

Logic Programming and Logarithmic Space

Clément Aubert¹, Marc Bagnol¹, Paolo Pistone¹, and Thomas Seiller² *

¹ Aix Marseille Université, CNRS, I2M UMR 7373, 13453, Marseille, France

² I.H.É.S., Le Bois-Marie, 35, Route de Chartres, 91440 Bures-sur-Yvette, France

Abstract. We present an algebraic view on logic programming, related to proof theory and more specifically linear logic and geometry of interaction. Within this construction, a characterization of logspace (deterministic and non-deterministic) computation is given *via* a syntactic restriction, using an encoding of words that derives from proof theory.

We show that the acceptance of a word by an observation (the counterpart of a program in the encoding) can be decided within logarithmic space, by reducing this problem to the acyclicity of a graph. We show moreover that observations are as expressive as two-ways multi-heads finite automata, a kind of pointer machines that is a standard model of logarithmic space computation.

Keywords: Implicit Complexity, Unification, Logic Programming, Logarithmic Space, Proof Theory, Pointer Machines, Geometry of Interaction, Automata

1 Introduction

Proof Theory and Implicit Complexity Theory Very generally, the aim of implicit complexity theory (ICC) is to describe complexity classes with no explicit reference to cost bounds: through a type system or a weakened recursion scheme for instance. The last two decades have seen numerous works relating proof theory (more specifically linear logic [14]) and ICC, the basic idea being to look for restricted substructural logics [18] with an expressiveness that corresponds exactly to some complexity class.

This has been achieved by various syntactic restrictions, which entail a less complex (any function provably total in second-order Peano Arithmetic [14] can be encoded in second-order linear logic) cut-elimination procedure: control over the modalities [29,9], type assignments [13] or stratification properties [5], to name a few.

Geometry of Interaction Over the recent years, the cut-elimination procedure and its mathematical modeling has become a central topic in proof theory. The aim of the geometry of interaction research program [15] is to provide the tools for such a modeling [1,24,30].

*This work was partly supported by the ANR-10-BLAN-0213 Logoi and the ANR-11-BS02-0010 Récré.

As for complexity theory, these models allow for a more synthetic and abstract study of the resources needed to compute the normal form of a program, leading to some complexity characterization results [6,19,2].

Unification The unification technique is one of the key-concepts of theoretical computer science: it is a classical subject of study for complexity theory and a tool with a wide range of applications, including logic programming and type inference algorithms.

Unification has also been used to build syntactic models of geometry of interaction [17,6,20] where first-order terms with variables allow for a manipulation of infinite sets through a finite language.

Logic Programming After the work of Robinson on the resolution procedure, logic programming has emerged as a new computation paradigm with concrete realizations such as the languages PROLOG and DATALOG.

On the theoretical side, a lot of efforts has been made to clarify expressiveness and complexity issues [10]: most problems arising from logic programming are undecidable in their most general form and some restrictions must be introduced in order to make them tractable. For instance, the notion of *finitely ground program* [8] is related to our approach.

Pointer Machines Multi-heads finite automata provide an elegant characterization of logspace computation, in terms of the (qualitative) type of memory used rather than the (quantitative) amount of tape consumed. Since they can scan but not modify the input, they are usually called “pointer machines”.

This model was already at the heart of previous works relating geometry of interaction and complexity theory [19,3,2].

Contribution and Outline We begin by exposing the idea of relating geometry of interaction and logic programming, already evoked [17] but never really developed, and by recalling the basic notions on unification theory needed for this article and some related complexity results.

We present in Sect. 2 the algebraic tools used later on to define the encoding of words and pointer machines. Section 2.2 and Sect. 2.3 introduce the syntactical restriction and associated tools that allow us to characterize logarithmic space computation. Note that, compared to earlier work [2], we consider a much wider class of programs while preserving bounded space evaluation.

The encoding of words enabling our results, which comes from the classical (Church) encoding of lists in proof theory, is given in Sect. 3. It allows to define the counterpart of programs, and a notion of acceptance of a word by a program.

Finally, Sect. 4 makes use of the tools introduced earlier to state and prove our complexity results. While the expressiveness part is quite similar to earlier presentations [3,2], the proof that acceptance can be decided within logarithmic space has been made more modular by reducing it to cycle search in a graph.

1.1 Geometry of Interaction and Logic Programming

The geometry of interaction program (GOI for short), started in 1989 [16], aims at describing the dynamics of computation by developing a fully mathematical model of cut-elimination. The original motivations of GOI must be traced back, firstly, to the *Curry-Howard correspondence* between sequent calculus derivations and typed functional programs: it is on the basis of this correspondence that cut-elimination had been proposed by proof-theorists as a paradigm of computation; secondly, to the finer analysis of cut-elimination coming from linear logic [14] and the replacement of sequent calculus derivations with simpler geometrical structures (proof-nets), more akin to a purely mathematical description.

In the first formulation of GOI [15], derivations in second order intuitionistic logic LJ^2 (which can be considered, by *Curry-Howard*, as programs in System F) are interpreted as pairs (U, σ) of elements (called *wirings*) of a \mathbb{C}^* -algebra, U corresponding to the axioms of the derivation and σ to the cuts.

The main property of this interpretation is *nilpotency*: there exists an integer n such that $(\sigma U)^n = 0$. The cut-elimination (equivalently, the normalization) procedure is then interpreted by the application of an *execution operator*

$$EX(U, \sigma) = \sum_k (\sigma U)^k$$

From the viewpoint of proof theory and computation, nilpotency corresponds to the *strong normalization property*: the termination of the normalization procedure with any strategy.

Several alternative formulations of geometry of interaction have been proposed since 1988 (see for instance [1,24,30]); in particular, wirings can be described as logic programs [17,6,20] made of particular clauses called *flows*, which will be defined in Sect. 2.1.

In this setting the resolution rule induces a notion of product of wirings (Definition 8) and in turn a structure of semiring: the *unification semiring* \mathcal{U} , which can replace the \mathbb{C}^* -algebras of the first formulations of GOI.³

The $EX(\cdot)$ operator of wirings can be understood as a way to compute the fixed point semantics of logic programs. The nilpotency property of wirings means then that the fixed point given by $EX(\cdot)$ is finite, which is close to the notion of *boundedness* [10] of logic programs.

In definitive, from the strong normalization property for intuitionistic second order logic (or any other system which enjoys a GOI interpretation), one obtains through the GOI interpretation a family of bounded (nilpotent) logic programs computing the recursive functions typable in System F.

This is quite striking in view of the fact that to decide whenever a program is *bounded*⁴ is – even with drastic constraints – an undecidable problem [21], and that in general boundedness is a property that is difficult to ensure.

³By adding complex scalar coefficients, one can actually extend \mathcal{U} into a \mathbb{C}^* -algebra [17].

⁴A program is *bounded* if there is an integer k such that the fixed point computation of the program is stable after k iterations, independently of the facts inputed.

1.2 Unification and Complexity

Unification is a classical subject of research, at the intersection of practical and theoretical considerations. We recall in the following some notations and some of the numerous links between complexity and unification, and by extension logic programming.

Notations. We consider a set of first-order terms \mathbf{T} , assuming an infinite number of variables $x, y, z, \dots \in \mathbf{V}$, a binary function symbol \bullet (written in *infix notation*), infinitely many constant symbols $\mathbf{a}, \mathbf{b}, \mathbf{c}, \dots$ including the (multipurpose) dummy symbol \star and, for any $n \in \mathbb{N}^*$, at least one n -ary function symbol \mathbf{A}_n .

Note that the binary function symbol \bullet is not associative. However, we will write it by convention as *right associating* to lighten notations: $t \bullet u \bullet v := t \bullet (u \bullet v)$.

For any $t \in \mathbf{T}$, we write $\mathbf{Var}(t)$ the set of variables occurring in t (a term is *closed* when $\mathbf{Var}(t) = \emptyset$) and $\mathbf{h}(t)$ the *height* of t : the maximal distance from the root to any other subterm in the tree structure of t .

The *height of a variable occurrence* in a term t is its distance from the root in the tree structure of the term. A *substitution* θ is a mapping from variables to terms such that $x\theta = x$ for all but finitely many $x \in \mathbf{V}$. A *renaming* is a substitution α mapping variables to variables and that is bijective. A term t' is a *renaming* of t if $t' = t\alpha$ for some renaming α .

Definition 1 (unification, matching and disjointness). *Two terms t, u are*

- *unifiable if there exists a substitution⁵ θ such that $t\theta = u\theta$,*
- *matchable if t', u' are unifiable, where t', u' are renamings of t, u such that $\mathbf{Var}(t') \cap \mathbf{Var}(u') = \emptyset$,*
- *disjoint if they are not matchable.*

It is well-known that the problem of deciding whether two terms are unifiable is PTIME-complete [11, Theorem 1], which implies that parallel algorithms for this problem do not improve much on serial ones. Finding classes of terms where the MGU research can be efficiently parallelized is a real challenge.

It has been proven that this problem remains PTIME-complete even if the arity of the function symbols or the height of the terms is bounded [26, Theorems 4.2.1 and 4.3.1], if both terms are linear or if they do not share variables [11, 12]. More recently [7], an innovative constraint on variables helped to discover an upper bound of the unification classes that are proven to be in NC.

Regarding space complexity, the result stating that the *matching problem* is in DLOGSPACE [11] (recalled as **Theorem 35**) will be used in **Sect. 4.2**.

2 The Unification Semiring

This section presents the technical setting of this work, the *unification semiring*: an algebraic structure with a composition law based on unification, that can be seen as an algebraic presentation of a fragment of logic programming.

⁵Called a *unifier*. Remember that two terms t, u that are unifiable have a *most general unifier* (MGU): a unifier θ such that any other unifier of t, u is an instance of θ .

2.1 Flows and Wirings

Flows can be thought of as very specific Horn clauses: safe (the variables of the head must occur in the body) clauses with exactly one atom in the body.

As it is not relevant to this work, we make no difference between predicate symbols and function symbols, for it makes the presentation easier.

Definition 2 (flows). A flow is a pair of terms $t \leftarrow u$ with $\text{Var}(t) \subseteq \text{Var}(u)$. Flows are considered up to renaming: for any renaming α , $t \leftarrow u = t\alpha \leftarrow u\alpha$.

Facts, that are usually defined as ground (using only closed terms) clauses with an empty body, can still be represented as a special kind of flows.

Definition 3 (facts). A fact is a flow of the form $t \leftarrow \star$.

Remark 4. Note that this implies that t is closed.

The main interest of the restriction to flows is that it yields an algebraic structure: a semigroup with a partially defined product.

Definition 5 (product of flows). Let $u \leftarrow v$ and $t \leftarrow w$ be two flows. Suppose we have representatives of the renaming classes such that $\text{Var}(v) \cap \text{Var}(w) = \emptyset$. The product of $u \leftarrow v$ and $t \leftarrow w$ is defined if v, t are unifiable with MGU θ as $(u \leftarrow v)(t \leftarrow w) := u\theta \leftarrow w\theta$.

Remark 6. The condition on variables ensures that facts form a “left ideal” of the set of flows: if \mathbf{u} is a fact and f a flow, then $f\mathbf{u}$ is a fact when it is defined.

Example 7.

$$\begin{aligned} (\mathbf{f}(x) \leftarrow x)(\mathbf{f}(x) \leftarrow \mathbf{g}(x)) &= \mathbf{f}(\mathbf{f}(x)) \leftarrow \mathbf{g}(x) \\ (x \cdot \mathbf{c} \leftarrow (y \cdot y) \cdot x)((\mathbf{c} \cdot \mathbf{c}) \cdot x \leftarrow y \cdot x) &= x \cdot \mathbf{c} \leftarrow \mathbf{c} \cdot x \\ (\mathbf{f}(x \cdot c) \leftarrow x \cdot \mathbf{d})(\mathbf{d} \cdot \mathbf{d} \leftarrow \star) &= \mathbf{f}(\mathbf{d} \cdot \mathbf{c}) \leftarrow \star \end{aligned}$$

The product of flows corresponds to the resolution rule in the following sense: given two flows $f = u \leftarrow v$ and $g = t \leftarrow w$ and a MGU θ of v and t , then the resolution rule applied to f and g would yield fg .

Wirings then correspond to logic programs (sets of clauses) and the nilpotency condition can be seen as an algebraic variant of the notion of boundedness of these programs.

Definition 8 (wirings). Wirings are finite sets of flows. The product of wirings is defined as $FG := \{ fg \mid f \in F, g \in G, fg \text{ defined} \}$.

We write \mathcal{U} the set of wirings and refer to it as the unification semiring.

The set of wirings \mathcal{U} has a structure of semiring. We use an *additive notation* for sets of flows to stress this point:

- The symbol $+$ will be used in place of \cup .
- We write sets as the sum of their elements: $\{ f_1, \dots, f_n \} := f_1 + \dots + f_n$.
- We write 0 for the empty set.

- The unit is $I := x \leftarrow x$.

We will call *semiring* any subset \mathcal{A} of \mathcal{U} such that

- $0 \in \mathcal{A}$,
- if $F_1, \dots, F_n \in \mathcal{A}$, then $F_1 + \dots + F_n \in \mathcal{A}$,
- if $F \in \mathcal{A}$ and $G \in \mathcal{A}$ then $FG \in \mathcal{A}$.

Definition 9 (nilpotency). A wiring F is nilpotent if $F^n = 0$ for some $n \in \mathbb{N}$. We may use the notation $\text{Nil}(F)$ to express the fact that F is nilpotent.

As mentioned in [Sect. 1.1](#), nilpotency is related with the notion of *boundedness* [10] of a logic program. Indeed, if we have a wiring F and a finite set of facts \mathbf{U} , let us consider the set of facts that can be obtained through F $\{\mathbf{u} \mid \mathbf{u} \in F^n \mathbf{U} \text{ for some } n\}$ which can also be written as $(I + F + F^2 + \dots) \mathbf{U}$ or $EX(F) \mathbf{U}$ (where $EX(\cdot)$ is the execution operator of [Sect. 1.1](#)).

If F is nilpotent, one needs to compute the sum only up to a finite rank that does not depend on \mathbf{U} , which implies the boundedness property.

Among wirings, those that can produce at most one fact from any fact will be of interest when considering deterministic *vs.* non-deterministic computation.

Definition 10 (deterministic wirings). A wiring F is deterministic if given any fact \mathbf{u} , $\text{card}(F\mathbf{u}) \leq 1$. We will write \mathcal{U}_d the set of deterministic wirings.

It is clear from the definition that \mathcal{U}_d forms a semiring. The lemma below gives us a class of wirings that are deterministic and easy to recognize, due to its more syntactic definition.

Lemma 11. Let $F = \sum_i u_i \leftarrow t_i$. If the t_i are pairwise disjoint ([Definition 1](#)), then F is deterministic.

Proof. Given a closed term t there is at most one of the t_i that matches t , therefore $F(t \leftarrow \star)$ is either a single fact or 0. \square

2.2 The Balanced Semiring

In this section, we study a constraint on variable height of flows which we call *balance*. This syntactic constraint can be compared with similar ones proposed in order to get logic programs that are *finitely ground* [8]: balanced wirings are a special case of *argument-restricted* programs in the sense of [25].

Balanced wirings will enjoy properties (see [Sect. 2.3](#)) that will allow to decide their nilpotency efficiently.

Definition 12 (balance). A flow $f = t \leftarrow u$ is balanced if for any variable $x \in \text{Var}(t) \cup \text{Var}(u)$, all occurrences of x in either t or u have the same height (recall notations [p. 4](#)) which we write $\text{h}_f(x)$, the height of x in f . A wiring F is balanced if it is a sum of balanced flows.

We write \mathcal{U}_b the set of balanced wirings and refer to it as the balanced semiring.

Note that in [Example 7](#), only the second line shows product of balanced flows.

Definition 13 (height). *The height $\mathbf{h}(f)$ of a flow $f = t \leftarrow u$ is $\max\{\mathbf{h}(t), \mathbf{h}(u)\}$. The height $\mathbf{h}(F)$ of a wiring F is the maximal height of flows in it.*

The following lemma summarizes the properties that are preserved by the product of balanced flows. It implies in particular that \mathcal{U}_b is indeed a semiring.

Lemma 14. *When it is defined, the product fg of two balanced flows f and g is still balanced and its height is at most $\max\{\mathbf{h}(f), \mathbf{h}(g)\}$.*

Proof (sketch). By showing that the variable height condition and the global height are both preserved by the basic steps of the unification procedure. \square

2.3 The Computation Graph

The main tool for a space-efficient treatment of balanced wirings is an associated notion of graph. This section focuses on the algebraic aspects of this notion, proving various technical lemmas, and leaves the complexity issues to [Sect. 4.2](#).

A separating space can be thought of as a finite subset of the Herbrand universe associated with a logic program, containing enough information to decide the problem at hand.

Definition 15 (separating space). *A separating space for a wiring F is a set of facts \mathbf{S} such that*

- *For all $\mathbf{u} \in \mathbf{S}$, $F\mathbf{u} \subseteq \mathbf{S}$.*
- *$F^n\mathbf{u} = 0$ for all $\mathbf{u} \in \mathbf{S}$ implies $F^n = 0$.*

We can define such a space for balanced wirings with [Lemma 14](#) in mind: balanced wirings behave well with respect to height of terms.

Definition 16 (computation space). *Given a balanced wiring F , we define its computation space $\mathbf{Comp}(F)$ as the set of facts of height at most $\mathbf{h}(F)$, built using only the symbols appearing in F and the constant symbol \star .*

Lemma 17 (separation). *If F is balanced, then $\mathbf{Comp}(F)$ is separating for F .*

Proof. By [Lemma 14](#), $F(u \leftarrow \star)$ is of height at most $\max\{\mathbf{h}(F), \mathbf{h}(u)\} \leq \mathbf{h}(F)$ and it contains only symbols occurring in F and u , therefore if $\mathbf{u} \in \mathbf{Comp}(F)$ we have $F\mathbf{u} \subseteq \mathbf{Comp}(F)$.

By [Lemma 14](#) again, F^n is still of height at most $\mathbf{h}(F)$. If $(F^n)\mathbf{u} = 0$ for all $\mathbf{u} \in \mathbf{Comp}(F)$, it means the flows of F^n do not match any closed term of height at most $\mathbf{h}(F)$ built with the symbols occurring in F (and eventually \star). This is only possible if F contains no flow, *ie.* $F = 0$. \square

As F is a finite set, thus built with finitely many symbols, $\mathbf{Comp}(F)$ is also a finite set. We can be a little more precise and give a bound to its cardinal.

Proposition 18 (cardinality). *Let F be a balanced wiring, A the maximal arity of function symbols occurring in F and S the set of symbols occurring in F , then $\text{card}(\mathbf{Comp}(F)) \leq (\text{card}(S) + 1)^{P_{\mathbf{h}(F)}(A)}$, where $P_k(X) = 1 + X + \dots + X^k$.*

Proof. The number of terms of height $\mathbf{h}(F)$ built over the set of symbols $S \cup \{\star\}$ of arity bounded by A is at most as large as the number of complete trees of degree A and height $\mathbf{h}(F)$ (that is, trees where nodes of height less than $\mathbf{h}(F)$ have exactly A childs), with nodes labeled by elements of $S \cup \{\star\}$. \square

Then, we can encode in a directed graph⁶ the action of the wiring on its computation space.

Definition 19 (computation graph). *If F is a balanced wiring, we define its computation graph $\mathbf{G}(F)$ as the directed graph:*

- *The vertices of $\mathbf{G}(F)$ are the elements of $\mathbf{Comp}(F)$.*
- *There is an edge from \mathbf{u} to \mathbf{v} in $\mathbf{G}(F)$ if $\mathbf{v} \in F\mathbf{u}$.*

We state finally that the computation graph of a wiring contains enough information on the latter to determine its nilpotency. This is a key ingredient in the proof of [Theorem 34](#), as the research of paths and cycles in graphs are problems that are well-known to be solvable within logarithmic space.

Lemma 20. *A balanced wiring F is nilpotent ([Definition 9](#)) iff $\mathbf{G}(F)$ is acyclic.*

Proof. Suppose there is a cycle of length n in $\mathbf{G}(F)$, and let \mathbf{u} be the label of a vertex which is part of this cycle. By definition of $\mathbf{G}(F)$, $\mathbf{u} \in (F^n)^k \mathbf{u}$ for all k , which means that $(F^n)^k \neq 0$ for all k and therefore F cannot be nilpotent.

Conversely, suppose there is no cycle in $\mathbf{G}(F)$. As it is a finite graph, this entails a maximal length N of paths in $\mathbf{G}(F)$. By definition of $\mathbf{G}(F)$, this means that $F^{N+1} \mathbf{u} = 0$ for all $\mathbf{u} \in \mathbf{Comp}(F)$ and with [Lemma 17](#) we get $F^{N+1} = 0$. \square

Moreover, the computation graph of a deterministic ([Definition 10](#)) wiring has a specific shape, which in turn induces a deterministic procedure in this case.

Lemma 21. *If F is a balanced and deterministic wiring, $\mathbf{G}(F)$ has an out-degree (the maximal number of edges a vertex can be the source of) bounded by 1.*

Proof. It is a direct consequence of the definitions of $\mathbf{G}(F)$ and determinism. \square

2.4 Tensor product and other semirings

Finally, we list a few other semirings that will be used in the next section, where we define the notions of representation of a word and observation.

The binary function symbol \bullet can be used to define an operation that is similar to the algebraic notion of tensor product.

⁶Here by directed graph we mean a set of *vertices* V together with a set of *edges* $E \subseteq V \times V$. We say that there is an edge *from* $e \in V$ *to* $f \in V$ when $(e, f) \in E$.

Definition 22 (tensor product). Let $u \leftarrow v$ and $t \leftarrow w$ be two flows. Suppose we have chosen representatives of their renaming classes that have disjoint sets of variables. We define their tensor product as $(u \leftarrow v) \dot{\otimes} (t \leftarrow w) := u \bullet t \leftarrow v \bullet w$. The operation is extended to wirings by $(\sum_i f_i) \dot{\otimes} (\sum_j g_j) := \sum_{i,j} f_i \dot{\otimes} g_j$. Given two semirings \mathcal{A}, \mathcal{B} , we define $\mathcal{A} \dot{\otimes} \mathcal{B} := \{ \sum_i F_i \dot{\otimes} G_i \mid F_i \in \mathcal{A}, G_i \in \mathcal{B} \}$.

The tensor product of two semirings is easily shown to be a semiring.

Example 23.

$$(\mathbf{f}(x) \bullet y \leftarrow y \bullet x) \dot{\otimes} (x \leftarrow \mathbf{g}(x)) = (\mathbf{f}(x) \bullet y) \bullet z \leftarrow (y \bullet x) \bullet \mathbf{g}(z)$$

Notation. As the symbol \bullet , the $\dot{\otimes}$ operation is not associative. We carry on the convention for \bullet and write it as *right associating*: $\mathcal{A} \dot{\otimes} \mathcal{B} \dot{\otimes} \mathcal{C} := \mathcal{A} \dot{\otimes} (\mathcal{B} \dot{\otimes} \mathcal{C})$.

Semirings can also naturally be associated to any set of closed terms or to the restriction to a certain set of symbols.

Definition 24. Given a set of closed terms E , we define the following semiring $E^\leftarrow := \{ \sum_i t_i \leftarrow u_i \mid t_i, u_i \in E \}$. If \mathbf{S} is a set of symbols and \mathcal{A} a semiring, we write $\mathcal{A}^{\mathbf{S}}$ the semiring of wirings of \mathcal{A} , that do not use the symbols in \mathbf{S} .

This operation yields semirings because composition of flows made of closed terms involves no actual unification: it is just equality of terms and therefore one never steps out of E^\leftarrow .

Finally, the unit $I = x \leftarrow x$ of \mathcal{U} yields a semiring.

Definition 25 (unit semiring). We call unit semiring the semiring $\mathcal{I} := \{ I \}$.

3 Words and Observations

We define in this section the global framework that will be used later on to obtain the characterization of logarithmic space computation. In order to discuss the contents of this section, let us first define two specific semirings.

Definition 26 (word and observation semirings). We fix two (disjoint) infinite sets of constant symbols \mathbf{P} and \mathbf{S} , and a unary function symbol \mathbf{M} . We denote by $\mathbf{M}(\mathbf{P})$ the set of terms $\mathbf{M}(\mathbf{p})$ with $\mathbf{p} \in \mathbf{P}$. We define the following two semirings that are included in \mathcal{U}_b :

- The word semiring is the semiring $\mathcal{W} := \mathcal{I} \dot{\otimes} \mathcal{I} \dot{\otimes} \mathbf{M}(\mathbf{P})^\leftarrow$.
- The observation semiring is the semiring $\mathcal{O} := \mathbf{S}^\leftarrow \dot{\otimes} \mathcal{U}_b \setminus \mathbf{P}$.

These two semirings will be used as parameters of a construction $\mathcal{M}_\Sigma(\cdot)$ to define the representation of words and a notion of abstract machine, that we shall call observations, on an alphabet Σ (we suppose $\star \notin \Sigma$).

Definition 27. We fix the set of constant symbols $\mathbf{LR} := \{\mathbf{L}, \mathbf{R}\}$.

Given a set of constant symbols Σ and a semiring \mathcal{A} we define the semiring $\mathcal{M}_\Sigma(\mathcal{A}) := (\Sigma \cup \{\star\})^\leftarrow \dot{\otimes} \mathbf{LR}^\leftarrow \dot{\otimes} \mathcal{A}$.

In the following of this section, we will show how to represent lists of elements of Σ by wirings in the semiring $\mathcal{M}_\Sigma(\mathcal{W})$. Then, we will explain how the semiring $\mathcal{M}_\Sigma(\mathcal{O})$ captures a notion of abstract machine. In the last section of the paper we will explain further how observations and words interact, and prove that this interaction captures logarithmic space computation.

3.1 Representation of Words

We now show how one can represent words by wirings in $\mathcal{M}_\Sigma(\mathcal{W})$. We recall this semiring is defined as $((\Sigma \cup \{\star\})^\leftarrow \dot{\otimes} \text{LR}^\leftarrow) \dot{\otimes} \mathcal{I} \dot{\otimes} \mathcal{I} \dot{\otimes} \mathbf{M}(\mathbf{P})^\leftarrow$.

The part $(\Sigma \cup \{\star\})^\leftarrow \dot{\otimes} \text{LR}^\leftarrow$ deals with, and is dependent on, the alphabet Σ considered; this is where the integer and the observation will interact. The two instances of the unit semiring \mathcal{I} correspond to the fact that the word cannot affect parts of the observation that correspond to internal configurations. The last part, namely the semiring $\mathbf{M}(\mathbf{P})^\leftarrow$, will contain the *position constants* of the representation of words.

Notation. We write $t \rightleftharpoons u$ for $t \leftarrow u + u \leftarrow t$.

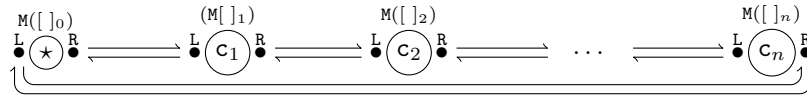
Definition 28 (word representations). Let $W = c_1 \dots c_n$ be a word over an alphabet Σ and $p = p_0, p_1, \dots, p_n$ be pairwise distinct constant symbols.

Writing $p_{n+1} = p_0$ and $c_0 = c_{n+1} = \star$, we define the representation of W associated with p_0, p_1, \dots, p_n as the following wiring:

$$\bar{W}_p = \sum_{i=0}^n c_i \bullet \text{L} \bullet x \bullet y \bullet \mathbf{M}(p_i) \rightleftharpoons c_{i+1} \bullet \text{R} \bullet x \bullet y \bullet \mathbf{M}(p_{i+1}) \quad (1)$$

We will write $\mathcal{R}(W)$ the set of representations of a given word W .

To understand better this representation, consider that each symbol in the alphabet Σ comes in two “flavors”, *left* and *right*. Then, one can easily construct the “context” $\bar{W} = \sum_{i=0}^n c_i \bullet \text{L} \bullet x \bullet y \bullet \mathbf{M}([]_i) \rightleftharpoons c_{i+1} \bullet \text{R} \bullet x \bullet y \bullet \mathbf{M}([]_{i+1})$ from the list as the sums of the arrows in the following picture (where x and y are omitted):



Then, choosing a set $p = p_0, \dots, p_n$ of position constants, intuitively representing physical memory addresses, the representation \bar{W}_p of a word associated with p is obtained by filling, for all $i = 0, \dots, n$, the hole $[]_i$ by the constant p_i .

This abstract representation of words is not an arbitrary choice. It comes from the interpretation of lists in geometry of interaction.

Indeed, in System F, the type of binary lists corresponds to the formula $\forall X (X \Rightarrow X) \Rightarrow (X \Rightarrow X) \Rightarrow (X \Rightarrow X)$. Any lambda-term in normal form of this type can be written as $\lambda f_0 f_1 x. f_{c_1} f_{c_2} \dots f_{c_k} x$, where $c_1 \dots c_k$ is a word on $\{0, 1\}$. The GOI representation of such a lambda-term yields the abstract

representation just defined.⁷ Notice that the additional symbol \star used to represent words corresponds to the variable x in the preceding lambda-term. Note also the fact that the representation of integer is *cyclic*, and that the symbol \star serves as a reference for the starting/ending point of the word.

Let us finally stress that the words are represented as *deterministic* wirings. This implies that the restriction to deterministic observations will correspond to restricting ourselves to deterministic pointer machines. The framework, however, allows for a number of generalization and variants. For instance, one can define a representation of trees by adapting Definition 28 in such a way that every vertex is related to its descendants; doing so would however yield non-deterministic wirings. In the same spirit, a notion of “one-way representations of words”, defined by replacing the symbol \leftrightsquigarrow by the symbol \leftarrow in (1) of Definition 28, could be used to characterize one-way multi-heads automata.

3.2 Observations

We now define *observations*. We will then explain how these can be thought of as a kind of abstract machines. An observation is an element of the semiring

$$(\Sigma \cup \{\star\})^{\leftarrow} \dot{\otimes} \mathbf{LR}^{\leftarrow} \dot{\otimes} (\mathbf{S}^{\leftarrow} \dot{\otimes} \mathcal{U}_b^{\setminus \mathbf{P}})$$

Once again, the part of the semiring $(\Sigma \cup \{\star\})^{\leftarrow} \dot{\otimes} \mathbf{LR}^{\leftarrow}$ is dependent on the alphabet Σ considered and represents the point of interaction between the words and the machine. The semiring \mathbf{S}^{\leftarrow} intuitively corresponds to the *states* of the observation, while the part $\mathcal{U}_b^{\setminus \mathbf{P}}$ forbids the machine to act non-trivially on the *position constant* of the representation of words. The fact that the machine cannot perform any operation on the memory addresses – the position constants – of the word representation explains why observations are naturally thought of as a kind of *pointer machines*.

Definition 29 (observation). *An observation is any element O of $\mathcal{M}_\Sigma(\mathcal{O})$.*

We can define the language associated to an observation. The condition of acceptance will be represented as the nilpotency of the product $O\bar{W}_p$ where $\bar{W}_p \in \mathcal{R}(W)$ represents a word W and O is an observation.

Definition 30 (language of an observation). *Let O be an observation on the alphabet Σ . We define the language accepted by O as*

$$\mathcal{L}(O) := \{ W \in \Sigma^* \mid \forall p, \mathbf{Nil}(O\bar{W}_p) \}$$

One important point is that the semirings $\mathcal{M}_\Sigma(\mathcal{W})$ and $\mathcal{M}_\Sigma(\mathcal{O})$ are not completely disjoint, and therefore allow for non-trivial interaction of observations and words. However, they are sufficiently disjoint so that this computation does not depend on the choice of the representative of a given word.

⁷A thorough explanation can be found in previous work by Aubert and Seiller [3].

Lemma 31. *Let W be a word, and $\bar{W}_p, \bar{W}_q \in \mathcal{R}(W)$. For every observation $O \in \mathcal{M}_\Sigma(\mathcal{O})$, $\mathbf{Nil}(O\bar{W}_p)$ if and only if $\mathbf{Nil}(O\bar{W}_q)$.*

Proof. As we pointed out, the observation cannot act on the position constants of the representations \bar{W}_p and \bar{W}_q . This implies that for all integer k the wirings $(O\bar{W}_p)^k$ and $(O\bar{W}_q)^k$ are two instances of the same *context*, i.e. they are equal up to the interchange of the positions constants $\mathbf{p}_0, \dots, \mathbf{p}_n$ and $\mathbf{q}_0, \dots, \mathbf{q}_n$. This implies that $(O\bar{W}_p)^k = 0$ if and only if $(O\bar{W}_q)^k = 0$. \square

Corollary 32. *Let O be an observation on the alphabet Σ . The set $\mathcal{L}(O)$ can be equivalently defined as the set*

$$\mathcal{L}(O) = \{ W \in \Sigma^* \mid \exists p, \mathbf{Nil}(O\bar{W}_p) \}$$

This result implies that the notion of acceptance has the intended sense and is finitely verifiable: whether a word W is accepted by an observation O can be checked without considering all representations of W .

This kind of situation where two semirings \mathcal{W} and \mathcal{O} are disjoint enough to obtain [Corollary 32](#) can be formalized through the notion of *normative pair* considered in earlier works [\[19,3,2\]](#).

4 Logarithmic Space

This section starts by explaining the computation one can perform with the observations, and prove that it corresponds to logarithmic space computation by showing how pointer machines can be simulated. Then, we will prove how the language of an observation can be decided within logarithmic space.

This section uses the complexity classes DLOGSPACE and CONLOGSPACE, as well as notions of completeness of a problem and reduction between problems. We use in [Sect. 4.2](#) the classical theorem of CONLOGSPACE-completeness of the acyclicity problem in directed graphs, and in [Sect. 4.1](#) a convenient model of computation, two-ways multi-heads finite automata, a generalization of automata also called “pointer machine”. The reader who would like some reminders may have a look at any classical textbook [\[28\]](#). Note that the non-deterministic part of our results concerns CONLOGSPACE, or equivalently NLOGSPACE by the famous Immerman-Szelepcsényi theorem.

4.1 Completeness: Observations as Pointer Machines

This section describes the kind of computation one can model with observations.

Let h_0, x, y be variables, $\mathbf{p}_0, \mathbf{p}_1, \mathbf{A}_0$ constants and $\Sigma = \{0, 1\}$, the excerpt of a dialogue in [Figure 1](#) between an observation $O = o_1 + o_2 + \dots$ and the representation of a word $\bar{W}_p = w_1 + w_2 + \dots$ should help the reader to grasp the mechanism.

We just depicted two transitions corresponding to an automata that reads the first bit of the word, and if this bit is a 1, goes back to the starting position, in state **b**. Remark that the answer of w_1 differs from the one of w_2 : there is no need

$$\begin{aligned}
\star \bullet \mathbf{L} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(h_0) &\leftarrow \star \bullet \mathbf{R} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(h_0) & (o_1) \\
\star \bullet \mathbf{R} \bullet x \bullet y \bullet \mathbf{M}(p_0) &\leftarrow 1 \bullet \mathbf{L} \bullet x \bullet y \bullet \mathbf{M}(p_1) & (w_1) \\
1 \bullet \mathbf{L} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(h_0) &\leftarrow 1 \bullet \mathbf{L} \bullet \mathbf{b} \bullet \mathbf{A}_0 \bullet \mathbf{M}(h_0) & (o_2) \\
1 \bullet \mathbf{L} \bullet x \bullet y \bullet \mathbf{M}(p_1) &\leftarrow \star \bullet \mathbf{R} \bullet x \bullet y \bullet \mathbf{M}(p_0) & (w_2)
\end{aligned}$$

By unification,

$$\begin{aligned}
\star \bullet \mathbf{L} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(p_0) &\leftarrow 1 \bullet \mathbf{L} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(p_1) & (o_1 w_1) \\
\star \bullet \mathbf{L} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(p_0) &\leftarrow 1 \bullet \mathbf{L} \bullet \mathbf{b} \bullet \mathbf{A}_0 \bullet \mathbf{M}(p_1) & (o_1 w_1 o_2) \\
\star \bullet \mathbf{L} \bullet \mathbf{init} \bullet \mathbf{A}_0 \bullet \mathbf{M}(p_0) &\leftarrow \star \bullet \mathbf{R} \bullet \mathbf{b} \bullet \mathbf{A}_0 \bullet \mathbf{M}(p_0) & (o_1 w_1 o_2 w_2)
\end{aligned}$$

This can be understood as the small following dialogue:

o_1 : [*Is in state init*] “I read \star from left to right, what do I read now?”
 w_1 : “Your position was p_0 , you are now in position p_1 and read 1.”
 o_2 : [*Change state to b*] “I do an about-turn, what do I read now?”
 w_2 : “You are now in position p_0 and read \star .”

Fig. 1. An example of dialogue between an observation and the representation of a word

to clarify the position (the variable argument of \mathbf{M}), since h_0 was already replaced by p_1 . Such an information is needed only in the first step of computation: after that, the updates of the position of the pointer take place on the word side. Remark that neither the state nor the constant \mathbf{A}_0 is an object of dialogue.

Note also that this excerpt corresponds to a deterministic computation. In general, several elements of the observation could get unified with the current configuration, yielding non-deterministic transitions.

Multiple Pointers and Swapping We now add some computational power to our observation by adding the possibility to handle several pointers. The observations will now use a k -ary function \mathbf{A}_k that allows to “store” k positions. This part of the observation is not affected by the word, which means that only one head (the *main pointer*) can move. The observation can exchange the position of the main pointer and the position stored in \mathbf{A}_k : we therefore refer to the arguments of \mathbf{A}_k as *auxiliary pointers* that can become the main pointer at some point of the computation. This is of course strictly equivalent to several heads that have the ability to move.

Consider the following flow, that encodes the transition “if the observation reads $1 \bullet \mathbf{R}$ in state \mathbf{s} , it stores the position of the main pointer (the variable h_0) at the i -th position in \mathbf{A}_k and start reading the input with a new pointer”:

$$1 \bullet \mathbf{R} \bullet \mathbf{s} \bullet \mathbf{A}_k(h_1, \dots, h_i, \dots, h_k) \bullet \mathbf{M}(h_0) \leftarrow \star \bullet \mathbf{R} \bullet \mathbf{s}' \bullet \mathbf{A}_k(h_1, \dots, h_0, \dots, h_k) \bullet \mathbf{M}(h_i)$$

Suppose now we want to write an observation that, if it reads $0 \bullet \mathbf{L}$ in state \mathbf{r} , then it restores the value that was stored in the i -th position of \mathbf{A}_k , it stores the

position in M and changes to state \mathbf{r}' , how should we proceed? The point is that the flow above did not “memorize” the value that was stored at that position.

$$0.\mathbf{L}.\mathbf{r}.\mathbf{A}_k(h_1, \dots, h_0, \dots, h_k).\mathbf{M}(h_i) \leftarrow \mathbf{x}._\mathbf{r}'.\mathbf{A}_k(h_1, \dots, h_i, \dots, h_k).\mathbf{M}(h_0)$$

The variable x should be free and will be unified with the flow of the word that corresponds to the position constant that was substituted to h_0 . But there are two such flows in the word, so the direction of the next movement should be indicated in the $_$ slot. Such an information could be encoded in the state \mathbf{r}' ⁸ and in the transition function.

Acceptance and Rejection Remember (Corollary 32) that the language of an observation is the set of words such that the wiring composed of the observation applied to a representation of the word is nilpotent. So one could add a flow with the head corresponding to the desired situation leading to acceptance, and the body being 0. But in fact, it is sufficient not to add any flow: doing nothing is accepting!

The real challenge is to reject a word: it means to loop forever. We cannot simply add the unit ($I := x \leftarrow x$) to our observation, since that would make our observation loop *no matter the word in input*. So we have to be clever than that, and to encode rejection as a re-initialization of the observation: we want the observation to put all the pointers on \star and to go back to an **init** state. So, a loop is in fact a “perform for ever the same computation”.

If we reject, suppose the main pointer was reading from right to left, and that we changed to \mathbf{b} . Then, for every $c \in \Sigma$, it is enough to add the transitions (**go-back-c**) and (**re-init**) to the observation,

$$\begin{aligned} c.\mathbf{R}.\mathbf{b}.\mathbf{A}(h_1, \dots, h_k).\mathbf{M}(h_0) &\leftarrow c.\mathbf{L}.\mathbf{b}.\mathbf{A}(h_1, \dots, h_k).\mathbf{M}(h_0) && (\text{go-back-c}) \\ \star.\mathbf{R}.\mathbf{b}.\mathbf{A}(h_1, \dots, h_k).\mathbf{M}(h_0) &\leftarrow \star.\mathbf{R}.\mathbf{init}.\mathbf{A}(h_0, \dots, h_0).\mathbf{M}(h_0) && (\text{re-init}) \end{aligned}$$

Once the main pointer is back on \star , (**re-init**) re-initializes all the positions of the auxiliary pointers to the position of \star and changes the state for **init**.

There is another justification for this design: as the observation and the representation of the word are sums, and as the computation is the application, any transition that can be applied will be applied, *i.e.* if the body (the right-member) of a flow of our observation and the head (the left-member) of a flow of the word can be unified, the computation will start in a possibly “wrong” initialization. That some of this incorrect runs accept for incorrect reason is no trouble, since only rejection is “meaningful” due to the nilpotency criterion. But, with this framework, an incorrect run will be re-initialized to the “right” initialization, and performs the correct computation: in that case, it will loop if and only if the input is rejected.

Two-Ways Multi-Heads Finite Automata and Completeness The model we just developed has clearly the same expressivity as two-ways multi-heads finite

⁸That is, we could have two states \mathbf{r}'_L and \mathbf{r}'_R and two flows accordingly.

automata, a model of particular interest to us for it is well studied, tolerant to a lot of enhancements or restrictions⁹ and gives an elegant characterization of DLOGSPACE and NLOGSPACE [22,27].

Then, by a plain and uniform encoding of two-ways multi-heads finite automata, we get [Theorem 33](#). That acceptance and rejection in the non-deterministic case are “reversed” (*i.e.* all path have to accept for the computation to accepts) makes us characterize CONLOGSPACE instead of NLOGSPACE.

Note that encoding a *deterministic* automaton yields a wiring of the form of [Lemma 11](#), which would be therefore a deterministic wiring.

Theorem 33. *If $L \in \text{CONLOGSPACE}$, then there is an observation O such that $\mathcal{L}(O) = L$. Moreover, if $L \in \text{DLOGSPACE}$, O can be chosen deterministic.*

4.2 Soundness of Observations

We now use the results of [Sect. 2.3](#) and [Sect. 3.2](#) to design a procedure that decides whether a word belongs to the language of an observation within logarithmic space. This procedure will use the classical reduction from a problem to testing the acyclicity of a graph, a problem well-known to be tractable with logarithmic space resources.

First, we show how the computation graph of the product of the observation and the word representation can be constructed deterministically using only logarithmic space; then, we prove that testing the acyclicity of such a graph can be done within the same bounds. Here, depending on the shape of the graph (which is dependent in itself of determinism of the wiring, recall [Lemma 21](#)), the procedure will be deterministic or non-deterministic.

Finally, using classical composition of logspace algorithms,¹⁰ [Lemma 20](#) and [Corollary 32](#), we will obtain the result expected, that is:

Theorem 34. *If O is an observation, then $\mathcal{L}(O) \in \text{CONLOGSPACE}$. If moreover O is deterministic, then $\mathcal{L}(O) \in \text{DLOGSPACE}$.*

A Foreword on Word and Size Given a word W over Σ , to build a representation \bar{W}_p as in [Definition 28](#) is clearly in DLOGSPACE: it is a plain matter of encoding. By [Lemma 31](#), it is sufficient to consider a single representation. So for the rest of this procedure, we consider given $\bar{W}_p \in \mathcal{R}(W)$ and write $F := O\bar{W}_p$. The size of Σ is a constant, and it is clear that the maximal arity and the height of the balanced wiring F remain fixed when W varies. The only point that fluctuates is the cardinality of the set of symbols that occurs in F , and it is linearly growing with the length of W , corresponding to the number of positions constant. In the following, any mention to a logarithmic amount of space is to be completed by “relatively to the length of W ”.

⁹In fact, most of the variations (the automata can be one-way, sweeping, rotating, oblivious, etc.) are studied in terms of number of states needed to simulate a variation with another, but most of the time they characterize the same complexity classes.

¹⁰The reader unused to the mechanism of composition of logspace-algorithms may have a look at a classical textbook [28, Fig. 8.10].

Building the Computation Graph We need two main ingredients to build the computation graph (Definition 19) of F : to enumerate the computation space $\mathbf{Comp}(F)$ (recall Definition 16), and to determine whether there is an edge between two vertices.

By Proposition 18, $\text{card}(\mathbf{Comp}(F))$ is polynomial in the size of W . Hence, given a balanced wiring F , a logarithmic amount of memory is enough to enumerate the members of $\mathbf{Comp}(F)$, that is the vertices of $\mathbf{G}(F)$.

Now the second part of the construction of $\mathbf{G}(F)$ is to determine if there is an edge between two vertices. Remember that there is an edge from $\mathbf{u} = u \leftarrow \star$ to $\mathbf{v} = v \leftarrow \star$ in $\mathbf{G}(F)$ if $\mathbf{v} \in F\mathbf{u}$. So one has to scan the members of $F = O\bar{W}_p$: if there exists $(t_1 \leftarrow t_2)(t'_1 \leftarrow t'_2) \in F$ such that $(t_1 \leftarrow t_2)(t'_1 \leftarrow t'_2)(u \leftarrow \star) = v \leftarrow \star$, then there is an edge from \mathbf{u} to \mathbf{v} . To list the members of F is in DLOGSPACE, but unification in general is a difficult problem (see Sect. 1.2). The special case of matching can be tested with a logarithmic amount of space:

Theorem 35 (Matching is in DLOGSPACE [11, p. 49]). *Given two terms t and u such that either t or u is closed, deciding if they are matchable is in DLOGSPACE.*

Actually, this result relies on a subtle manipulation of the representation of the terms as *simple directed acyclic graphs*,¹¹ where the variables are “shared”. Translations between this representation of terms and the usual one can be performed in logarithmic space [11, p. 38].

Deciding if $\mathbf{G}(F)$ is Acyclic We know thanks to Lemma 20 that answering this question is equivalent to deciding if F is nilpotent. We may notice that $\mathbf{G}(F)$ is a directed, potentially unconnected graph of size $\text{card}(\mathbf{Comp}(F))$.

It is well-known that testing for acyclicity of a directed graph is a CONLOGSPACE [23, p. 83] problem. Moreover, if F is deterministic (which is the case when O is), then $\mathbf{G}(F)$ has out-degree bounded by 1 (Lemma 21) and one can test its acyclicity without being non-deterministic: it is enough to list the vertices of $\mathbf{Comp}(F)$, and for each of them to follow $\text{card}(\mathbf{Comp}(F))$ edges and to test for equality with the vertex picked at the beginning. If a loop is found, the algorithm rejects, otherwise it accepts after testing the last vertex. Only the starting vertex and the current vertex need to be stored, which fits within logarithmic space, and there is no need to do any non-deterministic transitions.

5 Conclusion

We presented the unification semiring, a construction that can be used both as an algebraic model of logic programming and as a setting for a dynamic model of logic. Within this semiring, we were able to identify a class of wirings that have the exact expressive power of logarithmic space computation.

¹¹The reader willing to learn more about this may have a look at a classical chapter [4].

If we try to step back a little, we can notice that the main tool in the soundness proof (Sect. 4.2) is the computation graph, defined in Sect. 2.3. More precisely, the properties of this graph, notably its cardinal (that turns out to be polynomial in the size of the input), allow to define a decision procedure that needs only logarithmic space. The technique is modular, hence not limited to logarithmic space: identifying other conditions on wirings that ensure size bounds on the computation graph would be a first step towards the characterization of other space complexity classes.

Concerning completeness, the choice of encoding pointer machines (Sect. 4.1) rather than log-space bounded Turing machines was quite natural. Balanced wirings correspond to the idea of computing with pointers: manipulation of data without writing abilities, and thus with no capacity to store any information other than a fixed number of positions on the input.

By considering other classes of wirings or by modifying the encoding it might be possible to capture other notions of machines characterizing some complexity classes: we already mentioned that a modification of the representation of the input that would model one-way finite automata.

The relation with proof theory needs to be explored further: the approach of this paper seems indeed to suggest a sort of “Curry-Howard” correspondance for logic programming.

As Sect. 1.1 highlighted, there are many notions that might be transferable from one field to the other, thanks to a common setting provided by geometry of interaction and the unification semiring. Most notably, the notion of nilpotency (on the proof-theoretic side: strong normalization) corresponds to a variant of boundedness of logic programs, a property that is usually hard to ensure.

Another direction could be to look for a proof-system counterpart of this work: a corresponding “balanced” logic of logarithmic space.

References

1. Asperti, A., Danos, V., Laneve, C., Regnier, L.: Paths in the lambda-calculus. In: LICS. pp. 426–436. IEEE Computer Society (1994)
2. Aubert, C., Bagnol, M.: Unification and logarithmic space. In: Dowek, G. (ed.) RTA-TLCA 2014. LNCS, vol. 8650, pp. 77–92. Springer (2014)
3. Aubert, C., Seiller, T.: Characterizing co-nl by a group action. Arxiv preprint abs/1209.3422 (2012)
4. Baader, F., Snyder, W.: Unification theory. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 445–532. Elsevier and MIT Press (2001)
5. Baillot, P., Mazza, D.: Linear logic by levels and bounded time complexity. Theoret. Comput. Sci. 411(2), 470–503 (2010)
6. Baillot, P., Pedicini, M.: Elementary complexity and geometry of interaction. Fund. Inform. 45(1–2), 1–31 (2001)
7. Bellia, M., Occhiuto, M.E.: N-axioms parallel unification. Fund. Inform. 55(2), 115–128 (2003)
8. Calimeri, F., Cozza, S., Ianni, G., Leone, N.: Computable functions in ASP: Theory and implementation. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP. LNCS, vol. 5366, pp. 407–424. Springer (2008)

9. Dal Lago, U., Hofmann, M.: Bounded linear logic, revisited. *Log. Meth. Comput. Sci.* 6(4) (2010)
10. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3), 374–425 (2001)
11. Dwork, C., Kanellakis, P.C., Mitchell, J.C.: On the sequential nature of unification. *J. Log. Program.* 1(1), 35–50 (1984)
12. Dwork, C., Kanellakis, P.C., Stockmeyer, L.J.: Parallel algorithms for term matching. *SIAM J. Comput.* 17(4), 711–731 (1988)
13. Gaboardi, M., Marion, J.Y., Ronchi Della Rocca, S.: An implicit characterization of pspace. *ACM Trans. Comput. Log.* 13(2), 18:1–18:36 (2012)
14. Girard, J.Y.: Linear logic. *Theoret. Comput. Sci.* 50(1), 1–101 (1987)
15. Girard, J.Y.: Geometry of interaction 1: Interpretation of system F. *Studies in Logic and the Foundations of Mathematics* 127, 221–260 (1989)
16. Girard, J.Y.: Towards a geometry of interaction. In: Gray, J.W., Ščedrov, A. (eds.) *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference held June 14–20, 1987. Categories in Computer Science and Logic*, vol. 92, pp. 69–108. AMS (1989)
17. Girard, J.Y.: Geometry of interaction III: accommodating the additives. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) *Advances in Linear Logic*, pp. 329–389. No. 222 in *London Math. Soc. Lecture Note Ser.*, CUP (Jun 1995)
18. Girard, J.Y.: Light linear logic. In: Leivant, D. (ed.) *LCC, LNCS*, vol. 960, pp. 145–176. Springer (1995)
19. Girard, J.Y.: Normativity in logic. In: Dybjer, P., Lindström, S., Palmgren, E., Sundholm, G. (eds.) *Epistemology versus Ontology, Logic, Epistemology, and the Unity of Science*, vol. 27, pp. 243–263. Springer (2012)
20. Girard, J.Y.: Three lightings of logic. In: Ronchi Della Rocca, S. (ed.) *CSL. LIPIcs*, vol. 23, pp. 11–23. *Schloss Dagstuhl - Leibniz-Zentrum für Informatik* (2013)
21. Hillebrand, G.G., Kanellakis, P.C., Mairson, H.G., Vardi, M.Y.: Undecidable boundedness problems for datalog programs. *J. Log. Program.* 25(2), 163–190 (1995)
22. Holzer, M., Kutrib, M., Malcher, A.: Multi-head finite automata: Characterizations, concepts and open problems. In: Neary, T., Woods, D., Seda, A.K., Murphy, N. (eds.) *CSP. EPTCS*, vol. 1, pp. 93–107 (2008)
23. Jones, N.D.: Space-bounded reducibility among combinatorial problems. *J. Comput. Syst. Sci.* 11(1), 68–85 (1975)
24. Laurent, O.: A token machine for full geometry of interaction (extended abstract). In: Abramsky, S. (ed.) *TLCA. LNCS*, vol. 2044, pp. 283–297. Springer (May 2001)
25. Lierler, Y., Lifschitz, V.: One more decidable class of finitely ground programs. In: Hill, P.M., Warren, D.S. (eds.) *ICLP. LNCS*, vol. 5649, pp. 489–493. Springer (2009)
26. Ohkubo, M., Yasuura, H., Yajima, S.: On parallel computation time of unification for restricted terms. *Tech. rep.*, Kyoto University (May 1987)
27. Pighizzini, G.: Two-way finite automata: Old and recent results. *Fund. Inform.* 126(2–3), 225–246 (2013)
28. Savage, J.E.: *Models of computation - exploring the power of computing*. Addison-Wesley (1998)
29. Schöpp, U.: Stratified bounded affine logic for logarithmic space. In: *LICS*. pp. 411–420. IEEE Computer Society (2007)
30. Seiller, T.: Interaction graphs: Multiplicatives. *Ann. Pure Appl. Logic* 163, 1808–1837 (Dec 2012)