Simple Parsimonious Types and Logarithmic Space

Damiano Mazza

CNRS, LIPN UMR 7030 Université Paris 13, Sorbonne Paris Cité
F-93430, Villetaneuse, France
Damiano.Mazza@lipn.univ-paris13.fr

Abstract

We present a functional characterization of deterministic logspace-computable predicates based on a variant (although not a subsystem) of propositional linear logic, which we call parsimonious logic. The resulting calculus is simply-typed and contains no primitive besides those provided by the underlying logical system, which makes it one of the simplest higher-order languages capturing logspace currently known. Completeness of the calculus uses the descriptive complexity characterization of logspace (we encode first-order logic with deterministic closure), whereas soundness is established by executing terms on a token machine (using the geometry of interaction).

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.1.3 Complexity Measures and Classes

Keywords and phrases implicit computational complexity, linear logic, geometry of interaction

Digital Object Identifier 10.4230/LIPIcs.CSL.2015.24

1 Introduction

Implicit computational complexity (ICC) is a research field at the intersection of logic, computational complexity and the theory of programming languages, arising from the seminal contributions of, among others, Bellantoni and Cook [3], Leivant and Marion [18], and Jones [16]. ICC may be thought of as the proof-theoretic counterpart of descriptive complexity [15], which is based on model theory instead. They both invoke logic as a guideline for understanding the nature of complexity classes, seeking alternatives to the notion of complete problem which is proper of structural complexity theory.

Linear logic has proved to be quite valuable for ICC, spurring a fruitful line of research [12, 14, 25, 9] which we continue with the present paper: we show how an affine propositional logical system characterizes in a natural way the class L of logspace computable predicates. Such a logical system stems from previous work by the author on the infinitary affine λ-calculus [19, 20]. In particular, the latter work introduced the so-called *parsimonious stratified λ-calculus*, which was shown to capture (non-uniform) polytime computation. In that paper, parsimony was considered merely as a restriction to be added on top of stratification in order to keep the complexity under control. Later, the author realized that parsimony has an independent logical meaning, *i.e.*, it corresponds to a well-defined logical system which is a variant (but not a subsystem) of linear/affine logic.

In its intuitionistic form, parsimonious logic is multiplicative affine logic (*i.e.*, with linear implication $\multimap$, multiplicative conjunction $\otimes$ and free weakening; categorically, the free SMCC with terminal unit) endowed with an exponential modality satisfying what we call *Milner's law* $!A \cong A \otimes !A$. The implication $!A \multimap A \otimes !A$, sometimes called *absorption*, holds in linear logic but its converse, which we deem *co-absorption*, does not.[1] It does hold in

[1] To be fair, in linear logic one should look for $!A \cong (A \,\&\, 1) \otimes !A$. Nevertheless, although implications in

24th EACSL Annual Conference on Computer Science Logic (CSL 2015).
Editor: Stephan Kreutzer; pp. 24–40
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

differential linear logic (it is the type of the derivative operator), but it is not the inverse of absorption. Therefore, Milner's law is not verified in any known variant of linear/affine logic.

From the computational point of view, Milner's law asserts that $!A$ is the type of streams of type A: absorption is "pop" and co-absorption is "push". The fact that these operations are inverses of each other is why we speak of (infinite) streams rather than (finite) stacks. Accordingly, the λ-calculus arising from parsimonious logic natively supports streams. Remember that parsimony arises from an infinitary affine calculus, so we are in principle capable of dealing with truly infinite streams. This yields *non-uniform* computation (in the same sense as circuit families) and is the object of [20, 21]. However, in the present paper we focus on *uniform* computation, which means that the streams will be morally finite: we will consider only terms of the form $u_1 :: \cdots :: u_n :: !t$ where $!t \equiv t :: !t$, *i.e.*, the stream is ultimately constant. In [21], it is proved that the non-uniform simply-typed parsimonious λ-calculus captures $\mathsf{L/poly}$ (non-uniform logspace). Although closely related, this does not imply the results presented here, it rather complements them.

The question we address here is: what is the expressive power of the simply-typed parsimonious λ-calculus? More precisely, if $\mathsf{Str} := !(o \multimap o) \multimap !(o \multimap o) \multimap o \multimap o$ is the affine version of the type of Church binary strings and $\mathsf{Bool} := o \otimes o \multimap o \otimes o$ is the affine type of Booleans, what languages are decidable by simply-typed parsimonious terms of type $\mathsf{Str}[A] \multimap \mathsf{Bool}$?[2] If we call PL the class of such languages, as anticipated above we have:

▶ **Theorem 1.** $\mathsf{PL} = \mathsf{L}$.

By contrast, the analogous question for the usual simply-typed λ-calculus, or for propositional linear logic, lacks to our knowledge such a straightforward, standard answer. This, we believe, supports the claim that parsimony is an interesting and natural notion.

The proof of Theorem 1 is of course in two parts. Completeness ($\mathsf{L} \subseteq \mathsf{PL}$, Sect. 3) is shown by programming in PL the descriptive complexity characterization of L (*i.e.*, first-order logic with deterministic transitive closure). The non-trivial part is, essentially, solving reachability for directed forests, a paradigmatic L-complete problem.

The main ingredient of soundness ($\mathsf{PL} \subseteq \mathsf{L}$, Sect. 5) consists of proof nets. These allow turning the normalization of terms into the execution of an automaton, Danos and Regnier's interaction abstract machine (IAM) [8], based on Girard's geometry of interaction (GoI) [11]. In its IAM formulation, the GoI computes the normal form of a proof net π by considering a token traveling through the nodes of π, instead of applying rewriting rules. To control the movements of the token, the machine has to keep track of a certain amount of information, consisting of two stacks S and B. The stack S is binary, and its length is bounded by the height of the types of π. The length of B is bounded by the *depth* of π (the maximum number of nested exponential modalities in the types of π) but, in linear logic, its elements may be quite complex. In parsimonious logic, the elements of B are just integers: they correspond to positions within streams. Therefore, running the IAM on a proof net π of size s, height h and depth d needs space $O(\log s + h + d \log m)$: $\log s$ is for the token's position, h for the S stack and $d \log m$ for the B stack, if m is the largest integer stored in it during execution.

The interesting case is when π is the translation of $t\,\underline{w}$, with $t : \mathsf{Str}[A] \multimap \mathsf{Bool}$ and $\underline{w}$ a Church string. Then, s is $O(|w|)$, whereas h and d are $O(1)$ (they only depend on A, which is fixed). Therefore, if we manage to prove that m is polynomial in s, we have a logarithmic

both directions are provable, they are not inverses of each other.

[2] $\mathsf{Str}[A]$ denotes Str where the base type o is replaced by an arbitrary simple type A. In the absence of polymorphism, it is common to allow such type expansions.

bound in $|w|$, as desired. Such a bound on m may be proved directly, as done in [21], or by combining the bound we previously gave in [20] with a nice result of Gaboardi, Roversi and Vercelli [10], which is the strategy we adopt here.

Related work

Implicit characterizations of L abound: of recursive-theoretic nature [22, 17], using imperative languages [16, 4] and higher-order languages [24, 25, 6]. Of these, only the latter are immediately comparable to our work. Another paper explicitly relating streams and logarithmic space is [23], which however does not have much of a connection with our work: the authors consider there corecursive definitions, *i.e.*, algorithms on infinite streams (as opposed to finite strings) and the space complexity they refer to is not the usual decision problem complexity.

The GoI plays a key role in both [25, 6]. The difference here is not so much in the use of the GoI, which is quite similar, but in the underlying programming language: in that work, the author(s) take the standpoint that the fundamental primitive of sublinear space computation is interaction (a point of view already taken in [24]) and forge their programming language around this. This leads, for instance, to the use of non-standard types for encoding strings, namely $\mathsf{Nat} \multimap \mathsf{Three}$ (a binary string x is seen as a function mapping i to x_i or to $\bot$ if $i \geq |x|$), whereas the language of [24] has an explicit list type.

With respect to the above, we believe that the highlight of our characterization is that it is closer to the original spirit of applying linear logic to ICC [12]: it is purely logical (there is no primitive datatype) and employs standard types. Our characterization also improves on previous ones in terms of simplicity: the types of [25] include full polymorphism and indexed exponential modalities, whereas the categorical construction of [6], while elegant, also yields a sort of indexed exponential modality in types, making type inference not straightforward (see [7]). By contrast, our calculus is simply-typed, has only 9 typing rules (Fig. 1) which are essentially syntax-directed, so type inference is easier. Programming is of course restricted but, as hopefully showcased by Sect. 3, quite reasonably so if we consider that all programs must run in logarithmic space.

2 The Parsimonious Lambda-Calculus

2.1 Terms and reduction

We let a (resp. x) range over a countably infinite set of *affine* (resp. *exponential*) variables. An *occurrence* of exponential variable is of the form x_i, where $i \in \mathbb{N}$. Occurrences of exponential variables are naturally ordered by $x_i \leq y_j$ whenever $x = y$ and $i \leq j$. Let Θ be a set of occurrences of exponential variables. We write $\uparrow\Theta$ for the upward closure of Θ. We denote by $\Theta \setminus \{x\}$ the set obtained from Θ by removing all occurrences of the form x_i (if any).

Parsimonious terms belong to the grammar

$$t, u ::= a \mid \lambda a.t \mid tu \mid \mathsf{let}\, a \otimes b = u \,\mathsf{in}\, t \mid t \otimes u \mid x_i \mid \mathsf{let}\, !x = u \,\mathsf{in}\, t \mid !t \mid t :: u.$$

However, they obey several constraints on how variables may appear, so we introduce them by means of a well-forming relation $\Theta; \Phi \rhd t$, where t is a (parsimonious) term, Φ is the set of its free affine variables and Θ is the set of its free *virtual* occurrences of exponential variables (which may be infinite):

- $\emptyset; \{a\} \rhd a$ and $\{x_i\}; \emptyset \rhd x_i$ always hold;
- if $\Theta; \Phi \rhd t$, then $\Theta; \Phi \setminus \{a\} \rhd \lambda a.t$;

- if $\Theta;\Phi \rhd t$ and $\Theta';\Phi' \rhd u$ and $\Theta \cap \Theta' = \Phi \cap \Phi' = \emptyset$, then $\Theta \cup \Theta';\Phi \cup \Phi' \rhd tu$, $\Theta \cup \Theta';\Phi \cup \Phi' \rhd t \otimes u$, $\Theta \cup \Theta';\Phi \cup \Phi' \rhd t :: u$, $\Theta \cup \Theta';\Phi \setminus \{a,b\} \cup \Phi' \rhd \mathsf{let}\, a \otimes b = u \,\mathsf{in}\, t$ and $\Theta \setminus \{x\} \cup \Theta';\Phi \cup \Phi' \rhd \mathsf{let}\, !x = u \,\mathsf{in}\, t$;
- if $\Theta;\emptyset \rhd t$ and every exponential variable has at most one occurrence in Θ, then $\uparrow\Theta;\emptyset \rhd !t$.

Let $\Theta;\Phi \rhd t$. We denote by t^{++} the term obtained from t by substituting each free occurrence x_i with x_{i+1}. We have $\Theta^{++};\Phi \rhd t^{++}$, where Θ^{++} is defined in the obvious way.

Terms of the form $!t$ are called *boxes*. Apart from disallowing free affine variables, the main purpose of the well-forming condition for boxes is to ensure that, if x occurs free in a parsimonious term t, then it occurs free in at most one box and at most once therein and, moreover, the index of such an occurrence, denoted by $\mathrm{maxind}_x(t)$, is the greatest in t.

Streams are terms generated by $\mathbf{u} ::= !t \mid t :: \mathbf{u}$. *Structural equivalence* is the contextual closure of the equation $!t \equiv t :: !(t^{++})$. This justifies the name "stream": the term $\mathbf{u} := u_1 :: \cdots :: u_n :: !t$ is morally an infinite stream $u_1 :: \cdots :: u_n :: t :: t^{++} :: (t^{++})^{++} :: \cdots$. Accordingly, given $i \in \mathbb{N}$, we define $\mathbf{u}(i)$ to be u_{i+1} if $i < n$ and $t^{++(i-n \text{ times})}$ if $i \geq n$. We also define $|\mathbf{u}| := n$.

We now define reduction. To avoid the commutative rules induced by the presence of let binders, we use Accattoli's contextual approach (see for instance [1]). We define let-*contexts* as

$$L ::= [\cdot] \mid \mathsf{let}\, \mathsf{p} = t \,\mathsf{in}\, L,$$

where p stands for $a \otimes b$ or $!x$. We denote by $L[t]$ the substitution of the term t for the hole $[\cdot]$ in the let-context L. We may now introduce the reduction rules:

$$\begin{array}{rcl} L[(\lambda a.t)]u & \to_\beta & L[t[u/a]] \\ \mathsf{let}\, a \otimes b = L[u \otimes v] \,\mathsf{in}\, t & \to_\otimes & L[t[u/a, v/b]] \\ \mathsf{let}\, !x = L[\mathbf{u}] \,\mathsf{in}\, t & \to_! & L[t\{\mathbf{u}/x\}] \end{array}$$

Modulo the presence of let-contexts, the rules β and $\otimes$ are standard. In the ! rule, in case x appears in a box in t, we require that $\mathrm{maxind}_x(t) \geq |\mathbf{u}|$. If x does not appear in a box, the rule may always be applied. In any case, $t\{\mathbf{u}/x\}$ stands for t in which every x_i is substituted by $\mathbf{u}(i)$. All rules are readily verified to preserve parsimony.

One-step reduction, denoted by $\to$, is defined as the contextual closure of the above rules, plus closure under structural equivalence, *i.e.*, $t \equiv t' \to u' \equiv u$ implies $t \to u$.

Reduction may be shown to be confluent. However, termination is not guaranteed: if we let $\Delta := \lambda !x.x_0 !x_1$ and $\Omega := \Delta !\Delta$, we have $\Omega \equiv \Delta(\Delta :: !\Delta) \to \Omega$. In fact, the untyped parsimonious λ-calculus is Turing-complete. This follows from observing that, although parsimony seems to exclude general fixpoint combinators, we do have *affine* fixpoints and these are enough to encode partial recursive functions, because the minimization scheme is an affine recurrence.[3]

Note how the syntax allows one to recover unambiguously whether a variable is affine or exponential. For this reason, we will occasionally use any letter to denote any kind of variable. It will also be convenient to use the abbreviations

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

as well as combinations such as $\lambda a \otimes !x.t := \lambda c.\mathsf{let}\, a \otimes d = c \,\mathsf{in}\, \mathsf{let}\, !x = d \,\mathsf{in}\, t$.

[3] The function $g(\overline{x}) := (\mu y.f(\overline{x}, y) = 0)$ may be defined as $g(\overline{x}) := h(0, \overline{x})$ where $h(n, \overline{x}) := \mathsf{if}\, (f(\overline{x}, n) = 0)\, \mathsf{then}\, n\, \mathsf{else}\, h(n+1, \overline{x})$. This recursive definition is affine because h appears only once on the right.

$$\frac{}{\Gamma;\Delta,a:A\vdash a:A}\ \mathsf{ax} \qquad \frac{\Gamma;\Delta,a:A\vdash t:B}{\Gamma;\Delta\vdash\lambda a.t:A\multimap B}\ \multimap\mathsf{I} \qquad \frac{\Gamma;\Delta\vdash t:A\multimap B\quad\Gamma';\Delta'\vdash u:A}{\Gamma,\Gamma';\Delta,\Delta'\vdash tu:B}\ \multimap\mathsf{E}$$

$$\frac{\Gamma;\Delta\vdash t:A\quad\Gamma';\Delta'\vdash u:B}{\Gamma,\Gamma';\Delta,\Delta'\vdash t\otimes u:A\otimes B}\ \otimes\mathsf{I} \qquad \frac{\Gamma';\Delta'\vdash u:A\otimes B\quad\Gamma;\Delta,a:A,b:B\vdash t:C}{\Gamma,\Gamma';\Delta,\Delta'\vdash\mathsf{let}\,a\otimes b=u\,\mathsf{in}\,t:C}\ \otimes\mathsf{E}$$

$$\frac{;\overline{a}:\Gamma\vdash t:A}{\overline{x}:\Gamma;\vdash !t[\overline{x}_0/\overline{a}]:!A}\ !\mathsf{I} \qquad \frac{\Gamma';\Delta'\vdash u:!A\quad\Gamma,x:A;\Delta\vdash t:C}{\Gamma,\Gamma';\Delta,\Delta'\vdash\mathsf{let}\,!x=u\,\mathsf{in}\,t:C}\ !\mathsf{E}$$

$$\frac{\Gamma,x:A;\Delta,a:A\vdash t:C}{\Gamma,x:A;\Delta\vdash t^{x++}[x_0/a]:C}\ \mathsf{abs} \qquad \frac{\Gamma;\Delta\vdash t:A\quad\Gamma';\Delta'\vdash u:!A}{\Gamma,\Gamma';\Delta,\Delta'\vdash t::u:!A}\ \mathsf{coabs}$$

Figure 1 The simply typed parsimonious calculus **PL**.

2.2 Simple types

The *simple types* are the formulas of intuitionistic propositional linear logic, generated by

$$A,B ::= o \mid A\multimap B \mid A\otimes B \mid !A,$$

where o is a ground type (we consider only one, although of course our results hold for any number of ground types). If A and B are types, we denote by $A[B]$ the type obtained by replacing every occurrence of o in A with B.

The type system we consider, which we call **PL** (for *parsimonious logic*), is defined in Fig. 1. In the abs rule, t^{x++} is defined like t^{++} but only the free occurrences of x are re-indexed. A straightforward induction on the last rule of the derivation gives

▶ **Lemma 2.** *Let* $\Gamma;\Delta\vdash t:A$. *Then:*
parsimony: *t is parsimonious;*
typical ambiguity: *for any type B, $\Gamma[B];\Delta[B]\vdash t:A[B]$.*

In fact, the type assignment is an instance of the Curry-Howard correspondence: if we forget term annotations, the type derivations are proofs in natural deduction and reduction of terms corresponds to normalization. The underlying logical system is the parsimonious logic mentioned in the introduction. The isomorphism $!A\cong A\otimes !A$ is realized by the terms

$$\lambda!x.x_0\otimes!x_1:!A\multimap A\otimes!A \qquad\qquad \lambda a\otimes!x.a::!x_0:A\otimes!A\multimap!A,$$

which use absorption and co-absorption, respectively.

In the sequel, we will sometimes write $A[]$ for $A[B]$ when the type $B\neq o$ is unimportant. This lack of information is harmless for composition: point 2 of Lemma 2 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 a *flat* typing, *i.e.*, of the form $A\multimap A$.

3 Simply-typed Parsimonious Programming

3.1 Basic data types

The types Nat, Bool and Str of unary (Church) integers, Booleans and binary strings, respectively, are defined in Fig. 2, together with the encoding of integers and Booleans. For binary strings, the encoding is similar: if $w=w_1\cdots w_n\in\mathbb{W}$, we have

$$\underline{w}:=\lambda!s^0.\lambda!s^1.\lambda z.s_0^{w_1}(\ldots s_i^{w_n}z\ldots),$$

$\mathsf{Nat} := !(o \multimap o) \multimap o \multimap o$, $\mathsf{Bool} := o \otimes o \multimap o \otimes o$, $\mathsf{Str} := !(o \multimap o) \multimap !(o \multimap o) \multimap o \multimap o$

$\underline{n}$	$:=$	$\lambda!s.\lambda z.s_0(\dots s_{n-1} z \dots)$	$: \mathsf{Nat}$
succ	$:=$	$\lambda n.\lambda!s.\lambda z.s_0(n\,!(s_1)\,z)$	$: \mathsf{Nat} \multimap \mathsf{Nat}$
pred	$:=$	$\lambda n.\lambda!s.\lambda z.n\,((\lambda a.a) :: !s_0)\,z$	$: \mathsf{Nat} \multimap \mathsf{Nat}$
dup	$:=$	$\lambda n.\mathsf{It}(n, \lambda m_1 \otimes m_2.(\mathsf{succ}\ m_1) \otimes (\mathsf{succ}\ m_2), \underline{0} \otimes \underline{0})$	$: \mathsf{Nat}[] \multimap \mathsf{Nat} \otimes \mathsf{Nat}$
store	$:=$	$\lambda n.\mathsf{It}(n, \lambda!x.!(\mathsf{succ}\ x_0), !\underline{0})$	$: \mathsf{Nat}[] \multimap !\mathsf{Nat}$
$\mathsf{tt} := \lambda c \otimes d.c \otimes d$,		$\mathsf{ff} := \lambda c \otimes d.d \otimes c$	$: \mathsf{Bool}$
not	$:=$	$\lambda b.\lambda c \otimes d.b(d \otimes c)$	$: \mathsf{Bool} \multimap \mathsf{Bool}$
xor	$:=$	$\lambda b.\lambda b'.\lambda c.b(b'c)$	$: \mathsf{Bool} \multimap \mathsf{Bool} \multimap \mathsf{Bool}$
and	$:=$	$\lambda b.\lambda b'.\mathsf{let}\ c \otimes d = b(b' \otimes \mathsf{ff})\ \mathsf{in}\ c$	$: \mathsf{Bool}[] \multimap \mathsf{Bool} \multimap \mathsf{Bool}$
len	$:=$	$\lambda w.\mathsf{It}(w, \mathsf{succ}, \mathsf{succ}, \underline{0})$	$: \mathsf{Str}[] \multimap \mathsf{Nat}$
shift	$:=$	$\lambda!x.!x_1$	$: !A \multimap !A$
toStrm	$:=$	$\lambda w.\mathsf{It}(w, \lambda s.(\mathsf{ff} :: s), \lambda s.(\mathsf{tt} :: s), !\mathsf{ff})$	$: \mathsf{Str}[] \multimap !\mathsf{Bool}$
leq	$:=$	$\lambda m.\lambda n.\mathsf{let}\ !x = \mathsf{It}(n, \mathsf{shift}, \mathsf{It}(m, \lambda s.(\mathsf{ff} :: s), !\mathsf{tt}))\ \mathsf{in}\ x_0$	$: \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$
isOne	$:=$	$\lambda w.\lambda n.\mathsf{let}\ !x = \mathsf{It}(n, \mathsf{shift}, \mathsf{toStrm}\ w)\ \mathsf{in}\ x_0$	$: \mathsf{Str}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$

Figure 2 Data types, encodings and basic functions.

where the j-th occurrence of s^0 from the left has index $j-1$, and similarly for s^1. For instance, $\underline{001} = \lambda!s^0.\lambda!s^1.\lambda z.s^0_0(s^0_1(s^1_0 z))$.

The type Nat supports iteration $\mathsf{It}(n, \mathsf{step}, \mathsf{base}) := n\,!(\mathsf{step}')\,\mathsf{base}$, typed as:

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

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

Unary successor and predecessor are implemented as in Fig. 2. Since their types are flat, they may be iterated to obtain addition and subtraction, of type $\mathsf{Nat}[] \multimap \mathsf{Nat} \multimap \mathsf{Nat}$. This is again flat with respect to the second argument, so a further iteration on addition leads to multiplication, of type $\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}$. Unary integers 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 $\mathsf{Nat}[] \multimap \mathsf{Nat}$.

These constructions can all be extended to the type Str, which also supports iteration, flat successors and predecessor, concatenation, and is duplicable and storable.

For the Booleans, we adopt the multiplicative type Bool used in [26]. This type too is duplicable and storable. An advantage of multiplicative Booleans is that they support flat exclusive-or in addition to flat negation (see Fig. 2). On the other hand, conjunction (and disjunction) has one non-flat argument (see again Fig. 2). This would be the case of exclusive-or too if we had chosen the traditional Boolean type $o \multimap o \multimap o$. We write $\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ u$ for $\mathsf{let}\ c \otimes \_ = b\ (t \otimes u)\ \mathsf{in}\ c$, which has type A if $t, u : A$ and $b : \mathsf{Bool}[A]$.

In the sequel we will abusively use affine variables of duplicable types non-linearly, *e.g.*, if $n : \mathsf{Nat}[]$ we write $n \otimes n$ meaning $\mathsf{let}\ n' \otimes n'' = \mathsf{dup}[n]\ \mathsf{in}\ n' \otimes n''$. Similarly, if a term step contains a free affine variable a of storable type A, we will abusively consider the result of its iteration to still have a free variable $a : A[]$ (instead of an exponential variable of type $!A$), by implicitly composing with store.

It will be useful to consider for loops, derived from iteration. Given $\mathsf{step}[i] : A \multimap A$ containing a free affine variable $i : \mathsf{Nat}[]$, we define $\mathsf{step}_+ := \lambda!j \otimes a.!(\mathsf{succ}\ j_1) \otimes (\mathsf{step}[j_0/i]\ a) :$

$$
\begin{aligned}
\mathsf{strmToW} &:= \lambda!x.\lambda m.m\,!(\mathsf{if}\ x_0\ \mathsf{then}\ \mathsf{succ}^1\ \mathsf{else}\ \mathsf{succ}^0)\,\underline{\epsilon} : !\mathsf{Bool}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Str}\\
\mathsf{forall} &:= \lambda w.\mathsf{It}(w, \lambda b.\mathsf{ff}, \lambda b.b, \mathsf{tt}) : \mathsf{Str}[] \multimap \mathsf{Bool}\\
\mathsf{Univ} &:= \lambda!R.\lambda m.\mathsf{forall}(\mathsf{strmToW}(\mathsf{for}\ k < m\ \mathsf{from}\ !\mathsf{ff}\ \mathsf{do}\ \lambda s.(R\,m\,k) :: s))\\
\mathsf{mkDep}_R &:= \mathsf{if}\ (R\,m\,i\,j)\ \mathsf{then}\ \lambda b \otimes !x \otimes !y.(\mathsf{xor}\,b\,x_0) \otimes !x_1 \otimes \mathsf{ff} :: !y_0\\
&\qquad\qquad \mathsf{else}\ \lambda b \otimes !x \otimes !y.b \otimes !x_1 \otimes x_0 :: !y_0\\
\mathsf{rev} &:= \lambda s.\mathsf{let}\ s' \otimes \_ = \mathsf{It}(m, \lambda!x \otimes !y.(y_0 :: !x_0) \otimes !y_1, !\mathsf{ff} \otimes s)\ \mathsf{in}\ s' : !\mathsf{Bool} \multimap !\mathsf{Bool}\\
\mathsf{mkFun}_R &:= \lambda s.\mathsf{let}\ s' \otimes \_ = \mathsf{for}\ j < m\ \mathsf{from}\ !\mathsf{ff} \otimes s\ \mathsf{do}\\
&\qquad \lambda p \otimes q.\mathsf{let}\ b \otimes \_ \otimes q' = (\mathsf{for}\ i < m\ \mathsf{from}\ (\mathsf{ff} \otimes q \otimes !\mathsf{ff})\ \mathsf{do}\ \mathsf{mkDep}_R[m,i,j])\ \mathsf{in}\\
&\qquad\qquad (b :: p) \otimes (\mathsf{rev}\ q')\\
&\qquad \mathsf{in}\ \mathsf{rev}\ s'\\
\mathsf{DTC}_R &:= \lambda m.\lambda n.\lambda n'.\mathsf{for}\ k < m\ \mathsf{from}\ \mathsf{ff}\ \mathsf{do}\\
&\qquad \mathsf{let}\ !x = \mathsf{It}(n', \mathsf{shift}, \mathsf{It}(k, \mathsf{mkFun}_R[m], \mathsf{It}(n, \lambda p.\mathsf{ff} :: p, \mathsf{tt} :: !\mathsf{ff})))\ \mathsf{in}\\
&\qquad \mathsf{if}\ x_0\ \mathsf{then}\ \lambda b.\mathsf{tt}\ \mathsf{else}\ \lambda b.b
\end{aligned}
$$

Figure 3 Encoding universal quantification and deterministic transitive closure.

$!\mathsf{Nat}[] \otimes A \multimap !\mathsf{Nat}[] \otimes A$ and, given $\mathsf{base} : A$, we set

$$\mathsf{for}\ i < n\ \mathsf{from}\ \mathsf{base}\ \mathsf{do}\ \mathsf{step} \quad := \quad \mathsf{It}(n, \mathsf{step}_+, !\underline{0} \otimes \mathsf{base}).$$

3.2 Expressing logspace computation

The class L of languages decidable by deterministic Turing machines with a logarithmically bounded work tape has a nice presentation in terms of descriptive complexity, due to Immerman [15]: it corresponds to first-order logic over totally ordered finite structures with the addition of a deterministic transitive closure operator. This may be equivalently presented in recursion-theoretic terms, as we do below.

We consider the following set of basic functions: the constant $0 \in \mathbb{N}$; negation $\mathsf{not} : \mathbb{B} \to \mathbb{B}$ and conjunction $\mathsf{and} : \mathbb{B}^2 \to \mathbb{B}$; the functions $\mathsf{leq} : \mathbb{N}^2 \to \mathbb{B}$, $\mathsf{sum}, \mathsf{times} : \mathbb{N}^3 \to \mathbb{B}$ corresponding to the integer relations $m \leq n$, $m + n = k$ and $m \cdot n = k$; the function $\mathsf{len} : \mathbb{W} \to \mathbb{N}$ returning the length of a string and $\mathsf{isOne} : \mathbb{W} \times \mathbb{N} \to \mathbb{B}$ s.t. $\mathsf{isOne}(w, i) = 1$ iff the i-th bit of w is 1. Now call $\mathcal{L}$ the smallest set of functions containing the above basic functions and closed by composition and the following schemata:

- **universal quantification:** if $R : \Gamma \times \mathbb{N}^2 \to \mathbb{B} \in \mathcal{L}$, then $\forall R : \Gamma \times \mathbb{N} \to \mathbb{B}$ mapping $(\gamma, m) \mapsto 1$ iff $R(\gamma, m, i) = 1$ for all $i < m$ is also in $\mathcal{L}$;
- **deterministic transitive closure:** let $R : \Gamma \times \mathbb{N}^{2k+1} \to \mathbb{B} \in \mathcal{L}$. This induces a partial map $R_* : \Gamma \times \mathbb{N}^{k+1} \to \mathbb{N}^k$ mapping $(\gamma, m, \overline{n}) \mapsto \overline{n}'$ if $\overline{n}'$ is unique s.t. $R(\gamma, m, \overline{n}, \overline{n}') = 1$, or undefined otherwise. Then, $\mathcal{L}$ also contains $\mathrm{DTC}(R) : \Gamma \times \mathbb{N}^{2k+1} \to \mathbb{B}$ mapping $(\gamma, m, \overline{n}, \overline{n}') \mapsto 1$ iff there exist $\overline{n}_0, \ldots, \overline{n}_l \in \mathbb{N}^k$, with $\overline{n}_0 = \overline{n}$, $\overline{n}_l = \overline{n}'$ and $\overline{n}_i \in \{0, \ldots, m-1\}^k$ for all $0 < i \leq l$, such that $R_*(\gamma, m, \overline{n}_i) = \overline{n}_{i+1}$ for all $0 \leq i < l$.

The class L corresponds exactly to the predicates $\mathbb{W} \to \mathbb{B}$ in $\mathcal{L}$.

We have already seen that the basic functions are representable in **PL**: they are either in Fig. 2 or were discussed in the previous section. The universal quantification schema is represented by the higher order term $\mathsf{Univ} : !(\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}[]) \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$ defined in Fig. 3. The idea is the following: given $R : \mathbb{N}^2 \to \mathbb{B}$ and $m \in \mathbb{N}$, we use iteration

to build a stream of Booleans whose first m bits contain $R(m,0),\dots,R(m,m-1)$; then, we use $\mathsf{strmToW}$ to convert this into a string and we check that it consists entirely of ones.

Note that the variable $!R$ representing the relation on which universal quantification is applied is exponential because it appears free in the subterm $\lambda s.(R\,m\,k) :: s$, which is iterated. This means that, when we want to apply universal quantification to $t : \Gamma \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$ representing a function in $\mathcal{L}$, we will first have to convert it to a term of type $!\Gamma \multimap !(\mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool})$ and then apply Univ to obtain a term of type $!\Gamma \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$. The extra modalities in $!\Gamma$ may then be removed because all types in Γ are storable (they are either Nat, Bool or Str). The same remark applies below.

Let us turn to representing $\mathrm{DTC}(R)$ with $R : \Gamma \to \mathbb{N}^{2k+1} \to \mathbb{B}$. First of all, we will restrict to the case $k = 1$. The general case may be treated by encoding a pairing function, which we omit here for briefness. Second, we observe that the particular determinization R_* of R used in the definition of DTC is inessential: we may as well define $R_*(\gamma, m, i)$ to be the smallest j such that $R(\gamma, m, i, j) = 1$, or undefined otherwise. Indeed, the important case is when R is already deterministic (*i.e.*, a partial function), in which the determinization is irrelevant. We will adopt the second definition here; it is possible to deal with the first at the expense of a more complex encoding.

The representation $\mathsf{DTC}_R : \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Nat}[] \multimap \mathsf{Bool}$ is given in Fig. 3, which we now explain. If $R : \mathbb{N}^3 \to \mathbb{B}$, computing $\mathrm{DTC}(R)(m,n,n')$ amounts to determining whether there is a path from n to n' in a graph G whose nodes are $[m] := \{0,\dots,m-1\}$ and s.t. there is an edge (n,n') iff $R_*(m,n) = n'$, so the out-degree of G is at most 1. To do this, we imagine a token traveling in G, its position being represented by a stream of type $!\mathsf{Bool}$ which is ff everywhere except where the token is. The edges of G may now be seen as a stream transformation $\varphi : !\mathsf{Bool} \multimap !\mathsf{Bool}$. Initially, the stream is tt at position n; applying φ will make the token move, and we may determine the existence of a path by checking the value at position n' after at most m applications of φ.

The idea behind the definition of φ is best explained with an example. Suppose that $m = 4$ and that the edges of G are $\{(0,1),(1,1),(3,2)\}$. Then, $\varphi = \lambda !x.\mathsf{ff} :: (\mathsf{xor}\,x_0\,x_1) :: x_3 :: \mathsf{ff} :: !x_4$. This works because the input stream $!x$ contains exactly one bit set to tt, so at most one of x_0, x_1 will be tt and exclusive-or is equivalent to disjunction. We cannot use disjunction because it is not flat. Observe by the way that the simultaneous presence of flat disjunction and flat duplication (*i.e.*, if dup had type $\mathsf{Bool} \multimap \mathsf{Bool} \otimes \mathsf{Bool}$ instead of its present type $\mathsf{Bool}[\mathsf{Bool} \otimes \mathsf{Bool}] \multimap \mathsf{Bool} \otimes \mathsf{Bool}$) would allow this solution to work for arbitrary relations (*i.e.*, graphs of arbitrary out-degree) and we would be able to compute arbitrary transitive closures, which is impossible unless $\mathsf{L} = \mathsf{NL}$.

The tricky task now is to compute φ from R. This is realized by $\mathsf{mkFun}_R : !\mathsf{Bool} \multimap !\mathsf{Bool}$, which operates by manipulating two streams p and q, the latter being initialized as the input stream s. For each $j \in [m]$, we determine its dependencies, *i.e.*, those $i \in [m]$ s.t. $R(m,i,j) = 1$. This is done by iterating over all $i \in [m]$ the term $\mathsf{mkDep}_R : C \multimap C$ (where $C := \mathsf{Bool} \otimes !\mathsf{Bool} \otimes !\mathsf{Bool}$): if $R(m,i,j) = 0$, the i-th element is saved in an auxiliary stream (it may contain the token, so we must preserve it); otherwise, we xor the current result with the i-th element and set this element to ff, so that it won't be considered later (if the token was there, it has now moved). This yields the determinization of R we defined above. At the end of this, the result is pushed to p and we start over with the (possibly) modified q (q also needs to be reversed because traversing it and pushing its elements into an auxiliary stream reversed their order). When we exit the outer loop, p contains the desired stream (but, again, in reverse order).

Finally, the term DTC_R does nothing but looping through all $0 \le k < m$ to determine

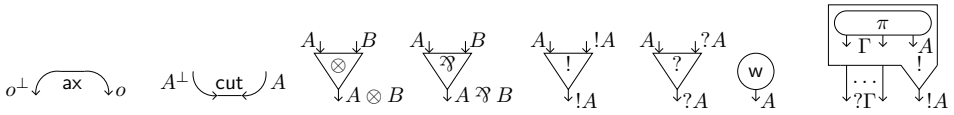


Figure 4 Cells for building nets, with their typing annotations.

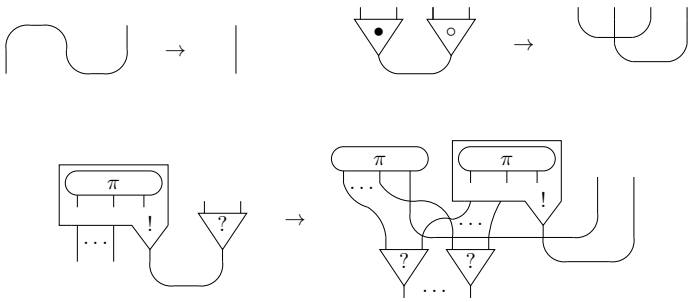


Figure 5 Cut-elimination steps on nets. On the top, $\bullet = \otimes$ and $\circ = \parr$, or $\bullet = !$ and $\circ = ?$.

whether, after k iterations of mkFun_R, the token has moved from n to n'.

4 Upper Bounds

4.1 Nets and cut-elimination (via stratification)

We will consider here *classical* simple types, generated by:

$$A, B ::= o \mid o^\perp \mid A \otimes B \mid A \parr B \mid !A \mid ?A.$$

Linear negation $A^\perp$ is defined as usual via De Morgan.

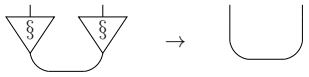
We use the standard definition of *net*, which is a labelled graph built by connecting the *cells* given in Fig. 4, respecting the orientation given therein. Each cell has a number of ports depending on the symbol it carries; the incoming are called *premises* and the outgoing conclusions. The conclusions which are not premise of any cell are called *conclusions* of the net, and so are their types. The rightmost cell in Fig. 4 is called a *box*; the conclusion labelled by $!A$ is called its *principal port*, the other are the *auxiliary ports*. A box contains a net π, whose conclusions are in bijection with the conclusions of the box itself.

The *size* of a net π is the number of its cells. Its *height* is the maximum height of its types (as trees). The *depth* of a cell or port of π is the number of nested boxes it is contained in. The *depth* of π is the maximum depth of its cells.

Nets are usually required to satisfy some form of correctness, yielding *proof nets*. We will not specify any correctness criterion here, we will rather take "proof net" as a synonym of *sequentializable net*, *i.e.*, corresponding to a sequent calculus proof (or to a typing derivation of **PL**, which will be our case).

Cut-elimination steps are defined in Fig. 5. Types (and orientations) are omitted because they can be recovered without ambiguity from Fig. 4. We prove cut-elimination for our nets using an idea of Gaboardi, Roversi and Vercelli [10] and which resorts to *stratification*.

We add the formula $\S A$ to our classical types and a corresponding cell in nets, with premise A and conclusion $\S A$, with the following cut-elimination step:



In this context, we call *plain* a net which contains no § in its types.

▶ **Definition 3** (Indexing, stratified net, level [2]). A *weak indexing* of a net π whose set of ports is P is a function $I : P \longrightarrow \mathbb{Z}$ satisfying the following:

- if p, p' are the conclusions (resp. premises) of an ax (resp. cut) cell of π, then $I(p) = I(p')$;
- if p is the conclusion of a $\otimes$ or $\parr$ cell whose premises are q, q', then $I(p) = I(q) = I(q')$;
- if p is the conclusion of a ! or ? cell whose left and right premises are l and r, then $I(p) = I(r) = I(l) - 1$;
- if p is the conclusion of a box containing π and q is the conclusion of π corresponding to p, then $I(p) = I(q) - 1$;
- if p is the conclusion of a § cell whose premise is q,then $I(p) = I(q) - 1$.

An *indexing* for π is a weak indexing I further satisfying that, for any two conclusions p, p' of π, $I(p) = I(p')$. A *stratified net* is a net admitting an indexing.

Note that, if I is a weak indexing, its range $\mathrm{rg}I$ is obviously a finite set. The *level* of I is $\ell(I) := \max \mathrm{rg}I - \min \mathrm{rg}I$.

Of course, one may easily verify that cut-elimination preserves stratification [2].

Now, the calculus we introduced in [20] (where parsimony was first introduced) is also stratified, *i.e.*, its terms may be mapped to stratified proof nets, in a way entirely analogous to the definitions we will give in Sect. 5.1 below. The polynomial-time normalization result of [20] was proved using linear explicit substitutions (in the style of [1]), which essentially amounts to using proof nets. Therefore, we have:

▶ **Proposition 4.** *Let π be a stratified proof net of size s and level l. Then, π normalizes while keeping the size of all reducts bounded by $O(s^{k(l)})$, where k depends only on l.*

Proof. The proof is similar to the usual normalization proofs for stratified systems of linear logic, such as those in [12, 2]. It actually gives a polynomial bound also on the length of the reduction and holds even in absence of types, although we will not need this. See Lemma 5 of [20]. ◀

The discovery of [10] is that simply typed nets embed in stratified nets, in a way compatible with cut-elimination.

▶ **Proposition 5** (Embedding [10]). *There is an embedding $(-)^{\S}$ of plain nets into stratified nets such that, for all π of height h:*

1. *$(\pi)^{\S}$ is of level at most h;*
2. *$\pi \to \pi'$ implies $(\pi)^{\S} \to^* (\pi')^{\S}$.*

Proof. Part 1 follows from [10, Proposition 1] and part 2 is precisely [10, Proposition 2]. The only delicate point is that those results are proved for linear logic and parsimonious logic is not a subsystem of it, because of coabsorption (the other difference is linearity vs. affinity, but it is inessential). The key observation is that indexings are oblivious to the distinction between ! and ?, so coabsorption behaves exactly as absorption, and coabsorption-free parsimonious logic *is* a subsystem of linear logic.

More precisely, given a net π, we may consider the net π^- in which all ! and ? are replaced by a self-dual modality $\sharp$, with suitable links coming from those for ! and ?, and on which indexings behave accordingly. This net will be well typed because $\sharp$ is self-dual. Then, one may check that π is stratified iff π^- is: an indexing of π induces an indexing of π^- and vice

versa. Now, it is not hard to check that, because of the above obliviousness, the embedding of [10, Proposition 1] works just as well for proof nets of the form π^-. ◀

▶ **Proposition 6.** *A plain proof net π of size s and height h normalizes while keeping the size of all reducts bounded by $O(k'(h) \cdot s^{k(h)})$, where k, k' depend solely on h.*

Proof. A consequence of point 2 of Proposition 5 and Proposition 4. The only thing to check is the size of $(\pi)^{\S}$. Let n be the maximum number of § cells added under a single axiom; the number of axioms is bounded by s, so the size of $(\pi)^{\S}$ is bounded by $(n+1)s$. But n, in turn, is bounded by h. Finally, what is the level of $(\pi)^{\S}$? Since § cells are only added to balance out the presence of exponential cells in π, the level will not exceed the depth of π, which is also bounded by h. So Proposition 4 gives us the size bound $(h+1)^{k(h)}s^{k(h)}$ (k is monotonic). ◀

4.2 Geometry of interaction

In the following definition, we use the wildcards $\bullet \in \{\otimes, ⅋\}$ and $\dagger \in \{!, ?\}$.

▶ **Definition 7** (Interaction Abstract Machine)**.** A *stack* is a finite string over $\{\mathsf{p}, \mathsf{q}\} \cup \mathbb{N}$. We use S to range over stacks and write $\alpha \cdot S$ for a stack whose first symbol is α. A stack S *matches* a formula A if: $S = \epsilon$ and $A = o$ or $A = o^\perp$; $S = \mathsf{m} \cdot S'$ with $\mathsf{m} \in \{\mathsf{p}, \mathsf{q}\}$, $A = A' \bullet A''$ and S' matches one of A', A''; $S = n \cdot S'$ for some $n \in \mathbb{N}$, $A = \dagger A'$ and S' matches A'.

A *box identifier* is a finite string over $\mathbb{N}$, ranged over by B. If $B = n_1 \cdots n_k$, we define $\|B\| := \max\{n_1, \dots, n_k\}$.

Given a net π, we define an automaton $\mathsf{IAM}(\pi)$ as follows. Its *states* are tuples (d, p, B, S), where $d \in \{\uparrow, \downarrow\}$ is a *direction*, p is a port of π, B is a box identifier and S is a stack. Such a state is *admissible* if the length of B equals the depth of p and S matches the type of p. The transition relation $\rightsquigarrow$ is the smallest such that:

ax: $(\uparrow, p, B, \epsilon) \rightsquigarrow (\downarrow, p', B, \epsilon)$ whenever p, p' are the conclusions of an ax cell;
cut: $(\downarrow, p, B, S) \rightsquigarrow (\uparrow, p', B, S)$ whenever p, p' are the premises of a cut cell;
•: $(\downarrow, p, B, S) \rightsquigarrow (\downarrow, p', B, \mathsf{m} \cdot S)$ whenever p is the left (resp. right) premise of a • cell and p' its conclusion, in which case $\mathsf{m} = \mathsf{p}$ (resp. $\mathsf{m} = \mathsf{q}$);
†l: $(\downarrow, p, B, S) \rightsquigarrow (\downarrow, p', B, 0 \cdot S)$ whenever p is the left premise of a † cell and p' its conclusion;
†r: $(\downarrow, p, B, n \cdot S) \rightsquigarrow (\downarrow, p', B, (n+1) \cdot S)$ whenever p is the right premise of a † cell and p' its conclusion;
†b: $(\downarrow, p, n \cdot B, S) \rightsquigarrow (\downarrow, p', B, n \cdot S)$ whenever q is the conclusion of a box containing π and p is the conclusion of π corresponding to q;
∗: $(d'^*, p', B', S') \rightsquigarrow (d^*, p, B, S)$ whenever $(d, p, B, S) \rightsquigarrow (d', p', B', S')$, with $\uparrow^* := \downarrow$ and $\downarrow^* := \uparrow$.

Observe that the transitions are deterministic and that they preserve admissible states.

We write $\rightsquigarrow_\pi$ when we want to specify that the transition relation is that of $\mathsf{IAM}(\pi)$ (as opposed to that induced by a different net).

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

The following is a standard property of the GoI. It tells us that $\mathsf{IAM}(\pi)$ behaves identically if we put π inside a box.

▶ **Lemma 8.** *Let p, p' be conclusions. Then, $(\uparrow, p, B, S) \rightsquigarrow^* (\downarrow, p', B', S')$ implies $B' = B$ and $(\uparrow, p, B_0, S) \rightsquigarrow^* (\downarrow, p', B_0, S')$ for all B_0.*

Proof. Standard. ◀

Note that cut-elimination steps preserve the number and ordering of conclusions. Hence, in the following, when $\pi \to \pi'$, we implicitly identify the conclusions of π' with those of π.

▶ **Proposition 9** (Soundness of the GoI). *Let π be a net and let $\pi \to \pi'$. Then, $\rightsquigarrow_{\pi'}^{max} = \rightsquigarrow_{\pi}^{max}$.*

Proof. The multiplicative steps are completely standard and pose no problem, so we assume that the cut-elimination step applied is exponential.

Let $s \rightsquigarrow_{\pi}^{\max} s'$. We need to show that $s \rightsquigarrow_{\pi'}^{\max} s'$. We say that the sequence crosses the left hand side of the rule if it can be decomposed into $s \rightsquigarrow_{\pi}^{*} s_1 \rightsquigarrow_{\pi}^{*} s_2 \rightsquigarrow_{\pi}^{*} s'$ such that the ports p_1, p_2 (not necessarily distinct) of the states s_1, s_2 belong to the interface of the left hand side of the rule. We will show that each crossing $s_1 \rightsquigarrow_{\pi}^{*} s_2$ induces a sequence $s_1 \rightsquigarrow_{\pi'}^{*} s_2$. This is enough to conclude, because the segments which are not crossings also exist in π' for trivial reasons (they concern subnets in which π' is identical to π).

We refer to Fig. 5, with the content of the box being renamed to ρ (since π is the proof net being reduced). Let us fix some notation. In the left hand side, we call $a_1, \dots, a_n$ the auxiliary ports and a_* the principal port of the box, and $b_1, \dots, b_n, b_*$ the associated conclusions of ρ; and q, q', r the left and right premise and the conclusion, respectively, of the ? cell. In the right hand side, we call $b'_1, \dots, b'_n, b'_*$ the conclusions of the copy of ρ which is outside the box, while the other copy still has conclusions b_i; the auxiliary ports of the box are $a'_1, \dots, a'_n$; the ports at the interface, as well as the principal port, are called as in the left hand side. We have three cases:

1. $p_1, p_2 \in \{q, q'\}$;
2. $p_1, p_2 \in \{a_1, \dots, a_n\}$;
3. $p_1 \in \{q, q'\}$ and $p_2 \in \{a_1, \dots, a_n\}$.

Actually, case 3 has a symmetric version with the roles of p_1, p_2 exchanged, but one reduces to the other thanks to the reversibility of transitions.

Case 1 works also in full linear logic. It is easy to see that we must have $p_1 = p_2$ and that everything goes well.

For case 2, let $s_1 = (\uparrow, a_i, B, n \cdot S)$. Then, we have $s_1 \rightsquigarrow_{\pi} (\uparrow, b_i, n \cdot B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b_j, n \cdot B, S') \rightsquigarrow_{\pi} (\downarrow, a_j, B, n \cdot S') = s_2$. In π', a_i becomes the conclusion of a ? link, so we have two cases: either $n = 0$, and then $s_1 \rightsquigarrow_{\pi'} (\uparrow, b'_i, B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b'_j, B, S') \rightsquigarrow_{\pi'} (\downarrow, a_j, B, 0 \cdot S') = s_2$; or $n > 0$, and then $s_1 \rightsquigarrow_{\pi'} (\uparrow, a'_i, B, (n-1) \cdot S) \rightsquigarrow_{\pi'} (\uparrow, b_i, (n-1) \cdot B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b_j, (n-1) \cdot B, S') \rightsquigarrow_{\pi'} (\downarrow, a'_j, B, (n-1) \cdot S') \rightsquigarrow_{\pi'} (\downarrow, a_j, B, n \cdot S') = s_2$. In both cases we used Lemma 8.

For case 3, suppose $p_1 = q$. We have $s_1 = (\downarrow, q, B, S) \rightsquigarrow_{\pi} (\downarrow, r, B, 0 \cdot S) \rightsquigarrow_{\pi} (\uparrow, a_*, B, 0 \cdot S) \rightsquigarrow_{\pi} (\uparrow, b_*, 0 \cdot B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b_i, 0 \cdot B, S') \rightsquigarrow_{\pi} (\downarrow, a_i, B, 0 \cdot S') = s_2$. In π', we have $s_1 \rightsquigarrow_{\pi'} (\uparrow, a'_*, B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b'_i, B, S') \rightsquigarrow_{\pi'} (\downarrow, a_i, B, 0 \cdot S') = s_2$. Suppose now $p_1 = q'$. Then, $s_1 = (\downarrow, q', B, n \cdot S) \rightsquigarrow_{\pi} (\downarrow, r, B, (n+1) \cdot S) \rightsquigarrow_{\pi} (\uparrow, a_*, B, (n+1) \cdot S) \rightsquigarrow_{\pi} (\uparrow, b_*, (n+1) \cdot B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b_i, (n+1) \cdot B, S') \rightsquigarrow_{\pi} (\downarrow, a_i, B, (n+1) \cdot S') = s_2$. In π', we have $s_1 \rightsquigarrow_{\pi'} (\uparrow, a_*, B, n \cdot S) \rightsquigarrow_{\pi'} (\uparrow, b_*, n \cdot B, S) \rightsquigarrow_{\rho}^{*} (\downarrow, b_i, n \cdot B, S') \rightsquigarrow_{\pi'} (\downarrow, a'_i, B, n \cdot S') \rightsquigarrow_{\pi'} (\downarrow, a_i, B, (n+1) \cdot S') = s_2$. We used again Lemma 8.

The above shows that $\rightsquigarrow_{\pi}^{\max} \subseteq \rightsquigarrow_{\pi'}^{\max}$. The converse is entirely analogous. ◀

▶ **Lemma 10.** *Let π be a proof net with no occurrence of ! in its conclusions, of size s and height h, let p_0 be a conclusion of π of type A containing no exponential modality and let $(\uparrow, p_0, \epsilon, S_0) \rightsquigarrow^{*} (d, p_1, B_1, S_1)$. Then, $\|B_1\| = O(k'(h) \cdot s^{k(h)})$.*

Proof. Before starting the proof, let us clarify on the restriction on A: we need it so that we have no problem in eliminating all cuts, similarly to Proposition 11 below.

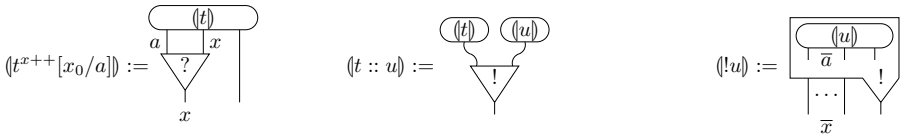
When visiting the ports of π via the execution of $\mathsf{IAM}(\pi)$, we may visit several times the same port. It is well known (cf. [5]) that the box identifier tells us which "copy" of the current port we are visiting (whence the name). For example, a box identifier of the form $3 \cdot 0 \cdot 1$ tells us that π contains three nested boxes $\mathcal{B}_1 \subseteq \mathcal{B}_2 \subseteq \mathcal{B}_3$ and that we are now in copy number 3 of $\mathcal{B}_1$, which is inside copy number 0 of $\mathcal{B}_2$, which is inside copy number 1 of $\mathcal{B}_3$ (in the usual GoI, box identifiers are more complex, in our nets they take this simple form). Therefore, in order to bound the integers appearing in box identifiers, it is enough to bound the number of times each box will be duplicated by an exponential rule while reducing π to its normal form. But this is precisely what is given by Proposition 6. ◀

5 From Simply Typed Terms to Logspace Algorithms

5.1 Translating the calculus into nets

Simple intuitionistic types are mapped to classical types in the usual way: $(\!|o|\!) := o$; $(\!|A \multimap B|\!) := (\!|A|\!)^\perp \parr (\!|B|\!)$; $(\!|A \otimes B|\!) := (\!|A|\!) \otimes (\!|B|\!)$; and $(\!|!A|\!) := !(\!|A|\!)$.

Given a type derivation of $\Gamma; \Delta \vdash t : A$, we associate with it a net of conclusions $?(\!|\Gamma|\!)^\perp, (\!|\Delta|\!)^\perp, (\!|A|\!)$. The definition is by induction on the last rule of the type derivation. The multiplicative cases are standard; η-expansion nets, defined as usual, are employed to translate the ax rule. The exponential rules (except !E, which is just a cut) are given below:



To avoid cluttering the pictures, we only drew the conclusions corresponding to those types in the context which play a role in the typing rules and we marked them with the corresponding variable. Also, types are omitted as they may be inferred from Fig. 1. In the following, we will abusively denote by $(\!|t|\!)$ the translation of a typing derivation of a term t.

The cut-elimination rules we considered are not enough to simulate reduction in the calculus. To obtain something intelligible, we must add garbage collection, *i.e.*, elimination of cuts on w cells. We define

as soon as $\pi = (\!|t|\!)$ for some term t.

▶ **Proposition 11.** *Let $\Gamma; \Delta \vdash t : A$ contain no positive (resp. negative) occurrence of ! in A (resp. Γ) and let t' be the normal form of t. Then, $(\!|t|\!) \to^* (\!|t'|\!)$ (possibly with garbage collection steps).*

Proof. The fact that cut-elimination in proof nets simulates reduction in the λ-calculus is standard, as is our translation. The type restriction here is necessary because, for conciseness, we omitted the rules reducing cuts on the auxiliary ports of boxes. ◀

We may forget about garbage collection steps, because our real interest is the following:

▶ **Corollary 12.** *Let $\Gamma; \Delta \vdash t : A$ and t' be as in Proposition 11 and let $s \rightsquigarrow^{max}_{(\!|t'|\!)} s'$. Then, we also have $s \rightsquigarrow^{max}_{(\!|t|\!)} s'$.*

Proof. We know that $(\!|t|\!) \to^* (\!|t'|\!)$ by possibly using garbage collection steps. Now, it is a standard and easy fact (see for instance [5]) that these may always be postponed, *i.e.*, we have a net π such that $(\!|t|\!) \to^* \pi \to^*_{\mathrm{gc}} (\!|t'|\!)$, where $\to^*_{\mathrm{gc}}$ denotes a reduction sequence consisting entirely of garbage collection steps while the first sequence contains none. But garbage collection does not alter maximal GoI transition sequences, because it substitutes "dead ends", *i.e.*, subnets which no maximal transition sequence may use, with shorter dead ends. Therefore, we already have $s \rightsquigarrow^{\max}_{\pi} s'$ and we may conclude by Proposition 9. ◀

5.2 Synthesis of logspace algorithms

We are at last ready to prove the inclusion $\mathsf{PL} \subseteq \mathsf{L}$. Let $t : \mathsf{Str}[A] \multimap \mathsf{Bool}$. We have to synthesize a deterministic logspace algorithm which, on input $w \in \mathbb{W}$, decides whether $t\,\underline{w} \to^* \mathsf{tt}$. In the following, every dependency (or lack thereof) is expressed w.r.t. $|w|$.

By looking at the net translation of the Boolean tt and using Corollary 12, we know that the above problem is equivalent to determining whether $(\uparrow, p, \epsilon, \mathsf{pp}) \rightsquigarrow^{\max}_{\pi} (\downarrow, p, \epsilon, \mathsf{qp})$, where p is the only conclusion of the net $\pi := (\!|t\,\underline{w}|\!)$. As observed in the introduction, the size of π is $O(|w|)$: $(\!|t|\!)$ is constant and the size of $(\!|\underline{w}|\!)$ is $c(|w|+1)+2|w|+4$, where c is a constant depending on A (it is the size of the η-expansion of $A^\perp, A$). By Lemma 10, the greatest integer that will ever appear in the B stack is bounded by $k'(h)s^{k(h)}$, where s and h are the size and height of π, respectively. This is polynomial because h is constant (it depends only on A). The depth of π is also constant: it is the maximum between the depth of $(\!|t|\!)$ (constant) and the depth of $(\!|\underline{w}|\!)$ (also constant, equal to the nesting of exponentials in A). Therefore, by the space bound given in the introduction, we may seemingly conclude.

There is however a subtlety: while we may assume $(\!|t|\!)$ to be wired into our algorithm, we still need to build $(\!|\underline{w}|\!)$ from w. Actually, instead of building the net, it will be enough to predict the behavior of $\mathsf{IAM}((\!|\underline{w}|\!))$. In fact, the only conclusion of π is a conclusion of $(\!|t|\!)$, so evaluation may start independently of w. At some point, the simulation of the automaton will reach (after crossing a cut at depth 0) a state of the form $(\uparrow, q, \epsilon, S)$, with q being where the conclusion of $(\!|\underline{w}|\!)$ should be. Our algorithm will then compute a stack S_1, according to the following cases:

1. $S = \mathsf{q}\cdot\mathsf{q}\cdot\mathsf{q}\cdot S'$: the automaton is asking for the first bit of w; if this is 1, $S_1 := \mathsf{q}\cdot\mathsf{p}\cdot 0\cdot S'$; if this is 0, $S_1 := \mathsf{p}\cdot 0\cdot S'$;
2. $S = \mathsf{q}\cdot\mathsf{q}\cdot\mathsf{p}\cdot S'$: the automaton is asking for the last bit of w; if this is 1, $S_1 := \mathsf{q}\cdot\mathsf{p}\cdot(n_1-1)\cdot S'$, where n_1 is the number of 1's in w; if this is 0, $S_1 := \mathsf{p}\cdot(n_0-1)\cdot S'$, where n_0 is the number of 0's in w;
3. $S = \mathsf{q}\cdot\mathsf{p}\cdot n\cdot\mathsf{m}\cdot S'$: the automaton is asking for the value of the neighbor of the $(n+1)$-th bit of w whose value is 1, *e.g.*, if $w = 001101$ and $n = 1$, the "second 1" is 001$\underline{1}$01, its left neighbor is 1, which is the "first 1", and its right neighbor is 0, which is the "third 0"). A request for the right (resp. left) neighbor corresponds to $m = \mathsf{p}$ (resp. $m = \mathsf{q}$). If there is no "$(n+1)$-th 1", the algorithm terminates immediately with a negative answer (the transition sequence sought in $\mathsf{IAM}(\pi)$ does not exist). Otherwise, let b be the value of the requested neighbor, and let n' be its position (as the "$(n'+1)$-th 1 or 0"). If $b = 1$, $S_1 := \mathsf{q}\cdot\mathsf{p}\cdot n'\cdot S'$; if $b = 0$ and this is the m-th zero of w, $S_1 := \mathsf{p}\cdot n'\cdot S'$;
4. $S = \mathsf{p}\cdot n\cdot\mathsf{m}\cdot S'$: the automaton is asking for the neighbor of the $(n+1)$-th bit of w whose value is 0. The same procedure as above is applied, with the roles of 0 and 1 reversed.

After computing S_1, the algorithm resumes the simulation of $\mathsf{IAM}(\pi)$ from the state $(\downarrow, q, \epsilon, S_1)$.

To understand where the four cases above come from, it is enough to look at the type of $(\!|\underline{w}|\!)$, namely $(\!|\mathsf{Str}[A]|\!) = ?(A \otimes A^\perp) \parr ?(A \otimes A^\perp) \parr A^\perp \parr A$, which is composed of four

subformulas combined by ⅋'s. The stack S must match that type; the cases correspond to the four subformulas, from right to left. The last two cases, which are similar, correspond both to $?(A \otimes A^{\perp})$; the integer and the constant m in the stack are there to match this formula. The stack S' matches A, and is returned unchanged because in $(\!|\underline{w}|\!)$ there are η-expansions of that type, acting as the identity on stacks.

The remaining details may be understood by looking at how the bits of w are represented in $(\!|\underline{w}|\!)$, but this is not really essential. What is important is to observe that predicting the behavior of $\mathsf{IAM}((\!|\underline{w}|\!))$ only requires inspecting w and updating counters bounded by $|w|$, which is all doable in logarithmic space.

6 Discussion and Perspectives

As mentioned in the introduction, we believe that our system **PL** gives the simplest functional characterization of L currently known. We also want to stress that parsimony offers a truly novel approach to applying linear logic to ICC, which is not just a variant of existing "light logics" (such as bounded, light or soft linear logic) or of systems such as those of [13, 14]. The most prominent difference with respect to "light logics" is the absence of stratification or other structural principles enforcing bounded-time cut-elimination: as mentioned above, the untyped parsimonious λ-calculus is Turing-complete, whereas light λ-calculi normalize with the same runtime independently of types. This is because parsimony is not about the global complexity of normalization but the local complexity of single reduction steps, via the notion of continuous linear approximations originally introduced in [19]. This allows dealing with non-uniform computation [20, 21], a perspective not offered by previous work on ICC.

If we add to **PL** the constant $\perp$ (typable with all types), we obtain *finitary terms* as terms whose boxes are all of the form $!\perp$. Essentially, these are purely multiplicative affine terms, or multiplicative proof nets: their size bounds the number of steps to normal form and they may be related to Boolean circuits [26]. A parsimonious term t induces a family of finitary approximations $(\lfloor t \rfloor_n)_{n\in\mathbb{N}}$: $\lfloor t \rfloor_n$ is defined by taking t and truncating all the streams appearing in it to length n ("truncating" means replacing the tail of the stream with $!\perp$). We know from [19] that reduction is continuous w.r.t. these approximations. Parsimony refines this by giving a polynomial "modulus of continuity" [20]: if $t\,\underline{w} \to^* \mathsf{b}$ with b a Boolean value, then there exists m polynomial in $|w|$ s.t. $\lfloor t \rfloor_m\,\underline{w} \to^* \mathsf{b}$, *i.e.*, a polynomial-size approximation of t is sufficient to compute the result.

Now, an arbitrary family of simply-typed finitary terms $(u_n)_{n\in\mathbb{N}} : \mathsf{Str}[] \multimap \mathsf{Bool}$ decides a language L in the same sense as a family of circuits. It is shown in [21] that, if the size of u_n is polynomial in n, then $L \in \mathsf{L/poly}$ (and conversely). If the u_n happen to be approximations of a generic **PL** term $t : \mathsf{Str}[] \multimap \mathsf{Bool}$, the family is uniform, and indeed we proved here that $L \in \mathsf{L}$. But how uniform is it? We can show that it is at least logspace-uniform, but we suspect the uniformity to be stronger (*e.g.* $\mathrm{U_E}$-uniform) and plan to investigate further on this.

Another interesting research direction is to consider second-order quantification, *i.e.*, parsimonious system **F**. In [21], it is shown that *linear* polymorphism (*i.e.*, comprehension restricted to !-free formulas) yields $\mathsf{P/poly}$ (non-uniform polynomial time). In the uniform case, we should of course obtain P, whereas we conjecture that the full parsimonious system **F** captures exactly primitive recursion.

Acknowledgments. The present formulation of this work owes much to discussions with Kazushige Terui, whom we wish to warmly thank here. We also acknowledge partial support

of ANR projects Logoi ANR-2010-BLAN-0213-02, Coquas ANR-12-JS02-006-01 and Elica ANR-14-CE25-0005.

References

1. Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In *Proceedings of CSL-LICS*, page 8, 2014.
2. Patrick Baillot and Damiano Mazza. Linear logic by levels and bounded time complexity. *Theor. Comput. Sci.*, 411(2):470–503, 2010.
3. Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
4. Guillaume Bonfante. Some programming languages for logspace and ptime. In *Proceedings of AMAST*, pages 66–80, 2006.
5. Ugo Dal Lago. Context semantics, linear logic, and computational complexity. *ACM Trans. Comput. Log.*, 10(4), 2009.
6. Ugo Dal Lago and Ulrich Schöpp. Functional programming in sublinear space. In *Proceedings of ESOP*, pages 205–225, 2010.
7. Ugo Dal Lago and Ulrich Schöpp. Type inference for sublinear space functional programming. In *Proceedings of APLAS*, pages 376–391, 2010.
8. Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal lambda-machines. *Theor. Comput. Sci.*, 227(1-2):79–97, 1999.
9. Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An implicit characterization of PSPACE. *ACM Trans. Comput. Log.*, 13(2):18, 2012.
10. Marco Gaboardi, Luca Roversi, and Luca Vercelli. A by-level analysis of multiplicative exponential linear logic. In *Proceedings of MFCS*, pages 344–355, 2009.
11. Jean-Yves Girard. Geometry of interaction I: Interpretation of system F. In *Proccedings of Logic Colloquium 1988*, pages 221–260, 1989.
12. Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.
13. Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Proceedings of CSL*, pages 275–294, 1997.
14. Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003.
15. Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.
16. Neil D. Jones. Logspace and ptime characterized by programming languages. *Theor. Comput. Sci.*, 228(1-2):151–174, 1999.
17. Lars Kristiansen. Neat function algebraic characterizations of logspace and linspace. *Computational Complexity*, 14(1):72–88, 2005.
18. Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2), 1993.
19. Damiano Mazza. An infinitary affine lambda-calculus isomorphic to the full lambda-calculus. In *Proceedings of LICS*, pages 471–480, 2012.
20. Damiano Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *Proceedings of ICALP, Part II*, pages 305–317, 2014.
21. Damiano Mazza and Kazushige Terui. Parsimonious types and non-uniform computation. In *Proceedings of ICALP, Part II*, pages 350–361, 2015.
22. Peter Møller Neergaard. A functional language for logarithmic space. In *Proceedings of APLAS*, pages 311–326, 2004.
23. Ramyaa Ramyaa and Daniel Leivant. Ramified corecurrence and logspace. *Electr. Notes Theor. Comput. Sci.*, 276:247–261, 2011.

24 Ulrich Schöpp. Space-efficient computation by interaction. In *Proceedings of CSL*, pages 606–621, 2006.

25 Ulrich Schöpp. Stratified bounded affine logic for logarithmic space. In *Proceedings of LICS*, pages 411–420, 2007.

26 Kazushige Terui. Proof nets and boolean circuits. In *Proceedings of LICS*, pages 182–191, 2004.