Parsimonious Types and Non-uniform Computation

Damiano Mazza[1] and Kazushige Terui[2]

[1] CNRS, UMR 7030, LIPN, Université Paris 13, Sorbonne Paris Cité
Damiano.Mazza@lipn.univ-paris13.fr
[2] RIMS, Kyoto University
terui@kurims.kyoto-u.ac.jp

Abstract. We consider a non-uniform affine lambda-calculus, called parsimonious, and endow its terms with two type disciplines: simply-typed and with linear polymorphism. We show that the terms of string type into Boolean type characterize the class L/poly in the first case, and P/poly in the second. Moreover, we relate this characterization to that given by the second author in terms of Boolean proof nets, highlighting continuous affine approximations as the bridge between the two approaches to non-uniform computation.

1 Introduction

This paper is a contribution to the line of research known as *implicit computational complexity* (ICC), whose aim is to characterize complexity classes by means of logical systems and programming languages (which are intimately related via the Curry-Howard correspondence). Seminal work concerning this methodology includes [3, 11, 9]. The highlight of this paper consists in dealing with *non-uniform* complexity classes, which very few ICC works have considered so far.

Computation is usually performed by a single machine or program which *uniformly* works on inputs of arbitrary length. On the other hand, some models of computation, such as Boolean circuits, only work for inputs of fixed length. Thus, to compute a function $f : \{0,1\}^* \longrightarrow \{0,1\}$, one needs to prepare a *family* $(C_n)_{n\in\mathbb{N}}$ of Boolean circuits, one for each input length n. The non-uniform perspective is important for hardware design and the study of small complexity classes. Two well-known non-uniform classes are P/poly, consisting of languages decided by families of polynomial size Boolean circuits, and L/poly, consisting of those decided by families of polynomial size branching programs (*i.e.*, decision trees with sharing). Such languages may well be non-recursive, but are still useful as they capture the combinatorial essence of P or L. For instance, a potential approach to separating P and NP is to show that NP is not included in P/poly.

We may call the above approach to non-uniform computation *family approach*. There is an alternative, which consists in using a uniform machine having access to arbitrary *advice*, *i.e.*, a fixed family $(w_n)_{n\in\mathbb{N}}$ of strings. Then, P/poly (resp. L/poly) is equivalently defined as the class of languages decided by a deterministic polytime (resp. logspace) Turing machine aided by a polynomial advice, *i.e.*, s.t. $|w_n|$ is polynomial in n. We may call this the *individual approach*.

Non-uniform complexity has been studied in the setting of proofs and functional programming. The second author showed in [19] that P/poly is precisely the class of languages decided by families of polynomial size proof nets of multiplicative linear logic (*family approach*), whereas the first author introduced an infinitary affine λ-calculus in [13] exactly capturing the class $\mathsf{P/poly}$ (*individual approach*).

The present paper combines and extends these previous works. Our goal is twofold: to reconcile the two approaches, and to capture also $\mathsf{L/poly}$.

The starting point is the idea of seeing the exponential modality as some sort of limit, which is quite fruitful and is at the core of several recent developments of linear logic [16, 5] as well as, albeit implicitly, older work on games semantics [1, 15] and intersection types [10]. Following this direction, in [12] the first author introduced an infinitary affine λ-calculus, in which the usual λ-calculus embeds, endowed with a topology such that finite affine terms form a dense subspace and reduction is continuous. This means that computation in the λ-calculus, which is non-linear, may be approximated arbitrarily well by computation on linear affine terms. In particular, since data type values (such as Booleans and strings) are only approximated by themselves, if a λ-term t is given in input a binary string w and $t\,\mathsf{w} \to^{l} \mathsf{b}$ (reduction in l steps) with b a Boolean, then there exists a finite affine term t_0 such that $t_0\mathsf{w} \to^{l} \mathsf{b}$.

However, the "modulus of continuity" of reduction in the λ-calculus is ill-behaved: the size of t_0 may be exponential in l. The contribution of [13] was to find a subset of (infinitary) terms, called *parsimonious*, on which the "modulus of continuity" is polynomial. At this point, if one manages to bound l polynomially in the length of the input w, computation falls within $\mathsf{P/poly}$: the polynomially-sized approximation t_0 of t may be given as advice, and normalization of the finite affine term $t_0\mathsf{w}$ may be done in deterministic polynomial time. In [13], *stratification* (originally due to Girard [9]) was used to obtain such a bound.

Here, instead, the bound will be enforced by types, which have the benefit of allowing us to modulate computational complexity via logical complexity. More specifically, in Sect. 2 we assign types to non-uniform parsimonious terms in two ways: with a simply-typed system called *non-uniform parsimonious logic* $\mathbf{nuPL}$, and with its extension with *linear* polymorphism $\mathbf{nuPL}_{\forall\ell}$. Let now nuPL (resp. $\mathsf{nuPL}_{\forall\ell}$) be the class of languages decided by programs typable in $\mathbf{nuPL}$ (resp. $\mathbf{nuPL}_{\forall\ell}$), and let APN (resp. APN^0) be the class of languages decided by polynomial size proof nets (resp. proof nets of *bounded height*). Our main result is

Theorem 1. $\mathsf{nuPL} = \mathsf{APN}^0 = \mathsf{L/poly}$ *and* $\mathsf{nuPL}_{\forall\ell} = \mathsf{APN} = \mathsf{P/poly}$.

The inclusion $\mathsf{L/poly} \subseteq \mathsf{nuPL}$ (resp. $\mathsf{P/poly} \subseteq \mathsf{nuPL}_{\forall\ell}$) is shown by exhibiting a very natural encoding of branching programs (resp. Boolean circuits) as parsimonious programs (Sect. 3), while $\mathsf{APN}^0 \subseteq \mathsf{L/poly}$ follows from a simple observation on the geometry of interaction (Sect. 4). The inclusions $\mathsf{nuPL} \subseteq \mathsf{APN}^0$ and $\mathsf{nuPL}_{\forall\ell} \subseteq \mathsf{P/poly}$ are based on polynomial step normalization (Proposition 2) on top of continuity, as sketched above (Sect. 5). Finally, the equality $\mathsf{APN} = \mathsf{P/poly}$ was proved in [19].

Note that continuous approximation allows us to generate a family of proof nets from a single parsimonious term. It is thus continuity that reconciles the two approaches to non-uniformity (family and individual). From a more practical perspective, generation of proof nets from a single program is reminiscent of the work of Ghica [7], who exhibits a way to synthesize VLSI circuits from a functional program.

Our parsimonious framework has two aspects. On the logical side, parsimonious logic amounts to multiplicative affine logic with what we call *Milner's exponential modality*, enjoying monoidal functorial promotion and *Milner's law* $!A \cong A \otimes !A$. With respect to the usual exponential modality, Milner's exponential refuses *digging* $!A \multimap !!A$ and *contraction* $!A \multimap !A \otimes !A$. One side of Milner's law is an asymmetric form of contraction $!A \multimap A \otimes !A$, also known as *absorption*, whereas its dual $A \otimes !A \multimap !A$ has a differential flavor [4]. Indeed, parsimonious logic resembles an affine subsystem of differential linear logic, but we have not fully explored the connection.

On the computational side, the parsimonious λ-calculus may be seen as an affine λ-calculus with built-in streams. This is because Milner's law naturally makes $!A$ be the type of streams on A: absorption is "pop" and coabsorption is "push". Stream calculi abound in the literature, also in connection with (classical) logic [18] and ICC [6, 17]. However, these are all orthogonal to the present work, both in terms of motivations (modeling streams is not our primary concern) and technically (our streams arise from a previously unremarked restriction of the exponential modality of linear logic).

We should also mention the companion paper [14], currently submitted, which focuses on the uniform version of the simply-typed parsimonious calculus presented here, showing that it exactly captures L, *i.e.*, uniform logspace. This complements in a nice way our results but neither implies nor is implied by them, so the papers overlap but are technically different. Also, [14] does not consider linear polymorphism.

2 The Non-uniform Parsimonious Lambda-calculus

We introduce an alternative term syntax for the infinitary parsimonious λ-calculus and its associated type system, improving on previous work [13, 14] in two respects. First, we avoid use of *indices* when referring to instances of exponential variables (that are reminiscent of indices in AJM games [1]). We instead use more conventional tools like list and let constructs. Second, the calculus has a closer fit with the type system, *i.e.*, the type system is *syntax-directed*. This offers a better logical account of the programs, not to mention easier type inference.

In the following, $[k]$ stands for the set $\{0, \dots, k-1\}$ for every $k \in \mathbb{N}$.

Terms. We let $a, b, c, \dots$ (resp. $x, y, z, \dots$) range over a denumerably infinite set of *linear* (resp. *exponential*) variables. A *pattern* p is either $a \otimes b$ or a list $a_0 :: a_1 :: \cdots :: a_{n-1} :: x$ with $n \geq 0$ and $a_0, \dots, a_{n-1}$ distinct. If $n = 0$, the latter denotes the one element list x. We often use notation $\mathsf{p}(x)$ to indicate the last exponential variable x. The *terms* are inductively generated by

$$t, u ::= \bot \mid a \mid x \mid \lambda a.t \mid tu \mid t \otimes u \mid t :: u \mid \mathsf{let}\ \mathsf{p} = t\ \mathsf{in}\ u \mid !_f(u_0, \dots, u_{k-1}),$$

where $k \geq 1$ and $f : \mathbb{N} \longrightarrow [k]$. The expression $!_f(u_0, \dots, u_{k-1})$ is called a *box*. We use $\mathbf{u}$ to range over boxes. When $k = 1$, f is obvious and we write $!(u_0)$, or even $!u_0$. Restricting to boxes of this form yields a *uniform* calculus, which is the object of [14].

The box $\mathbf{u}$ generates an infinite stream $u_{f(0)} :: u_{f(1)} :: u_{f(2)} :: \cdots$. We denote the n-th component of the stream by $\mathbf{u}(n)$, and the result of removing the first n elements by $\mathbf{u}^{+n}$, so that the stream can be expressed as $\mathbf{u}(0) :: \mathbf{u}(1) :: \cdots :: \mathbf{u}(n-1) :: \mathbf{u}^{+n}$. More precisely, $\mathbf{u}^{+n}$ denotes $!_{f^{+n}}(u_0, \ldots, u_{k-1})$, where $f^{+n}(i) := n + i$.

Binders behave as expected; $\lambda a.u$ and ($\mathsf{let}\ a \otimes b = t\ \mathsf{in}\ u$) bind linear variable a (and also b in the latter case) occurring in u, while ($\mathsf{let}\ \mathsf{p} = t\ \mathsf{in}\ u$) with $\mathsf{p} = a_0 :: \cdots :: a_{n-1} :: x$ binds both linear variables $a_0, \ldots, a_{n-1}$ and an exponential variable x occurring in u. We adopt the standard α-equivalence and Barendregt's variable convention. Constant $\perp$ corresponds to *coweakening* in logic. It is only used for auxiliary purposes in this paper.

Informally, $t :: \mathbf{u}$ expresses the result of pushing an element t to the stream $\mathbf{u}$. It is convenient to think of pattern $a_0 :: \cdots :: a_{n-1} :: x$ as a "non-uniform" variable. When "substituting" a box $\mathbf{u}$ for it, the first n variables $a_0, \ldots a_{n-1}$ are replaced by the first n components $\mathbf{u}(0), \ldots, \mathbf{u}(n-1)$ (with free variables renamed), while x takes $\mathbf{u}^{+n}$.

A *slice* of a term is obtained by removing all components but one from each box. A term is *parsimonious* if (i) all its slices are affine, *i.e.*, each variable (linear or not) occurs at most once; (ii) box subterms do not contain free linear variables; and (iii) all exponential variables belong to a box subterm. Thus each exponential variable corresponds to an "auxiliary door" of a unique box. The set of parsimonious terms in the above sense is denoted by $\mathrm{nuP}\Lambda$.

Reduction. One-step reduction $t \xrightarrow{\sigma} u$ is defined relatively to a finite set σ of the form $\{b_1 :: x_1, \ldots, b_k :: x_k\}$. We denote the set $\{b_1, \ldots, b_k, x_1, \ldots, x_k\}$ by $\mathrm{fv}(\sigma)$. There are seven elementary rules, among which (beta), (com1) and (com2) are standard:

$$\begin{array}{llll}
(\mathsf{beta}) & (\lambda a.t)u \xrightarrow{\emptyset} t[u/a] & (\mathsf{merge}) & \mathsf{let}\ x = \mathbf{u}\ \mathsf{in}\ w \xrightarrow{\emptyset} w\{\mathbf{u}/x\} \\
(\mathsf{com2}) & \mathsf{let}\ \mathsf{p} = C[t]\ \mathsf{in}\ u \xrightarrow{\emptyset} C[\mathsf{let}\ \mathsf{p} = t\ \mathsf{in}\ u] & (\mathsf{com1}) & C[t]u \xrightarrow{\emptyset} C[tu]
\end{array}$$

$$\begin{array}{ll}
(\mathsf{cons}) & \mathsf{let}\ a :: \mathsf{p} = t :: v\ \mathsf{in}\ w \xrightarrow{\emptyset} \mathsf{let}\ \mathsf{p} = v\ \mathsf{in}\ w[t/a] \\
(\mathsf{dup}) & \mathsf{let}\ a :: \mathsf{p} = \mathbf{u}\ \mathsf{in}\ w \xrightarrow{\sigma} \mathsf{let}\ \mathsf{p} = \mathbf{u}^{+1}\ \mathsf{in}\ w[\mathbf{u}(0)'/a] \\
(\mathsf{aux}) & \mathsf{let}\ x = t :: v\ \mathsf{in}\ w[\mathbf{u}] \xrightarrow{\sigma} \mathsf{let}\ x = v\ \mathsf{in}\ w[\mathbf{u}(0)'[t/x] :: \mathbf{u}^{+1}]
\end{array}$$

where $C[\bullet]$ stands for a context of the form ($\mathsf{let}\ \mathsf{q} = v\ \mathsf{in}\ \bullet$). The term $\mathbf{u}(0)'$ is obtained from $\mathbf{u}(0)$ by replacing its free exponential variables $x_1, \ldots, x_m$ (except x in (aux)) with fresh linear variables $b_1, \ldots, b_m$. Thus the (dup) and (aux) rules introduce new free variables, which are recorded in $\sigma := \{b_1 :: x_1, \ldots, b_m :: x_m\}$ and are bound later. In the (aux) rule, the notation $w[\mathbf{u}]$ means that w contains a box $\mathbf{u}$ s.t. $x \in \mathrm{fv}(\mathbf{u})$. If no such box exists, the term $t :: v$ is erased, *i.e.*, the right hand side is w. This rule corresponds to a cut between a cocontraction and the auxiliary port of a box in differential linear logic. We include it for completeness but we never need it.

The substitution $w\{\mathbf{u}/x\}$ in the (merge) rule needs an explanation. Suppose that $\mathbf{u} = !_f(u_0, \ldots, u_{k-1})$ and the variable x occurs in a box $\mathbf{w} = !_g(w_0, \ldots, w_{l-1})$. Our intention is to replace the stream $\ldots \mathbf{w}(i) \ldots$ with $\ldots \mathbf{w}(i)[\mathbf{u}(i)/x] \ldots$ To achieve this, let $v_{ik+j} := w_i[u_j/x]$ for each $i \in [l]$ and $j \in [k]$. Then $w\{\mathbf{u}/x\}$ is the result of replacing the box $\mathbf{w}$ with $!_h(v_0, \ldots, v_{lk-1})$, where $h(m) = g(m)k + f(m)$.

The above rules are extended contextually. We have:

$$\frac{t \xrightarrow{\sigma} u}{C[t] \xrightarrow{\sigma} C[u]} \qquad \frac{w \xrightarrow{\sigma \cup \{b::x\}} w'}{\mathsf{let}\ \mathsf{p}(x) = v\ \mathsf{in}\ w \xrightarrow{\sigma} \mathsf{let}\ \mathsf{p}(b :: x) = v\ \mathsf{in}\ w'}$$

where the left rule applies when $C[t]$ or $C[u]$ does not bind any variable in $\mathrm{fv}(\sigma)$.

$$\dfrac{}{\Gamma; \Delta, a : A \vdash a : A}\ \mathsf{ax} \qquad\qquad \dfrac{}{\Gamma; \Delta \vdash \bot : A}\ \mathsf{coweak}$$

$$\dfrac{\Gamma; \Delta, a : A \vdash t : B}{\Gamma; \Delta \vdash \lambda a.t : A \multimap B}\ {\multimap}\mathsf{I} \qquad\qquad \dfrac{\Gamma_1; \Delta_1 \vdash t : A \multimap B \quad \Gamma_2; \Delta_2 \vdash u : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash tu : B}\ {\multimap}\mathsf{E}$$

$$\dfrac{\Gamma_1; \Delta_1 \vdash t : A \quad \Gamma_2; \Delta_2 \vdash u : B}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t \otimes u : A \otimes B}\ {\otimes}\mathsf{I} \qquad \dfrac{\Gamma_1; \Delta_1 \vdash t : A \otimes B \quad \Gamma_2; \Delta_2, a : A, b : B \vdash u : C}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash \mathsf{let}\ a \otimes b = t\ \mathsf{in}\ u : C}\ {\otimes}\mathsf{E}$$

$$\dfrac{\Gamma, \mathsf{p} : A; \Delta, a : A \vdash t : C}{\Gamma, (a :: \mathsf{p}) : A; \Delta \vdash t : C}\ \mathsf{abs} \qquad\qquad \dfrac{\Gamma_1; \Delta_1 \vdash t : A \qquad\qquad \Gamma_2; \Delta_2 \vdash u : !A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash (t :: u) : !A}\ \mathsf{coabs}$$

$$\dfrac{; \Delta \vdash u_0 : A \quad \cdots \quad ; \Delta \vdash u_{k-1} : A}{\Delta'; \vdash !_f(u'_0, \dots, u'_{k-1}) : !A}\ !\mathsf{I} \qquad\qquad \dfrac{\Gamma_1; \Delta_1 \vdash t : !A \quad \Gamma_2, \mathsf{p} : A; \Delta_2 \vdash u : C}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash \mathsf{let}\ \mathsf{p} = t\ \mathsf{in}\ u : C}\ !\mathsf{E}$$

$$\dfrac{\Gamma; \Delta \vdash t : A \quad \alpha \notin \mathrm{fv}(\Gamma, \Delta)}{\Gamma; \Delta \vdash t : \forall \alpha.A}\ \forall\mathsf{I} \qquad\qquad \dfrac{\Gamma; \Delta \vdash t : \forall \alpha.A \quad B \text{ is } !\text{-free}}{\Gamma; \Delta \vdash t : A[B/\alpha]}\ \forall\mathsf{E}$$

Fig. 1. The typing system $\mathbf{nuPL}_{\forall\ell}$. Removing the last two rules yields $\mathbf{nuPL}$.

Finally we write $t \to u$ if $t \xrightarrow{\emptyset} u$ holds. This is our notion of one-step reduction. *Reduction* is the reflexive-transitive closure of $\to$, denoted by $\to^*$.

For instance, if $t := \mathsf{let}\ a :: \mathsf{p} = !u(x)\ \mathsf{in}\ a \otimes v$ and $t' := \mathsf{let}\ \mathsf{p} = !u(x)\ \mathsf{in}\ u(b) \otimes v$, we have $t \xrightarrow{\{b::x\}} t'$, so $\mathsf{let}\ x = w\ \mathsf{in}\ t \to \mathsf{let}\ b :: x = w\ \mathsf{in}\ t'$. Thus the "uniform" variable x is replaced by the "non-uniform" $b :: x$.

Reduction may be shown to be confluent. However, termination is not guaranteed: take $\Delta := \lambda b.\mathsf{let}\ a :: x = b\ \mathsf{in}\ a\,!x$ and let $\Omega := \Delta\,!\Delta$. These terms are parsimonious, and we have $\Omega \to \Omega$, like the namesake usual λ-term.

Types. We take as types the formulas of intuitionistic second order linear logic:

$$A, B ::= \alpha \mid A \multimap B \mid A \otimes B \mid !A \mid \forall \alpha.A$$

where α is a type variable. The set of types (resp. $\forall$-free types) is denoted by $\mathsf{Type}_{\forall\ell}$ (resp. Type).

We adopt a type system with dual contexts as in [2]. Typing judgments are of the form $\Gamma; \Delta \vdash t : A$, where Γ is a set of assignments of the form $\mathsf{p}(x) : C$, while Δ consists of assignments of the form $a : C$. Moreover, all variables occurring in Γ, Δ are distinct.

A term t is *typable* in $\mathbf{nuPL}_{\forall\ell}$ if the judgment $\Gamma; \Delta \vdash t : A$ may be derived according to the rules of Fig. 1. Likewise, t is *typable* in $\mathbf{nuPL}$ if $\Gamma; \Delta \vdash t : A$ is derived without any use of the quantifier $\forall$.

In the rule $!\mathsf{I}$, if $\Delta = \{b_1 : B_1, \dots, b_m : B_m\}$, then $\Delta' := \{y_1 : B_1, \dots, y_m : B_m\}$ with $y_1, \dots, y_m$ fresh, and $u'_i := u_i[y_1/b_1, \dots, y_m/b_m]$. Notice that the rule $\forall\mathsf{E}$ is applicable *only when B is $!$-free*. An induction on the last rule of derivations gives

Lemma 1. *Suppose that* $; \Delta \vdash t : A$. *Then:*

1. **parsimony:** $t \in \mathrm{nuP}\Lambda$*;*
2. **typical ambiguity:** $; \Delta[B/\alpha] \vdash t : A[B/\alpha]$ *for any type B;*
3. **subject reduction:** $t \to t'$ *implies* $; \Delta \vdash t' : A$.

In the sequel, we will mainly deal with simple types in Type. Polymorphic types in $\mathsf{Type}_{\forall\ell}$ are considered only when necessary. When working with Type, it is convenient to fix a propositional variable, which we denote by o. If $A, B \in \mathsf{Type}$, $A[B]$ stands for $A[B/o]$. We will even omit B, just writing $A[]$. This lack of information is harmless for composition: point 2 of Lemma 1 guarantees that terms of type $A[X] \multimap B$ and $B[Y] \multimap C$ may be composed to yield $A[X[Y]] \multimap C$. The only delicate point is iteration (see below), which requires *flat* terms, *i.e.*, of type $A \multimap A$.

Examples. We set $\lambda \mathsf{p}.t := \lambda c.\mathsf{let}\ \mathsf{p} = c\ \mathsf{in}\ t$ with c a fresh variable, so that we have

$$(\lambda a \otimes b.t(a,b))\, u \otimes v \to^* t(u,v), \qquad (\lambda a :: x.t(a,x))\, u :: v \to^* t(u,v).$$

With this notation, the two terms implementing Milner's law may be written as

$$\lambda a :: x.a \otimes !x : !A \multimap A \otimes !A, \qquad \lambda a \otimes b.\mathsf{let}\ x = b\ \mathsf{in}\ a :: !x : A \otimes !A \multimap !A.$$

Fig. 2 shows some examples of data and functions represented in $\mathbf{nuPL}$. As usual, a natural number n is expressed by a Church numeral n of type Nat. The type Nat supports iteration $\mathsf{It}(n, step', base) := n\,!(step')\,base$, typed as:

$$\frac{;\Delta \vdash step : A \multimap A \quad \Gamma; \Sigma \vdash base : A}{\Delta', \Gamma; \Sigma, n : \mathsf{Nat}[A] \vdash \mathsf{It}(n, step', base) : A}$$

where Δ' and $step'$ are the results of systematically replacing linear variables by exponential ones. Note that the type of $step$ must be flat.

Since succ has a flat type $\mathsf{Nat} \multimap \mathsf{Nat}$, it may be iterated to result in the addition function of type $\mathsf{Nat}[] \multimap \mathsf{Nat} \multimap \mathsf{Nat}$. It is again flat with respect to the second argument, so a further iteration leads to the multiplication function of type $\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}$. Subtraction is defined similarly.

Notably, Church numerals are duplicable and storable as shown in Fig. 2. Using addition, multiplication, subtraction and duplication we may represent any polynomial with integer coefficients as a closed term of type $\mathsf{Nat}[] \multimap \mathsf{Nat}$.

Moreover, all these unary constructions can be extended to binary ones. We define $\mathsf{Str} := !(o \multimap o) \multimap !(o \multimap o) \multimap o \multimap o$. The Church representation for the binary string $w = b_0 \cdots b_{n-1} \in \{0,1\}^n$ is given by

$$\mathsf{w} := \lambda s_0 :: \cdots :: s_{n-1} :: x.\lambda s'_0 :: \cdots :: s'_{n-1} :: y.\lambda d.c_0(\cdots c_{n-1} d \cdots) : \mathsf{Str},$$

where $c_i = s_i$ or s'_i depending on the bit b_i being 0 or 1.

For the Boolean values, we adopt the multiplicative type Bool used in [19]. Just as numerals, words and Booleans are duplicable and storable.

An advantage of multiplicative Booleans is that they support *flat* exclusive-or $\oplus$ in addition to flat negation $\neg$. On the other hand, conjunction and disjunction cannot be flatly represented multiplicatively. The best we have is $\wedge : \mathsf{Bool}[] \multimap \mathsf{Bool} \multimap \mathsf{Bool}$, i.e., one of the arguments must be non-flat.

In contrast, $\wedge$ admits a flat typing in $\mathbf{nuPL}_{\forall\ell}$ by re-defining $\mathsf{Bool} := \forall\alpha.\alpha \otimes \alpha \multimap \alpha \otimes \alpha$. As we will see in the next section, this results in greater expressiveness. *This is the main difference between two type systems* $\mathbf{nuPL}$ *and* $\mathbf{nuPL}_{\forall\ell}$.

Let $t : \mathsf{Str}[] \multimap \mathsf{Bool}$ be a closed term typable in $\mathbf{nuPL}$ (resp. in $\mathbf{nuPL}_{\forall\ell}$). It defines a language $L(t) := \{w \in \{0,1\}^* : t\mathsf{w} \to^* \mathsf{tt}\}$. Let nuPL (resp. $\mathsf{nuPL}_{\forall\ell}$) be the class of all such languages.

Nat	$:= !(o \multimap o) \multimap o \multimap o, \qquad \mathsf{Bool} := o \otimes o \multimap o \otimes o$	
n	$:= \lambda s_0 :: \cdots :: s_{n-1} :: x.\lambda d.s_0(\ldots s_{n-1}d \ldots)$	$: \mathsf{Nat}$
succ	$:= \lambda n.\lambda s :: x.\lambda d.s(n!(x)d)$	$: \mathsf{Nat} \multimap \mathsf{Nat}$
pred	$:= \lambda n.\lambda x.\lambda d.n((\lambda a.a) :: !x)d$	$: \mathsf{Nat} \multimap \mathsf{Nat}$
dup	$:= \lambda n.\mathsf{It}(n, \lambda m_1 \otimes m_2.(\mathsf{succ}\ m_1) \otimes (\mathsf{succ}\ m_2), \mathsf{0} \otimes \mathsf{0})$	$: \mathsf{Nat}[] \multimap \mathsf{Nat} \otimes \mathsf{Nat}$
store	$:= \lambda n.\mathsf{It}(n, \lambda x.!(\mathsf{succ}\ x), !\mathsf{0})$	$: \mathsf{Nat}[] \multimap !\mathsf{Nat}$
tt	$:= \lambda c \otimes d.c \otimes d, \qquad \mathsf{ff} := \lambda c \otimes d.d \otimes c$	$: \mathsf{Bool}$
$\neg$	$:= \lambda b.\lambda c \otimes d.b(d \otimes c)$	$: \mathsf{Bool} \multimap \mathsf{Bool}$
$\oplus$	$:= \lambda b_1.\lambda b_2.\lambda c.b_1(b_2 c)$	$: \mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool}$
$\wedge$	$:= \lambda b_1.\lambda b_2.\mathsf{let}\ c \otimes d = b_1(b_2 \otimes \mathsf{ff})\ \mathsf{in}\ c$	$: \mathsf{Bool}[] \multimap \mathsf{Bool} \multimap \mathsf{Bool}$

Fig. 2. Some data types and encodings

3 Expressiveness of Non-uniform Parsimonious Terms

In this section, we prove

Theorem 2. $\mathsf{L/poly} \subseteq \mathsf{nuPL}$ *and* $\mathsf{P/poly} \subseteq \mathsf{nuPL}_{\forall\ell}$.

An *n-input branching program* is a triple $P = (G^P, v_s^P, v_t^P)$ where: G^P is a finite directed acyclic graph (dag); its nodes have out-degree 2 or 0 and are labelled in $[n]$; each node of out-degree 2 has one outgoing edge labelled by 0 and one by 1; and v_s^P, v_t^P are the *source* and *target* node, the former having in-degree 0, and the latter having out-degree 0. The *size* of P is the number of nodes of G^P.

A binary string $w = b_0 \cdots b_{n-1} \in \{0,1\}^n$ and an n-input branching program P induce a directed forest $P(w)$, as follows: take G^P and, for each node v of out-degree 2 whose label is i, erase the edge labelled by $1 - b_i$. Thus, $P(w)$ is a dag of out-degree at most 1, *i.e.*, a forest, whose edges are directed towards the roots. We say that P accepts w if v_s^P is a leaf of the tree whose root is v_t^P; otherwise, it rejects.

A *family* of branching programs is a sequence $(P_n)_{n \in \mathbb{N}}$ s.t., for all $n \in \mathbb{N}$, P_n is an n-input branching program. It is of *polynomial size* if there exists a polynomial p such that, for all $n \in \mathbb{N}$, the size of P_n is bounded by $p(n)$. It is well known that $\mathsf{L/poly}$ is exactly the class of languages decided by families of branching programs of polynomial size. Therefore, to prove the first part of Theorem 2 it will be enough to encode polysize families of branching programs, as sketched below.

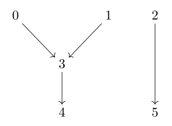


Fig. 3. A forest.

Encoding a forest. We have a very simple encoding of a forest G thanks to a flat encoding of the exclusive-or function. Suppose that the nodes of G are $\{0, \ldots, m-1\}$ and think of a token traversing G. We express the state "the token is placed at node i" by the term $@\mathsf{i} := !_{\delta_i}(\mathsf{ff}, \mathsf{tt})$, where $\delta_i(j) = 1$ iff $i = j$. The term $@\mathsf{i}$ is of type $!\mathsf{Bool}$ and expresses the stream in which only the ith component is tt and the rest is ff.

A forest is expressed by a term of type $!\mathsf{Bool} \multimap !\mathsf{Bool}$. For instance the forest in Fig. 3 is expressed by the term t below:

$$\lambda(c_0 :: c_1 :: c_2 :: c_3 :: c_4 :: c_5 :: x).\ (\mathsf{ff} :: \mathsf{ff} :: \mathsf{ff} :: c_0 \oplus c_1 :: c_3 \oplus c_4 :: c_2 \oplus c_5 :: !x).$$

This term replaces the 0th, 1st and 2nd components of a given stream with ff since there are no edges coming into the nodes $0, 1, 2$. The 3rd is the exclusive-or of the 0th and 1st, while the 4th (resp. 5th) is the exclusive-or of the 3rd and 4th (resp. 2nd and 5th). Thus the term actually represents a graph where self-loops are added to terminal nodes 4 and 5. As a consequence, we have $\mathsf{It}(\mathsf{k}, t, @\mathsf{i}) \to^* @\mathsf{j}$ iff terminal node j is reachable from node i, as far as $k \geq 2$. It is clear that the same encoding works for arbitrary forests of arbitrary size.

Encoding a branching program. Actually the edges of a forest are to be chosen according to the input binary string $w = b_0 \cdots b_{n-1}$ ($b_i \in \{0,1\}$) fed to a branching program P_n. Hence for instance the component $c_0 \oplus c_1$ above should be replaced by a term like $(b_i \wedge c_0) \oplus (\neg b_j \wedge c_1)$ where b_i, b_j are the Boolean values of type $\mathsf{Bool}[]$ depending on which the edges $0 \to 2$ and $1 \to 2$ are drawn; the former is drawn when $b_i = 1$, while the latter is drawn when $b_j = 0$. This raises no typing problem since conjunction $\wedge : \mathsf{Bool}[] \multimap \mathsf{Bool} \multimap \mathsf{Bool}$ is flat with respect to the second argument. We therefore obtain a term $t_n : !\mathsf{Bool}[] \multimap !\mathsf{Bool} \multimap !\mathsf{Bool}$ expressing a branching program $P_n = (G_n, 0, 1)$ (assuming that the source and target are respectively node 0 and 1). Converting an input string of type Str into a stream of type $!\mathsf{Bool}[]$ is easy.

Encoding a family of branching programs. This is not the end of the story. We cannot store the whole family $(P_n)_{n\in\mathbb{N}}$ in a term as it is, in spite of the infinite facility of our calculus. What we can store in a box $!_f(u_0, \dots, u_{n-1})$ is only an infinitary repetition of finitely many items. We are thus led to consider an *advice*, a stream of instructions, according to which each P_n is "woven" step by step.

For instance, to compute the Boolean value associated to a single node k (rather than the whole state of type $!\mathsf{Bool}$) from the previous state, it is sufficient to consider four instructions: `iskip`, `nskip`, `pos`, `neg`. Let $\mathsf{Adv} := o^4 \multimap o$ and represent each instruction by $\lambda a_0 a_1 a_2 a_3 . a_i$ with $i \in [4]$. Then any advice can be represented by a box $!_f(\mathtt{iskip}, \mathtt{nskip}, \mathtt{pos}, \mathtt{neg}) : !\mathsf{Adv}[]$ with a suitable f.

We consider the following term which expects four inputs: advice (encoded by a term of type $!\mathsf{Adv}[]$), input string ($!\mathsf{Bool}[]$), previous state ($!\mathsf{Bool}$) and temporal value of node k (Bool), and returns the updated values of the same type.

$$\begin{aligned}\lambda a{::}x.\, \lambda b{::}y.\, \lambda c{::}z.\, \lambda d.\ \mathsf{case}\ a\ \mathsf{of}\ & \mathtt{iskip} \to !(x) \otimes !(y) \otimes c{::}!(z) \otimes d;\\ & \mathtt{nskip} \to !(x) \otimes b{::}!(y) \otimes !(z) \otimes d;\\ & \mathtt{pos} \quad\ \to !(x) \otimes !(y) \otimes !(z) \otimes ((b \wedge c) \oplus d);\\ & \mathtt{neg} \quad\ \to !(x) \otimes !(y) \otimes !(z) \otimes ((\neg b \wedge c) \oplus d).\end{aligned}$$

The instruction `iskip` (resp. `nskip`) skips the first bit b of the input (resp. c of the previous state). The behaviors of `pos` and `neg` are as expected. For instance, starting from initial input string $b_0 :: \cdots :: b_{n-1}$, previous state $c_0 :: \cdots :: c_{m-1}$ and temporal value ff, iteration of the above term four times with advice $\mathtt{iskip} :: \mathtt{pos} :: \mathtt{nskip} :: \mathtt{neg}$ yields value $(b_1 \wedge c_0) \oplus (\neg b_2 \wedge c_2)$ (together with the rest of input $b_3 :: \cdots :: b_{n-1}$ and the rest of nodes $c_3 :: \cdots :: c_{m-1}$).

Actually things are more complicated, since unused values c_i in the previous state are not just thrown away but to be preserved for later use. Each bit b_j of the input string is to be used several times. Most importantly, we need to compute not just one value d (of type Bool) but a stream expressing the next state (of type $!\mathsf{Bool}$). We can manage to do that by using more complicated instructions and types.

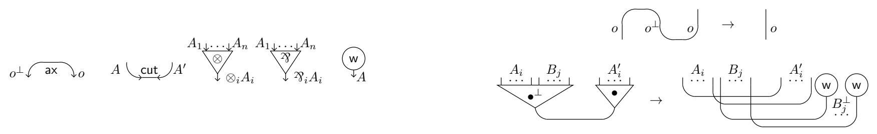


Fig. 4. Links (left) and cut-elimination steps (right). In the cut link, $A \perp A'$; $\bullet \in \{\otimes, \parr\}$.

In the end, we obtain a term t_P : $!\mathsf{Adv}[] \multimap !\mathsf{Bool}[] \multimap !\mathsf{Bool} \multimap !\mathsf{Bool}$ that "weaves" a forest $P_n(w)$ when an advice $\mathbf{a}_n$ of polynomial length $r(n)$ and an input string $\mathbf{w}$ of length n are provided. Actually the advice is given as a concatenation of all $(\mathbf{a}_i)_{i\in\mathbb{N}}$, from which a suitable advice for P_n is extracted by skipping the first $\sum_{i=1}^{n-1} r(i)$ components.

So far we gave a proof sketch of the first part of Theorem 2. For the second part, recall that $\mathbf{nuPL}_{\forall\ell}$ allows us to encode not only exclusive-or, but also conjunction and disjunction by terms of *flat* type $\mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool}$. Hence we may build a term similar to t_P above, which is able to "weave" a family of polynomial size Boolean circuits. This observation immediately leads to the second part of Theorem 2.

4 Boolean Nets and Logarithmic Space

We consider here classical propositional multiplicative linear formulas, generated by

$$A, B ::= o \mid o^\perp \mid \otimes_{i\le n} A_i \mid \parr_{i\le n} A_i,$$

where $n \in \mathbb{N}$ (the bound n will be omitted in the sequel unless necessary). Linear negation $(\cdot)^\perp$ is defined as usual via De Morgan laws (exchanging $\otimes$ and $\parr$). The *height* of a formula is its height as a tree. We will also need a more liberal notion of duality, which we denote by $A \perp B$, defined to be the smallest symmetric relation on formulas such that: $o \perp o^\perp$; if $A_1 \perp B_1, \dots, A_n \perp B_n$, then $\otimes_{i\le n} A_i \perp \parr_{i\le n+k} B_i$ and $\parr_{i\le n} A_i \perp \otimes_{i\le n+k} B_i$, with $B_{n+1}, \dots, B_{n+k}$ arbitrary. Of course $A \perp A^\perp$. Also, it is easy to check that, if $A \perp A'$, then A, A' have equal height.

A *net* is formula-labelled directed graph built by composing the nodes of Fig. 4 (left), called *links*. Composition must respect the orientation, and the labeling must respect the constraints given in Fig. 4. Edges are allowed to have unconnected extremities. The edges incoming in (resp. outgoing from) a link are called *premises* (resp. *conclusions*) of the link. The number of premises of a $\otimes$ or $\parr$ link is called its *arity*. The premises of $\otimes$ and $\parr$ links are ordered, so we may speak of "the i-th premise". Each edge must be the conclusion of a link. The *size* of a net is the number of its links. Cut-elimination steps are defined in Fig. 4 (right).

Definition 1 (Boolean net). *An n-input* Boolean net *π is a net whose conclusions have type* $\mathsf{Bool}[A_1]^\perp, \dots, \mathsf{Bool}[A_n]^\perp, \mathsf{Bool}$. *The* height *of π is the maximum height of A_i. A* family *of Boolean nets is sequence $(\pi_n)_{n\in\mathbb{N}}$ where each π_n is an n-input Boolean net. A family $(\pi_n)_{n\in\mathbb{N}}$ accepts (resp. rejects) a string $w \in \mathbb{W}$ if the net obtained by cutting $\pi_{|w|}$ with $\mathsf{w}_0, \dots, \mathsf{w}_{|\mathsf{w}|-1}$ normalizes to* tt *(resp.* ff*). We denote by* APN^0 *the class of languages decided by families of Boolean nets of polynomial size and bounded height.*

Observe that our nets differ from those of [19] because of the presence of free weakening and because of the more liberal notion of duality. However, these features are harmless: a Boolean proof net in the sense of [19] (with garbage disposed) is a Boolean net in our sense; for the converse, our nets may be simulated by the nets of [19] by adjusting the arity of links and introducing garbage (represented by weakening in our context). Therefore, the class of problems decidable by our families of Boolean nets of polynomial size and bounded height is exactly what [19] calls APN^0.

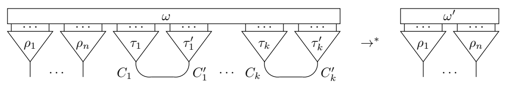


Fig. 5. A generic net π and its normal form π'.

A net π with n conclusions and k cut links has the shape given in the left hand side of Fig. 5, where $\rho_1, \dots, \rho_n, \tau_1, \tau_1', \dots, \tau_k, \tau_k'$ are trees of $\otimes$, $⅋$ and w links, and ω consists solely of ax links. Moreover, since the leaves of ρ_i are labelled by atomic formulas, and since only ax and w links may have atomic conclusions, the normal form π' of π (which exists and is unique) has the shape given in the right hand side of Fig. 5, where the conclusions of ω' are conclusions of ax or w links (ω' may contain irreducible cut links but this will not be important for us).

We will be interested in detecting *companion leaves* of ρ_i, ρ_j in π' (we allow $i = j$), *i.e.*, leaves which are conclusions of the same ax link of ω'. The geometry of interaction (GoI, [8]) gives us a way of doing this directly in π, without applying any cut-elimination step. The following is an immediate application of standard GoI definitions. See Appendix A for the details.

Proposition 1. *Let π be as in Fig. 5, of size s, and let h be the maximum height of $C_1, \dots, C_k$. If e, e' are two leaves of ρ_i, ρ_j, deciding whether e, e' are companions in the normal form of π may be done in space $O(h \log s)$.*

Theorem 3. $\mathsf{APN}^0 \subseteq \mathsf{L/poly}$.

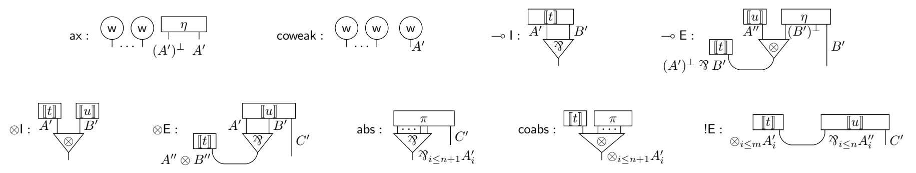
Proof. Let $(\pi_n)_{n\in\mathbb{N}}$ be such a family, of size $p(n)$. Since p is a polynomial, the whole family may be encoded as advice. Deciding whether $w \in \{0,1\}^n$ is accepted amounts to detecting whether a certain pair of leaves in π_n cut with the representations of the bits of w are companions in the normal form, which by Proposition 1 may be done in space $O(h \log p(n)) = O(h \log n)$. But, by definition, the height h of cut formulas does not depend on n, so we conclude. □

5 Approximations and Boolean Nets

We say that a term is *finite* if it contains only boxes of the form $!\bot$. Finite terms may be mapped to nets in a standard way, as follows. We first introduce the relation $A' \sqsubseteq A$ between classical multiplicative formulas and simple types, as the smallest such that: $o \sqsubseteq o$; if $A' \sqsubseteq A$ and $B' \sqsubseteq B$, then $(A')^\perp ⅋ B' \sqsubseteq A \multimap B$ and $A' \otimes B' \sqsubseteq A \otimes B$; if $A_1', \dots, A_n' \sqsubseteq A$, then $\otimes_{i\leq n} A_i' \sqsubseteq !A$. A straightforward induction on A gives

Lemma 2. *$A', A'' \sqsubseteq A$ implies $(A')^\perp \perp A''$.*

Let t be finite. We will map a type derivation of $\Gamma; \Delta \vdash t : A$ in $\mathbf{nuPL}$ to a net, which we abusively denote by $[\![t]\!]$, of conclusions Γ', Δ', A' such that $(\Gamma')^\perp \sqsubseteq !\Gamma$, $(\Delta')^\perp \sqsubseteq \Delta$ and $A' \sqsubseteq A$. The definition is by induction on the last rule of the derivation:



In the ax and coweak case, we choose Γ', Δ' and A' so that every ! is approximated in the minimal way, *i.e.*, $n = 1$ in the definition of $\sqsubseteq$. In the cases ax and $\multimap$ E, η denotes the η-expansion net, defined as usual. In the cases $\multimap$ E, $\otimes$E and !E, Lemma 2 guarantees the soundness of the typing. For the rule !I, by finiteness the only possibility is that $k = 1$ and $u_0 = \bot$, so this case is treated as coweak. In the case abs (resp. coabs), the net π represents the net $[\![t]\!]$ (resp. $[\![u]\!]$) from which the $\parr$ (resp. $\otimes$) link corresponding to $\mathsf{p} : A$ (resp. $!A$) has been removed. If no such link exists, it means that the conclusion came from a w (it cannot come from an atomic axiom), in which case we simply add a binary $\parr$ (resp. $\otimes$) link.

Note that the size of $[\![t]\!]$ is $O(s \cdot |t|)$, where s is the size of the types in t (this is because of the η nets). The following is standard:

Lemma 3. *Let t be finite and typable in* $\mathbf{nuPL}$. *Then, $t \to t'$ implies $[\![t]\!] \to^* [\![t']\!]$.*

In what follows, for convenience we associate with each non-linear variable x a sequence $a_0, a_1, a_2, \dots$ of pairwise distinct linear variables. We further suppose that if a (resp. b) is associated with x (resp. y), then $x \neq y$ implies $a \neq b$. The n-th *approximation* of $t \in \mathrm{nuP}\Lambda$ is a finite term $\lfloor t \rfloor_n$ defined as follows: $\lfloor \bot \rfloor_n := \bot$; $\lfloor a \rfloor_n := a$; $\lfloor x \rfloor_n := \bot$; $\lfloor \lambda a.t \rfloor_n := \lambda a.\lfloor t \rfloor_n$; $\lfloor tu \rfloor_n := \lfloor t \rfloor_n \lfloor u \rfloor_n$; $\lfloor t \otimes u \rfloor_n := \lfloor t \rfloor_n \otimes \lfloor u \rfloor_n$; $\lfloor t :: u \rfloor_n := \lfloor t \rfloor_n :: \lfloor u \rfloor_n$; $\lfloor \mathsf{let}\ a \otimes b = u\ \mathsf{in}\ t \rfloor_n := \mathsf{let}\ a \otimes b = \lfloor u \rfloor_n\ \mathsf{in}\ \lfloor t \rfloor_n$; $\lfloor \mathsf{let}\ \mathsf{p}(x) = u\ \mathsf{in}\ t \rfloor_n := \mathsf{let}\ \mathsf{p}(a_0 :: \cdots :: a_{n-1} :: x) = \lfloor u \rfloor_n\ \mathsf{in}\ \lfloor t \rfloor_n$, where the a_i are associated with x; $\lfloor \mathbf{u} \rfloor_n := \lfloor \mathbf{u}(0)[\overline{a}_0/\overline{x}] \rfloor_n :: \cdots :: \lfloor \mathbf{u}(n-1)[\overline{a}_{n-1}/\overline{x}] \rfloor_n :: !\bot$, where $\overline{x}$ are the free variables of the box and $\overline{a}_i$ are associated with $\overline{x}$.

Proposition 2. *Let t : Bool in $\mathbf{nuPL}_{\forall\ell}$ (or $\mathbf{nuPL}$). There exists a polynomial p depending solely on the types appearing in t and on its depth (the maximum number of nested boxes) such that, if $t \to^l$ b with b $\in \{\mathsf{tt}, \mathsf{ff}\}$, then $l \leq p(|t|)$. As a consequence, there is a polynomial q (with the same dependencies) s.t. $\lfloor t \rfloor_{q(|t|)} \to^*$ b.*

Proof. The bound on the reduction length is proved by a careful reformulation of a standard cut-elimination argument, see Appendix B. This immediately induces the approximation bound via continuity, as proved in [13] (Lemmas 3 and 4). □

Theorem 4. nuPL $\subseteq$ APN^0 *and* nuPL$_{\forall\ell}$ $\subseteq$ P/poly.

Proof. Given t : Str$[A] \multimap$ Bool and $n \in \mathbb{N}$, it is easy to obtain a term t' whose type is $; b_0$: Bool$[A \multimap A], \dots, b_{n-1}[A \multimap A]$: Bool $\vdash t'(b_0, \dots, b_{n-1})$: Bool. It takes n Booleans, converts them into a string of type Str$[A]$, and then passes it to t. If $w = b_0 \cdots b_{n-1} \in \{0,1\}^n$, we have $u := t'(\mathsf{b}_0, \dots, \mathsf{b}_{n-1}) \to^* tw \to^*$ b. By Proposition 2, we have $\lfloor u \rfloor_{p(|u|)} \to^*$ b for p a polynomial not depending on w. But $|u| = O(n)$ by construction. Hence, by Lemma 3 the language decided by t may be decided by the family of Boolean nets $\pi_n := [\![\lfloor t' \rfloor_{q(n)}]\!]$ with q a polynomial, so the family is of polynomial size. Moreover, the conclusions of π_n are Bool$[B_n]^\bot, \dots,$ Bool$[B_n]^\bot$, Bool, with

$B_n \sqsubseteq A \multimap A$, whose height does not depend on n (only the arity). Hence the family is also of bounded height. The second part is an immediate consequence of Proposition 2 as delineated in the introduction: the height of formulas plays no role, we normalize the underlying untyped term. $\square$

Acknowledgments. This work was partially supported by ANR projects LOGOI ANR-2010-BLAN-0213-02, COQUAS ANR-12-JS02-006-01 and ELICA ANR-14-CE25-0005.

References

1. Abramsky, S., Jagadeesan, R., Malacaria, P.: Full abstraction for PCF. Inf. Comput. 163(2), 409–470 (2000)
2. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. Inf. Comput. 207(1), 41–62 (2009)
3. Bellantoni, S., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity 2, 97–110 (1992)
4. Ehrhard, T., Regnier, L.: Differential interaction nets. Electr. Notes Theor. Comput. Sci. 123, 35–74 (2005)
5. Ehrhard, T., Regnier, L.: Uniformity and the taylor expansion of ordinary lambda-terms. Theor. Comput. Sci. 403(2-3), 347–372 (2008)
6. Gaboardi, M., Péchoux, R.: Upper bounds on stream I/O using semantic interpretations. In: Proceedings of CSL. pp. 271–286 (2009)
7. Ghica, D.R.: Geometry of synthesis: a structured approach to VLSI design. In: Proceedings of POPL. pp. 363–375 (2007)
8. Girard, J.Y.: Geometry of interaction I: Interpretation of system F. In: Proccedings of Logic Colloquium 1988. pp. 221–260 (1989)
9. Girard, J.Y.: Light linear logic. Inf. Comput. 143(2), 175–204 (1998)
10. Kfoury, A.J.: A linearization of the lambda-calculus and consequences. J. Log. Comput. 10(3), 411–436 (2000)
11. Leivant, D., Marion, J.Y.: Lambda calculus characterizations of poly-time. Fundam. Inform. 19(1/2) (1993)
12. Mazza, D.: An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In: Proceedings of LICS. pp. 471–480 (2012)
13. Mazza, D.: Non-uniform polytime computation in the infinitary affine lambda-calculus. In: Proceedings of ICALP, Part II. pp. 305–317 (2014)
14. Mazza, D.: Simple affine types and logspace (2015), submitted to LICS 2015, available on the author's web page.
15. Melliès, P.A.: Asynchronous games 2: The true concurrency of innocence. Theor. Comput. Sci. 358(2-3), 200–228 (2006)
16. Melliès, P., Tabareau, N., Tasson, C.: An explicit formula for the free exponential modality of linear logic. In: Proceedings of ICALP, Part II. pp. 247–260 (2009)
17. Ramyaa, R., Leivant, D.: Ramified corecurrence and logspace. Electr. Notes Theor. Comput. Sci. 276, 247–261 (2011)
18. Saurin, A.: Typing streams in the lambdaμ-calculus. ACM Trans. Comput. Log. 11(4) (2010)
19. Terui, K.: Proof nets and boolean circuits. In: Proceedings of LICS. pp. 182–191 (2004)

A The Geometry of Interaction (Proposition 1)

In the following definition, we use the wild card $\bullet \in \{\otimes, \parr\}$.

Definition 2 (Interaction Abstract Machine). *Given a net π, we define an automaton $\mathsf{IAM}(\pi)$ as follows. Its* states *are triples (d, e, S), where $d \in \{\uparrow, \downarrow\}$, e is an edge of π and S is a finite stack of natural numbers. Its transition relation $\rightsquigarrow_\pi$ is the smallest s.t.:*
ax*: $(\uparrow, e, S) \rightsquigarrow_\pi (\downarrow, e', S)$ if e, e' are the conclusions of an* ax *link;*
cut*: $(\downarrow, e, S) \rightsquigarrow_\pi (\uparrow, e', S)$ if e, e' are the premises of a* cut *link;*
- $\bullet$*: $(\downarrow, e, S) \rightsquigarrow_\pi (\downarrow, e', i \cdot S)$ and $(\uparrow, e', i \cdot S) \rightsquigarrow_\pi (\uparrow, e, S)$ if e is the i-th premise of a $\bullet$ link and e' its conclusion.*

Observe that the transitions are deterministic. A sequence of transitions $s \rightsquigarrow_\pi^* s'$ of $\mathsf{IAM}(\pi)$ is maximal *if the edges of s and s' are conclusions of π (not necessarily distinct). In that case, we write $s \rightsquigarrow_\pi^{max} s'$.*

Lemma 4 (Invariance [8]). *If $\pi \rightarrow^* \pi'$ as in Fig. 5, then $\rightsquigarrow_\pi^{max} = \rightsquigarrow_{\pi'}^{max}$. In particular, two leaves e, e' of ρ_i, ρ_j are companions in π' iff $(\uparrow, e, \epsilon) \rightsquigarrow_\pi^* (\downarrow, e', \epsilon)$.*

Proposition 1. *Let π be as in Fig. 5, of size s, and let h be the maximum height of $C_1, \ldots, C_k$. If e, e' are two leaves of ρ_i, ρ_j, deciding whether e, e' are companions in the normal form of π may be done in space $O((1+h)\log s)$.*

Proof. By Lemma 4, it is enough to simulate $\mathsf{IAM}(\pi)$. For this, we need one pointer to the edge of the current state, of size $\log s$, plus the space to store the stack. It is easy to see that the maximum length of the stack is reached when visiting the premise of a cut link c, in which case the length equals the height of the cut formula. Now, the integers (stored in binary) in a stack are bounded by the maximum arity of the links of π, which in turn is bounded by s, hence the thesis. □

B Polynomial Step Normalization (Proposition 2)

Throughout this section, we assume that every term t comes equipped with a typing derivation in $\mathbf{nuPL}_{\forall\ell}$ so that each subterm has a type in $\mathsf{Type}_{\forall\ell}$. Although a subterm may have several types, the rank defined below will be invariant due to the restriction of polymorphism to linear types. We omit the connective $\otimes$ just for simplicity.

For each term t, the *size* $|t|$ is the number of nodes in its generation tree. The *depth* $d(t)$ of t is the maximum number of nested boxes (as for proof nets). Note that the depth never increases by reduction, since parsimonious logic is digging-free. For each formula A, the *rank* $r(A)$ is defined as follows. $r(\alpha) = 0$, $r(A \multimap B) = \max\{r(A), r(B)\}$, $r(\forall\alpha.A) = r(A)$, and $r(!A) = r(A) + 1$. The *rank* $r(t)$ of a term t is the maximum rank of types occurring in the typing derivation for t.

Let $\Lambda_{d,r}$ be the set of closed terms of !-free type (*e.g.* Bool) which is of depth $\leq d$ and of rank $\leq r$. The goal of this section is to prove

Theorem 5. *For every $d, r \in \mathbb{N}$ there is a polynomial $p_{d,r}$ such that every term $t \in \Lambda_{d,r}$ normalizes in $p_{d,r}(|t|)$ steps.*

To prove this, we do not use the (aux) rule. The restriction to !-free types makes it possible.

We first introduce some concepts. A term is *l-normal* if the reduction rules other than (dup) and (aux) do not apply to it. Since no duplication is involved, reduction of a term t to its l-normal form takes at most $O(|t|^2)$ steps, which are negligible.

Given an l-normal term t, let $L = \{l_1, \dots, l_k\}$ be the set of subterms of the form $(\mathsf{let}\ \mathsf{p}(x) = u\ \mathsf{in}\ v)$ such that $u : A$ is of maximal rank (*i.e.* $r(A) = r(t)$). We remark that u is either a box or of the form $u_1 :: u_2$ because the rank is maximal and the type of t is !-free. L is partially ordered by: $l_i \prec l_j$ if l_j is of the form $(\mathsf{let}\ \mathsf{p}(x) = u\ \mathsf{in}\ v)$ and l_i is included in a box of u. It is clear that the length of any chain in L is bounded by $d(t)$. Each l_i involves a list $\mathsf{p}(x)$ of variables, whose length is denoted by m_i. The *pattern size* $|t|_p$ of t is defined to be the sum $m_1 + \cdots + m_k$.

Suppose that $t \in \Lambda_{d,r}$ is l-normal but not normal. We claim that there is always a redex $l_i \in L$ of the form $(\mathsf{let}\ a :: \mathsf{p} = \mathbf{u}\ \mathsf{in}\ v)$ such that (i) l_i is $\prec$-minimal and (ii) no free variable x of $\mathbf{u}$ is bound by another (dup)-*redex* in L (it is possible that x is bound by a (aux)-redex in L of the form $\mathsf{let}\ x = u_1 :: u_2\ \mathsf{in}\ C[l_i]$). Now let us observe what happens to $|t|$ and $|t|_p$ when such a redex l_i is fired:

$$l_i \quad \equiv \quad \mathsf{let}\ a :: \mathsf{p} = \mathbf{u}\ \mathsf{in}\ v \quad \xrightarrow{\sigma} \quad \mathsf{let}\ \mathsf{p} = \mathbf{u}^{+1}\ \mathsf{in}\ v[\mathbf{u}(0)'/a] \quad \equiv \quad l_i'.$$

The main effect is that it creates a copy $\mathbf{u}(0)'$. However, the term $\mathbf{u}(0)'$ does not contain a let binder of maximal rank by condition (i). A delicate point is that it may contain some new linear variable b of maximal rank, which is bound in a wider context, influencing the pattern size. However by condition (ii), such a variable is propagated to a (aux)-redex, *i.e.*, we have:

$$\mathsf{let}\ x = u_1 :: u_2\ \mathsf{in}\ C[l_i] \quad \rightarrow \quad \mathsf{let}\ b :: x = u_1 :: u_2\ \mathsf{in}\ C[l_i'].$$

Then b can be immediately killed by the reduction rule (cons). Hence the new variables in $\mathbf{u}(0)'$ do not contribute to the pattern size after all. Since the variable a is killed by the reduction, the pattern size has strictly decreased by 1.

On the other hand, the size $|t|$ increases at most by the size $m(t)$ of a maximal box in t (the *box size* of t). Hence after reducing all $\prec$-minimal redexes in L, the term size is bounded by $|t| + m(t) \cdot |t|_p \leq |t| \cdot (|t|_p + 1)$. The box size may increase, but is bounded by $|t| \cdot (|t|_p + 1)$. By repeating this at most d times, one can eliminate all redexes of maximal rank, obtaining a term of size $|t| \cdot (|t|_p + 1)^d \leq (|t| + 1)^{d+1}$. Reduction to the l-normal form only takes a quadratic number of steps.

This way the rank of a term decreases by 1. By repeating the same procedure r times, we obtain a normal form whose size is bounded by $p_{d,r}(|t|)$, where the polynomial $p_{d,r}$ is determined only by d and r.