractions**. *Math. USSR Izvestiya*, 18:41–59, 1982.

[6] G.G. Hillebrand, P.C. Kanellakis, and H.G. Mairson. **Database query languages embedded in the typed lambda calculus**. In *Proc. 8-th Annual IEEE Symposium on Logic in Computer Science, Montreal*, pages 332–343, June 1993.

[7] M. Kanovich and Y. Lafont. **On Elementary Linear Logic**. in preparation. 1997.

[8] M. Kanovich, M. Okada, and A. Scedrov. **Phase semantics for light linear logic**. *Electronic Notes in Theoretical Computer Science*, Proceedings of the Thirteenth Conference on the Mathematical Foundations of Programming Semantics, Pittsburgh, Pennsylvania, USA, 1997:http://www1.elsevier.nl/mcs/tcs/pc/Menu.html, 1997.

[9] Y. Lafont. **From proof-nets to interaction nets**. In Girard, Lafont, and Regnier, editors, *Advances in Linear Logic*, Cambridge, 1995. *Cambridge University Press.*

[10] D. Leivant. **A foundational delineation of poly-time**. *Information and Computation*, 110:391–420, 1994. (Special issue of selected papers from LICS'91, edited by G. Kahn.).

[11] D. Leivant and J.-Y. Marion. **Lambda calculus characterizations of poly-time**. *Fundamenta Informaticae*, 19:167–184, 1993. (Special Issue: Lambda Calculus and Type Theory, edited by J. Tiuryn.).