Disclaimer

This document is a post-print version of an article published in Lecture Notes in Computer Science (as allowed by their policy).

To cite the published version (which is the only worth citing), please use

```
@InProceedings{aubert2016fossacs,
  title     = {Unary Resolution: Characterizing Ptime},
  author    = {Aubert, Clément and Bagnol, Marc and Seiller, Thomas},
  year      = {2016},
  editor    = {Jacobs, Bart and Löding, Christof},
  booktitle = {FOSSACS},
  publisher = {Springer},
  series    = {Lecture Notes in Computer Science},
  pages     = {373--389},
  doi       = {10.1007/978-3-662-49630-5_22},
  volume    = {9634},
  url       = {http://link.springer.com/chapter/10.1007%2F978-3-662-49630-5_22},
  note      = {Post-print version available at
               \url{https://lacl.fr/~caubert/
               recherche/Unary_Resolution_Characterizing_Ptime.pdf}}
}
```

2016/05/12

Unary Resolution: Characterizing Ptime*

Clément Aubert[1], Marc Bagnol[2], and Thomas Seiller[3]

[1] Department of Computer Science, Appalachian State University
[2] Department of Mathematics and Statistics, University of Ottawa
[3] Department of Computer Science, University of Copenhagen

Abstract. We give a characterization of deterministic polynomial time computation based on an algebraic structure called the resolution semiring, whose elements can be understood as logic programs or sets of rewriting rules over first-order terms. This construction stems from an interactive interpretation of the cut-elimination procedure of linear logic known as the *geometry of interaction*.
This framework is restricted to terms (logic programs, rewriting rules) using only unary symbols, and this restriction is shown to be complete for polynomial time computation by encoding pushdown automata. Soundness w.r.t. Ptime is proven thanks to a saturation method similar to the one used for pushdown systems and inspired by the memoization technique.
A Ptime-completeness result for a class of logic programming queries that uses only unary function symbols comes as a direct consequence.

Keywords: implicit complexity, unary queries, logic programming, geometry of interaction, proof theory, pushdown automata, saturation, memoization

1 Introduction

Complexity theory classifies computational problems relatively to the amount of time or memory needed to solve them. Complexity classes are defined as sets of problems that can be solved by algorithms whose executions need comparable amounts of resources. For instance, the class Ptime is the set of predicates over binary words that can be decided by a Turing machine whose execution time is bounded by a polynomial in the size of its input.

One of the main motivations for an implicit computational complexity (ICC) theory is to find machine-independent characterizations of complexity classes. The aim is to characterize classes not "*by constraining the amount of resources a machine is allowed to use, but rather by imposing linguistic constraints on the way algorithms are formulated.*" [18, p. 90] This has been already achieved

* This work was partly supported by the ANR-14-CE25-0005 Elica, the ANR-11-INSE-0007 Rever, the ANR-10-BLAN-0213 Logoi, the ANR-11-BS02-0010 Récré, the ANR 12 JS02 006 01 project Coquas and the European Union's Marie Skłodowska-Curie Individual Fellowship (H2020-MSCA-IF-2014) 659920 - ReACT.

via different approaches, for instance by considering restricted programming languages or computational principles [12,37,38].

A number of results in this area also arose from proof theory, through the study of subsystems of linear logic [25]. More precisely, the Curry-Howard—or *proofs as programs*—correspondence expresses an isomorphism between formal proofs and typed programs. In this approach, once a formula `Nat` corresponding to the type of binary integers is set, proofs of the formula `Nat` $\Rightarrow$ `Nat` *are* algorithms computing functions from integers to integers, via the cut-elimination procedure. By considering restricted subsystems, one allows *less* proofs of type `Nat` $\Rightarrow$ `Nat`, hence *less* algorithms can be implemented, and the class of accepted proofs, or programs, may correspond[4] to some complexity class: elementary complexity [29,21], polynomial time [36,11], logarithmic [19] and polynomial [24] space.

More recently, new methods for obtaining implicit characterizations of complexity classes based on the *geometry of interaction* (GoI) research program [27] have been developed. The GoI approach offers a more abstract and algebraic point of view on the cut-elimination procedure of linear logic. One works with a set of *untyped programs* represented as some geometric objects, e.g. graphs [20,40] or generalizations of graphs [42], bounded linear maps between Hilbert spaces (operators) [26,30,41], clauses (or "flows") [28,8]. This set of objects is then considered together with an abstract notion of execution, seen as an interactive procedure: a function does not process a static input, but rather communicate with it, asking for values, reading its answers, asking for another value, etc. (this interactive point of view on computation has proven crucial in characterizing logarithmic space computation [19]).

This method does not proceed by restricting a type system, but by imposing original conditions, of an algebraic nature, on the representation of programs. Note that one still benefits from the work in the typed case: for instance, the representation of words used here directly comes from their representation in linear logic. The first results in this direction were based on operator algebra [31,5,6]. This paper considers a more syntactic flavor of the GoI interpretation, where untyped programs are represented in the so-called *resolution semiring* [8], a semiring based on the resolution rule [39] and a specific class of logic programs. This setting presents some advantages: it avoids the heavy structure of operator algebras theory, eases the discussions in terms of complexity (as first-order terms have natural notions of size, height, etc.) and offers a straightforward connection with complexity of logic programming [22]. Previous works in this direction led to characterizations of logarithmic space predicates Logspace and co-NLogspace [2,3], by considering restrictions on the height of variables.

The main contribution of this paper is a characterization of the class Ptime by studying a natural limitation, the restriction to *unary* function symbols. Pushdown automata are easily related to this simple restriction, for they can be represented as logical programs satisfying this "unarity" restriction. This implies the completeness of the model under consideration for polynomial time predicates.

[4] In an *extensional* correspondence: they can compute the same functions.

Soundness follows from a variation of the saturation algorithm for pushdown systems [13], inspired by S. Cook's memoization technique [17] for pushdown automata, that proves that any such unary logic program can be decided in polynomial time.

Compared to other ICC characterizations of Ptime, and in particular those coming from proof theory, this method has a simple formulation and provides an original point of view on this complexity class. It also constitutes the first characterization of a time-complexity class directly on the semantic side of the GoI interpretation. Nevertheless, the results presented here can be read independently of any knowledge of GoI.

An immediate consequence of this work is a Ptime-completeness result for a specific class of logic programming queries corresponding to unary flows.

1.1 Outline

Sect. 2.1 gives the formal definition of the resolution semiring; then representation of words and programs in this structure is briefly explained (Sect. 2.2). Sect. 2.3 introduce the restricted semiring that will be under study, the $\mathcal{S}tack$ semiring. We believe the technical results presented in this section to be of importance, as they describe an algebraic restriction corresponding to Ptime and broaden previous algebraic restrictions for (co-N)Logspace.

The next two sections are respectively devoted to the completeness and soundness results for Ptime. Proving completeness needs to first review multi-head finite automata with pushdown stack, that characterize Ptime, and then to represent them as elements built from the $\mathcal{S}tack$ semiring (Sect. 3). The soundness result is then obtained by "saturating" elements of the stack semiring, so that they become decidable with Ptime resources (Sect. 4).

Ptime-completeness of unary logic programming queries is then proved to be implied by this result (Sect. 5).

Sketched proofs are detailed in a technical report [4] and in the second author's Ph.D. thesis [8].

2 The Resolution Semiring

2.1 Flows and Wirings

We begin with some reminders on first-order terms and unification theory.

Notation 1 (terms). We consider first-order terms, written $\mathrm{t}, \mathrm{u}, \mathrm{v}, \ldots$, built from variables and function symbols with assigned finite arity. Symbols of arity 0 will be called *constants*.

Sets of variables and of function symbols of any arity are supposed infinite. Variables will be noted in italics font (e.g. x, y) and function symbols in typewriter font (e.g. $\mathtt{c}, \mathtt{f}(\cdot), \mathtt{g}(\cdot,\cdot)$).

We distinguish a binary function symbol $\bullet$ (in infix notation) and a constant symbol $\star$. We will omit the parentheses for $\bullet$ and write $\mathrm{t}\bullet\mathrm{u}\bullet\mathrm{v}$ for $\mathrm{t}\bullet(\mathrm{u}\bullet\mathrm{v})$.

We write $\mathtt{var}(\mathrm{t})$ the set of variables occurring in the term t and say that t is *closed* if $\mathtt{var}(\mathrm{t}) = \varnothing$. The *height* $\boldsymbol{h}(\mathrm{t})$ of a term t is the maximal distance between its root and leaves; a variable occurrence's height in t is its distance to the root.

We will write $\theta\mathrm{t}$ for the result of applying the substitution θ to the term t and will call *renaming* a substitution α that bijectively maps variables to variables.

We focus on the resolution of equalities $\mathrm{t} = \mathrm{u}$ between terms, and hence need some definitions.

Definition 2 (unification, matching and disjointness). *Two terms* t, u *are:*
- unifiable *if there exists a substitution* θ*—a* unifier *of* t *and* u*—such that* $\theta\mathrm{t} = \theta\mathrm{u}$*. If any other unifier of* t *and* u *is such that there exists a substitution that maps it to* θ*, we say* θ *is the* most general unifier *(MGU) of* t *and* u*;*
- matchable *if* t', u' *are unifiable, where* t', u' *are renamings of* t, u *such that* $\boldsymbol{var}(\mathrm{t}') \cap \boldsymbol{var}(\mathrm{u}') = \varnothing$*;*
- disjoint *if they are not matchable.*

A fundamental result of unification theory is that when two terms are unifiable, a MGU exists and is computable. More specifically, the problem of deciding whether two terms are unifiable is Ptime-complete [23, Theorem 1]. The notion of MGU allows to formulate the *resolution rule*, a key concept of logic programming defining the composition of Horn clauses (expressions of the form $H \dashv B_1, \ldots, B_n$):

$$\frac{\begin{array}{ll} V \dashv T_1, \ldots, T_n & \mathtt{var}(U) \cap \mathtt{var}(V) = \varnothing \\ H \dashv B_1, \ldots, B_m, U & \theta \text{ is a MGU of } U \text{ and } V \end{array}}{\theta H \dashv \theta B_1, \ldots, \theta B_m, \theta T_1, \ldots, \theta T_n}\ \mathtt{Res}$$

Note that the condition on variables implies that we are matching U and V rather than unifying them. In other words, the resolution rule deals with variables as if they were bounded.

From this perspective, "flows"—defined below—are a specific type of Horn clauses $H \dashv B$, with exactly one formula B at the right of $\dashv$ and all the variables of H already occurring in B. The product of flows will be defined as the resolution rule restricted to this specific type of clauses.

Definition 3 (flow). *A* flow *is an ordered pair* f *of terms* $f \coloneqq \mathrm{t} \leftharpoonup \mathrm{u}$*, with* $\boldsymbol{var}(\mathrm{t}) \subseteq \boldsymbol{var}(\mathrm{u})$*. Flows are considered up to renaming: for any renaming* α*,* $\mathrm{t} \leftharpoonup \mathrm{u} = \alpha\mathrm{t} \leftharpoonup \alpha\mathrm{u}$*.*

A flow can be understood as a rewriting rule over the set of first-order terms, acting at the root. For instance, the flow $\mathtt{g}(x) \leftharpoonup \mathtt{f}(x)$ corresponds to "rewrite terms of the form $\mathtt{f}(\mathrm{v})$ as $\mathtt{g}(\mathrm{v})$".

Next comes the definition of the *product* of flows. From the term-rewriting perspective, this operation corresponds to composing two rules—when possible, i.e. when the result of the first rewriting rule allows the application of the second—into a single one. For instance, one can compose the flows $f_1 \coloneqq \mathtt{h}(x) \leftharpoonup \mathtt{g}(x)$ and

$f_2 \coloneqq \mathtt{g}(\mathtt{f}(x)) \leftharpoonup \mathtt{f}(x)$ to produce the flow $f_1 f_2 = \mathtt{h}(\mathtt{f}(x)) \leftharpoonup \mathtt{f}(x)$. Notice by the way that this (partial) product is not commutative, as composing these rules the other way around is impossible, i.e. $f_2 f_1$ is not defined.

Definition 4 (product of flows). *Let* $\mathrm{t} \leftharpoonup \mathrm{u}$ *and* $\mathrm{v} \leftharpoonup \mathrm{w}$ *be two flows. Suppose we picked representatives of the renaming classes such that* $\mathtt{var}(\mathrm{u}) \cap \mathtt{var}(\mathrm{v}) = \varnothing$.

The product *of* $\mathrm{t} \leftharpoonup \mathrm{u}$ *and* $\mathrm{v} \leftharpoonup \mathrm{w}$ *is defined when* u *and* v *are unifiable, with MGU* θ*, as* $(\mathrm{t} \leftharpoonup \mathrm{u})(\mathrm{v} \leftharpoonup \mathrm{w}) \coloneqq \theta\mathrm{t} \leftharpoonup \theta\mathrm{w}$.

We now define wirings, which are just finite sets of flows and therefore correspond to logic programs. From the term-rewriting perspective they are just sets of rewriting rules. The definition of product of flows naturally lifts to wirings.

Definition 5 (wiring). *A* wiring *is a finite set of flows. Their product is defined as* $FG \coloneqq \{\, fg \mid f \in F,\, g \in G,\, fg \text{ defined} \,\}$. *The* resolution semiring $\mathcal{R}$ *is the set of all wirings.*

The set of wirings $\mathcal{R}$ indeed enjoys a structure of semiring.[5] We will use an *additive notation* for sets of flows to highlight this situation:

- The symbol $+$ will be used in place of $\cup$, and we write sets as sums of their elements: $\{\, f_1, \ldots, f_n \,\} \coloneqq f_1 + \cdots + f_n$.
- We denote by 0 the empty set, i.e. the unit of the sum.
- The unit for the product is the wiring $I \coloneqq x \leftharpoonup x$.

As we will always be working within $\mathcal{R}$, the term "semiring" will be used instead of "subsemiring of $\mathcal{R}$". Finally, let us extend the notion of height to flows and wirings, and recall the definition of nilpotency.

Definition 6 (height). *The* height $\hbar(f)$ *of a flow* $f = \mathrm{t} \leftharpoonup \mathrm{u}$ *is defined as* $\max\{\hbar(\mathrm{t}), \hbar(\mathrm{u})\}$. *A wiring's* height *is defined as* $\hbar(F) = \max\{\, h(f) \mid f \in F \,\}$. *By convention* $\hbar(0) = 0$.

Definition 7 (nilpotency). *A wiring* F *is* nilpotent*—written* $\mathbf{Nil}(F)$*—if and only if* $F^n = 0$ *for some* n.

That standard notion of nilpotency, coming from abstract algebra, has a specific reading in our case. In terms of logic programming, it means that all chains obtained by applying the resolution rule to the set of clauses we consider cannot be longer than a certain bound. From a rewriting point of view, it means that the set of rewriting rules is terminating with a uniform bound on the length of rewriting chains—again, we consider rewriting at the root of terms, while general term rewriting systems [7] allow for in-context rewriting.

[5] A *semiring* is a set R equipped with two operations $+$ (the sum) and $\times$ (the product), and an element $0 \in R$ such that: $(R, +, 0)$ is a commutative monoid; $(R, \times)$ is a *semigroup*; the product distributes over the sum; and the element 0 is absorbent, i.e. $0 \times r = r \times 0 = 0$ for all $r \in R$.

2.2 Representation of Words and Programs

This section explains and motivates the representation of words as flows. Studying their interaction with wirings from a specific semiring allows to define notions of program and accepted language. First, let us extend the binary function symbol $\bullet$, used to construct terms, to flows and then semirings.

Definition 8. *Let* $\mathrm{u} \leftharpoonup \mathrm{v}$ *and* $\mathrm{t} \leftharpoonup \mathrm{w}$ *be two flows. Suppose we have chosen representatives of their renaming classes that have disjoint sets of variables.*

We define $(\mathrm{u} \leftharpoonup \mathrm{v}) \bullet (\mathrm{t} \leftharpoonup \mathrm{w}) := \mathrm{u} \bullet \mathrm{t} \leftharpoonup \mathrm{v} \bullet \mathrm{w}$. *The operation is extended to wirings by* $(\sum_i f_i) \bullet (\sum_j g_j) := \sum_{i,j} f_i \bullet g_j$. *Then, given two semirings* $\mathcal{A}$ *and* $\mathcal{B}$, *we define the semiring* $\mathcal{A} \bullet \mathcal{B} := \{ \sum_i F_i \bullet G_i \mid F_i \in \mathcal{A}, G_i \in \mathcal{B} \}$.

The operation indeed defines a semiring because for any wirings F, F', G and G', we have $(F \bullet G)(F \bullet G) = FF' \bullet GG'$. Moreover, we carry on the convention of writing $\mathcal{A} \bullet \mathcal{B} \bullet \mathcal{C}$ for $\mathcal{A} \bullet (\mathcal{B} \bullet \mathcal{C})$.

Notation 9. We write $\mathrm{t} \leftrightharpoons \mathrm{u}$ the sum $\mathrm{t} \leftharpoonup \mathrm{u} + \mathrm{u} \leftharpoonup \mathrm{t}$.

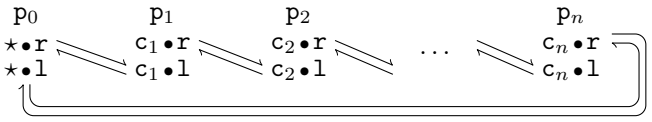
Definition 10 (word representation). *We fix for the rest of this paper an infinite set of constant symbols* $\mathtt{P}$ *(the* position constants*) and a finite alphabet* Σ *disjoint from* $\mathtt{P}$ *with* $\star \notin \Sigma$ *(we write* Σ^* *the set of words over* Σ*).*

Let $W = \mathtt{c}_1 \cdots \mathtt{c}_n \in \Sigma^*$ *and* $p = \mathtt{p}_0, \mathtt{p}_1, \ldots, \mathtt{p}_n$ *be pairwise distinct elements of* $\mathtt{P}$. *Writing* $\mathtt{p}_{n+1} = \mathtt{p}_0$ *and* $\mathtt{c}_{n+1} = \mathtt{c}_0 = \star$, *we define the* representation *of* W *associated with* $\mathtt{p}_0, \mathtt{p}_1, \ldots, \mathtt{p}_n$ *as the following wiring, where* x *and* y *are two arbitrary but distinct variables:*

$$\bar{W}_p = \sum_{i=0}^{n} \mathtt{c}_i \bullet \mathtt{r} \bullet x \bullet y \bullet \mathtt{HEAD}(\mathtt{p}_i) \leftrightharpoons \mathtt{c}_{i+1} \bullet \mathtt{l} \bullet x \bullet y \bullet \mathtt{HEAD}(\mathtt{p}_{i+1})$$

Position constants $\mathtt{p}_0, \mathtt{p}_1, \ldots, \mathtt{p}_n$ represent memory cells storing the symbols $\star$, $\mathtt{c}_1$, $\mathtt{c}_2$, ... The representation of words is *dynamic*, i.e. we may think intuitively of *movement instructions* for an automaton reading the input, getting instructions to move from a symbol to the next (at the left, hence the $\mathtt{l}$) or the previous (to the right, hence the $\mathtt{r}$). More details on this point of view will be given in the proof of Theorem 4. A term $\mathtt{c}_i \bullet \mathtt{r}$ (resp. $\mathtt{c}_i \bullet \mathtt{l}$) at position $\mathtt{p}$ will be linked by flows of the representation to an element $\mathtt{c}_{i+1} \bullet \mathtt{l}$ at position $\mathtt{p}_{i+1}$ (resp. $\mathtt{c}_{i-1} \bullet \mathtt{r}$ at position $\mathtt{p}_{i-1}$).

Taking $\mathtt{c}_{n+1} = \mathtt{c}_0 = \star$ reflects the Church encoding of words and make the representation of the input circular. Flows representing the word $\mathtt{c}_1 \cdots \mathtt{c}_n$ can be pictured as follows:



The notion of *observation* will be the counterpart of a program in our construction. We first give a general definition, that will be instantiated later to particular classes of observations characterizing complexity classes. The important point is that observations cannot use any position constant, so that they interact the same way with all the representations $\bar{W}_{\mathtt{p}}$ of a word W.

Definition 11 (observation semiring). *We define the semirings* $\mathtt{P}^{\perp}$ *of flows that do not use the symbols in* $\mathtt{P}$*; and* $\Sigma_{\mathtt{lr}}$ *the semiring generated by flows of the form* $\mathtt{c}\bullet\mathtt{d} \leftharpoonup \mathtt{c}'\bullet\mathtt{d}'$ *with* $\mathtt{c},\mathtt{c}' \in \Sigma \cup \{\star\}$ *and* $\mathtt{d},\mathtt{d}' \in \{\mathtt{l},\mathtt{r}\}$.

We define the semiring of observations *as* $\mathcal{O} \coloneqq (\Sigma_{\mathtt{lr}} \bullet \mathcal{R}) \cap \mathtt{P}^{\perp}$*, and the semiring of* observations over the semiring $\mathcal{A}$ *as* $\mathcal{O}[\mathcal{A}] \coloneqq (\Sigma_{\mathtt{lr}} \bullet \mathcal{A}) \cap \mathtt{P}^{\perp}$.

The following property is a consequence of the fact that observations cannot use position constants [8, Theorem IV.5].

Theorem 1 (normativity). *Let* $\bar{W}_p$ *and* $\bar{W}_q$ *be two representations of a word* W *and* O *an observation. Then* $\mathbf{Nil}(O\bar{W}_p)$ *if and only if* $\mathbf{Nil}(O\bar{W}_q)$.

How a word can be accepted by an observation can now be safely defined: the following definition is independent of the specific choice of a representation.

Definition 12 (accepted language). *Let* O *be an observation. The* language accepted by O *is defined as* $\mathcal{L}(O) := \{\, W \in \Sigma^* \mid \forall p,\, \mathbf{Nil}(O\bar{W}_p) \,\}$.

A previous work [3] investigated the semiring of *balanced wirings*, that are defined as sets of balanced—or "height-preserving"—flows.

Definition 13 (balance). *A flow* $f = \mathrm{t} \leftharpoonup \mathrm{u}$ *is* balanced *if for any variable* $x \in \mathtt{var}(\mathrm{t}) \cup \mathtt{var}(\mathrm{u})$*, all occurrences of* x *in both* t *and* u *have the same height (recall notations p. 4). A* balanced *wiring* F *is a sum of balanced flows and the set of balanced wirings is written* $\mathcal{R}_{\mathbf{b}}$.

A balanced observation *is an element of* $\mathcal{O}[\,\mathcal{R}_{\mathbf{b}} \bullet \mathcal{R}_{\mathbf{b}}\,]$.

This simple restriction was shown to characterize (non-deterministic) logarithmic space computation [3, Theorems 34-35], with a natural subclass of balanced wirings corresponding to the deterministic case. The balanced restriction won't be further considered, even if previous results on the nilpotency problem for balanced wirings [3, p. 54], [8, Theorem IV.12] are required to complete the detailed proof of Theorem 5 [4,8].

2.3 The $\mathcal{S}tack$ Semiring

This paper deals with another restriction on flows, namely the restriction to *unary flows*, i.e. defined with unary function symbols only. The semiring of wirings composed only of unary flows is called the $\mathcal{S}tack$ semiring, and will be shown to give a characterization of polynomial time computation. Below are the needed definitions and results about this semiring, a more complete picture is in the second author's Ph.D. thesis [8].

Definition 14 (unary flows). *A* unary flow *is a flow built using only unary function symbols and a variable. The semiring $\mathcal{S}tack$ is the set of wirings of the form $\sum_i \mathrm{t}_i \leftharpoonup \mathrm{u}_i$ where the $\mathrm{t}_i \leftharpoonup \mathrm{u}_i$ are unary flows.*

Example 1. The flows $\mathtt{f}(\mathtt{f}(x)) \leftharpoonup \mathtt{g}(x)$ and $x \leftharpoonup \mathtt{g}(x)$ are both unary, while $x \bullet \mathtt{f}(x) \leftharpoonup \mathtt{g}(x)$ and $\mathtt{f}(\mathtt{c}) \leftharpoonup x$ are not.

The notion of *cyclic flow* is crucial to prove the characterization of polynomial time computation. It is complementary to the nilpotency property for elements of $\mathcal{S}tack$, i.e. a wiring in $\mathcal{S}tack$ will be either cyclic or nilpotent.

Definition 15 (cyclicity). *A flow* $\mathrm{t} \leftharpoonup \mathrm{u}$ *is a* cycle *if* t *and* u *are matchable (Definition 2). A wiring F is* cyclic *if there is a k such that F^k contains a cycle.*

Remark 1. A flow f is a cycle iff $f^2 \neq 0$, which in turn implies $f^n \neq 0$ for all n in the case f is unary. This does not hold in general: $f = x \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet x$ is a cycle as $f^2 = \mathtt{c} \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet \mathtt{d} \neq 0$, but $f^3 = (x \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet x)(\mathtt{c} \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet \mathtt{d}) = 0$.

Theorem 2 (nilpotency). *A wiring $F \in \mathcal{S}tack$ is nilpotent iff it is acyclic.*

Proof (sketch [8, theorem II.52]). An immediate consequence of Remark 1 is that if F is acyclic, then it is not nilpotent. The converse is a consequence of a bound on the height of elements of F^n when F is acyclic [10]. From this, a contradiction can be obtained by realizing that manipulating bounded height terms built from a finite pool of symbols implies that one is wandering in a finite set and will eventually be cycling in it. □

Example 2. The following nilpotent element of $\mathcal{S}tack$ illustrates how the nilpotency problem can be tricky to solve efficiently:

$$\begin{aligned} F \coloneqq \quad & \mathtt{f}_1(x) \leftharpoonup \mathtt{f}_0(x) \\ & + \mathtt{f}_0(\mathtt{f}_1(x)) \leftharpoonup \mathtt{f}_1(\mathtt{f}_0(x)) \\ & + \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_1(x))) \leftharpoonup \mathtt{f}_1(\mathtt{f}_1(\mathtt{f}_0(x))) \\ & + \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \leftharpoonup \mathtt{f}_1(\mathtt{f}_1(\mathtt{f}_1(x))) \end{aligned}$$

Taking the sequence $\mathtt{f}_x\mathtt{f}_y\mathtt{f}_z$ to be the integer $x + 2y + 4z$, this wiring implements a counter from 0 to 7 in binary notation, that resets to 0 when it reaches 8. It is clear with this intuition in mind that this wiring is cyclic. Indeed, an easy computation shows that $\mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \leftharpoonup \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \in F^8$.

Lifting this example to the case of a counter from 0 to $2^n - 1$, gives a wiring for which the number of iterations needed to find a cycle is exponential in its size. This rules out a polynomial time decision procedure for the nilpotency problem that would simply compute iterations of a wiring until it finds a cycle.

Finally, let us define a new class of observations, based on the $\mathcal{S}tack$ semiring.

Definition 16 (balanced observation with stack). *A* balanced observation with stack *is an element of the semiring $\mathcal{O}^{\mathbf{b+s}} \coloneqq \mathcal{O}[\,\mathcal{S}tack \bullet \mathcal{R}_{\mathbf{b}}\,]$.*

3 Pushdown Automata and Ptime Completeness

Automata form a simple model of computation that can be extended in different ways. For instance, allowing multiple heads that can move in two directions on the input tape gives a model of computation equivalent to read-only Turing machines. If one adds moreover a "pushdown stack" one defines "pushdown automata", well-known to capture polynomial-time computation. Ptime-completeness of balanced observation with stacks will be attained by encoding pushdown automata: we recall briefly their definition and characterization of Ptime, before sketching how to represent them as observations.

Definition 17 (pushdown automata (2MFA+S)). *For $k \geqslant 1$, a* pushdown automaton *(formally, a* 2-way k-head finite automaton with pushdown stack (2MFA+S(k))*) is a tuple $M = \{\boldsymbol{S}, \boldsymbol{i}, A, B, \triangleright, \triangleleft, \boxdot, \sigma\}$ where:*

- *$\boldsymbol{S}$ is the finite set of* states*, with $\boldsymbol{i} \in \boldsymbol{S}$ the* initial state*;*
- *A is the* input alphabet*, B the* stack alphabet*;*
- *$\triangleright$ and $\triangleleft$ are the* left *and* right endmarkers*, $\triangleright, \triangleleft \notin A$;*
- *$\boxdot$ is the* bottom symbol of the stack*, $\boxdot \notin B$;*
- *σ is the* transition relation*, i.e. a subset of the product $(\boldsymbol{S} \times (A_{\bowtie})^k \times B_{\boxdot}) \times (\boldsymbol{S} \times \{-1, 0, +1\}^k \times \{\texttt{pop}, \texttt{push}(b)\})$ where $A_{\bowtie}$ (resp. $B_{\boxdot}$) denotes $A \cup \{\triangleright, \triangleleft\}$ (resp. $B \cup \{\boxdot\}$). The instruction -1 corresponds to moving the head one cell to the left, 0 corresponds to keeping the head on the current cell and $+1$ corresponds to moving it one cell to the right. Regarding the pushdown stack, the instruction* `pop` *means "erase the top symbol", while, for all $b \in B$,* $\texttt{push}(b)$ *means "write b on top of the stack".*

The automaton *rejects the input* if it loops, otherwise it *accepts*. This condition is equivalent to the standard way of defining acceptance and rejection by "reaching a final state" [35, Theorem 2]. Modulo another standard transformation, we restrict the transition relation so that at most one head moves at each transition.

We used in our previous work [3,6] the characterization of Logspace and NLogspace by 2-way k-head finite automata *without* pushdown stacks [43, pp. 223–225]. The addition of a pushdown stack improves the expressiveness of the machine model, as stated in the following theorem.

Theorem 3. *Pushdown automata characterize* Ptime*.*

Proof. Without reproving this classical result of complexity theory, we review the main ideas supporting it.

Simulating a Ptime *Turing machine with a Pushdown automata* amounts to designing an equivalent Turing machine whose movements of heads follow a regular pattern. That permits to seamlessly simulate their contents with a pushdown stack. A complete proof [16, pp. 9–11] as well as a precise algorithm [43, pp. 238–240] can be found in the literature.

Simulating a pushdown automata with a polynomial-time Turing machine cannot amount to simply simulate step-by-step the automaton with the Turing machine. The reason is that for any pushdown automaton, one can design a pushdown automaton that recognizes the same language but runs exponentially slower [1, p. 197]. That the pushdown automaton can accept its input after an exponential computation time is similar with the situation of the counter in Example 2.

The technique invented by Alfred V. Aho et al. [1] and made popular by Stephen A. Cook consists in building a "memoization table" allowing the Turing machine to create shortcuts in the simulation of the pushdown automaton, decreasing drastically its computation time. In some cases, an automaton with an exponentially long run can even be simulated in linear time [17]. □

Let us now consider the proof of Ptime-*completeness* for the set of balanced observations with stacks. It relies on an encoding that is similar to the previously developed encoding of 2-way k-head finite automata (*without* pushdown stack) by flows [3, Sect. 4.1]. The only difference is the addition of a "plug-in" that allows to represent stacks in observations.

Remember that acceptance by observations is phrased in terms of nilpotency of the product $O\bar{W}_{\mathtt{p}}$ of the observation and the representation of the input (Definition 12). Hence the computation in this model is defined as an iteration: one computes by considering the sequence $O\bar{W}_{\mathtt{p}}, (O\bar{W}_{\mathtt{p}})^2, (O\bar{W}_{\mathtt{p}})^3, \dots$ and the computation either ends at some point (i.e. accepts)—that is $(O\bar{W}_{\mathtt{p}})^n = 0$ for some integer n—or loops (i.e. rejects). This iteration represents a dialogue between the observation and its input: whereas an automaton is often thought of as manipulating some the "passive" data, in our setting, the observation and the word representation interact, taking turns in making the situation evolve.

Theorem 4. *If $L \in$ Ptime, then there exists a balanced observation with stack $O \in \mathcal{O}^{\mathbf{b+s}}$ such that $L = \mathcal{L}(O)$.*

Proof. Let $A = \Sigma$ be the input alphabet and M the 2MFA+S(k + 1) that recognizes L. By Theorem 3, such a M exists, and its transition relation is encoded as a balanced observation with stack (Definition 16). More precisely, the automaton will be represented as an element O_M of $\mathcal{O}^{\mathbf{b+s}} = \mathcal{O}[\,\mathcal{S}tack \bullet \mathcal{R}_{\mathbf{b}}\,]$ which can be written as a sum of flows of the form

$$\begin{aligned}\mathtt{c}' \bullet \mathtt{d}' \bullet \sigma(x) \bullet \mathtt{q}' \bullet \mathtt{AUX}_k(y'_1, \dots, y'_k) \bullet \mathtt{HEAD}(z') \leftharpoondown \\ \mathtt{c} \bullet \mathtt{d} \bullet \mathtt{s}(x) \bullet \mathtt{q} \bullet \mathtt{AUX}_k(y_1, \dots, y_k) \bullet \mathtt{HEAD}(z)\end{aligned}$$

with

- $\mathtt{c}, \mathtt{c}' \in \Sigma \cup \{\star\}$ and $\mathtt{d}, \mathtt{d}' \in \{\mathtt{l}, \mathtt{r}\}$,
- σ a finite sequence of unary function symbols,
- $\mathtt{s}$ a unary function symbol,
- $\mathtt{q}, \mathtt{q}'$ two constant symbols,
- $\mathtt{AUX}_k$ and $\mathtt{HEAD}$ two functions symbols of respective arity k and 1.

The intuition behind the encoding is that a configuration of a 2MFA+S(k + 1) processing an input can be seen as a closed term

$$\mathtt{c} \bullet \mathtt{d} \bullet \tau(\boxdot) \bullet \mathtt{q} \bullet \mathtt{AUX}_k(\mathtt{p}_{i_1}, \dots, \mathtt{p}_{i_k}) \bullet \mathtt{HEAD}(\mathtt{p}_j)$$

where the $\mathtt{p}_i$ are position constants representing the positions of the *main pointer* ($\mathtt{HEAD}(\mathtt{p}_j)$) and of the *auxiliary pointers* ($\mathtt{AUX}_k(\mathtt{p}_{i_1}, \dots, \mathtt{p}_{i_k})$); the symbol $\mathtt{q}$ represents the state the automaton is in; $\tau(\boxdot)$ represents the current stack; the symbol $\mathtt{d}$ represents the direction of the next move of the main pointer; the symbol $\mathtt{c}$ represents the symbol currently read by the main pointer.

When a configuration matches the right side of the flow, the transition is followed, leading to an updated configuration.

More precisely, the iterations of $O_M \bar{W}_{\mathtt{p}}$, the product of the encoding of M with a word representation, is observed. Let us now explain how the basic operations of M are simulated:

Moving the pointers. Looking back at the definition of the encoding of words (Definition 10) gives a new reading of the action of the representation of a word: it moves the main pointer in the required direction. From that perspective, the position holding the symbol $\star$ in Definition 10 allows to simulate the behavior of the endmarkers $\triangleright$ and $\triangleleft$.

On the other hand, the observation is not able to manipulate the position of pointers directly (remember observations are forbidden to use the position constants) but can change the direction symbol $\mathtt{d}$, rearrange pointers (hence changing which one is the main pointer) and modify its state and the symbol $\mathtt{c}$ accordingly. For instance, a flow of the form

$$\cdots \bullet \mathtt{AUX}_k(x, \dots, y_k) \bullet \mathtt{HEAD}(y_1) \leftharpoondown \cdots \bullet \mathtt{AUX}_k(y_1, \dots, y_k) \bullet \mathtt{HEAD}(x)$$

encodes the instruction "swap the main pointer and the first auxiliary pointer".

Note however that our model has no built-in way to remember the values of the auxiliary pointers—it remembers only their *positions* as arguments of $\mathtt{AUX}_k(\cdots)$—, but this can be implemented easily using additional states.

Handling the stack. Given a unary function symbol $\underline{\mathtt{b}}(\cdot)$ for each symbol b of the stack alphabet $B_{\boxdot}$, reading the stack, pushing and popping elements are easily implemented:

$$\cdots \bullet x \bullet \cdots \leftharpoondown \cdots \bullet \underline{\mathtt{b}}(x) \bullet \cdots \qquad \text{(Read } b \text{ and pop it)}$$

$$\cdots \bullet \underline{\mathtt{c}}(\underline{\mathtt{b}}(x)) \bullet \cdots \leftharpoondown \cdots \bullet \underline{\mathtt{b}}(x) \bullet \cdots \qquad \text{(Read } b \text{ and push } c\text{)}$$

Changing the state. Given a constant $\mathtt{q}$ for each state $\mathbf{q}$ of M, updating the state amounts to picking the right $\mathtt{q}$ and $\mathtt{q}'$ in the flow representing the transition.

Acceptance and rejection. The encoding of acceptance and rejection is slightly more delicate, as detailed in a previous article [5, 6.2.3.].

The basic idea is that acceptance in our model is defined as nilpotency, that is to say: the absence of loops. If no transition in the automaton can be fired, then no flow in our encoding can be unified, and the computation ends.

Conversely, a loop in the automaton will refrain the wiring from being nilpotent. Loops should be represented as a re-initialization of the computation, so

that the observation performs forever the same computation when rejecting the input. Another encoding may interfere with the representation of acceptation as termination: an "in-place loop" triggered when reaching a particular state would make the observation cyclic, hence preventing the observation from being nilpotent *no matter the word representation processed*.

Indeed, the "loop" in Definition 17 of pushdown automata is to be read as "perform forever the same computation". □

Observations resulting from encoding pushdown automata are sums of flows of a particular form (shown at the beginning of the preceding proof). However, using general observations with stack, not constrained in this way, does not increase the expressive power: the next section is devoted to prove that the language recognized by any observation with stack lies in Ptime.

4 Nilpotency in $\mathcal{S}tack$ and Ptime soundness

We now introduce the *saturation* technique, which allows to decide nilpotency of $\mathcal{S}tack$ elements in polynomial time. This technique relies on the fact that in certain cases, the height of flows does not grow when computing their product. It adapts memoization [32] to our setting: we repeatedly extend the wiring by adding pairwise products of flows, allowing for more and more "transitions".

Remark 2. As pointed out by a reviewer of a previous version of this work, deciding the nilpotency of a unary flow is reminiscent of the problem of acyclicity for the configuration graph of a pushdown system (PDS) [13], a problem known to lie in Ptime [15]. However, our algorithm treats every state of the corresponding PDS as initial, and would detect cycles even in non-connected components: our problem is probably closer to the "uniform halting problem" [34], a problem known to be decidable [14, p. 10]. Whether this last problem, equivalent to deciding the *nœthériennité* of a finite system rewriting suffix words, is known to lie in Ptime, and if our Theorem 5 entails that bound, are both unknown to us.

Notation 18. Let τ and σ be sequences of unary function symbols. If $h\big(\tau(x)\big) \geq h\big(\sigma(x)\big)$ (reps. $h\big(\tau(x)\big) \leq h\big(\sigma(x)\big)$), we say that $\tau(x) \leftharpoondown \sigma(x)$ is *increasing* (resp. *decreasing*).

A wiring in $\mathcal{S}tack$ is *increasing* (resp. *decreasing*) if it contains only increasing (resp. *decreasing*) unary flows.

Lemma 1 (stability of height). *Let τ and σ be sequences of unary function symbols. If τ is decreasing and σ is increasing, then $h(\tau\sigma) \leq \max\{h(\tau), h(\sigma)\}$.*

With this lemma in mind, we can define a *shortcut* operation that augments an element of $\mathcal{S}tack$ by adding new flows while keeping the maximal height unchanged. Iterating this operation, we obtain a *saturated* version of the initial wiring, containing shortcuts, shortcuts of shortcuts, etc. In a sense we are designing an *exponentiation by squaring* procedure for elements of $\mathcal{S}tack$, the algebraic reading of memoization in our context.

Definition 19 (saturation). *If $F \in \mathcal{S}tack$ we define its increasing $F^{\uparrow} \coloneqq \{ f \in F \mid f \text{ is increasing} \}$ and decreasing $F^{\downarrow} \coloneqq \{ f \in F \mid f \text{ is decreasing} \}$ subsets. We set the* shortcut *operation $\mathtt{short}(F) \coloneqq F + F^{\downarrow}F^{\uparrow}$ and its least fixpoint, which we call the* saturation *of F: $\mathtt{satur}(F) \coloneqq \sum_{n\in\mathbb{N}} \mathtt{short}^n(F)$ (where $\mathtt{short}^n$ denotes the n^{th} iteration of $\mathtt{short}$).*

The point of this operation is that it is computable in Ptime (the fixpoint is reached in polynomial time) because of Lemma 1. This leads to a Ptime decision procedure for nilpotency of elements of $\mathcal{S}tack$.

Theorem 5 (nilpotency is in Ptime). *Given any integer h, there is a procedure $\textsc{Nilp}_h(\cdot) \in$ Ptime that, given a $F \in \mathcal{S}tack$ such that $\hbar(F) \leq h$ as an input, accepts iff F is nilpotent.*

Proof (sketch [8, theorem IV.15]). This relies on the fact that $\mathtt{satur}(\cdot)$ is computable in polynomial time and that the cyclicity of F and that of $\mathtt{satur}(F)$ are related. More precisely F is cyclic iff either $\mathtt{satur}(F)^{\uparrow}$ or $\mathtt{satur}(F)^{\downarrow}$ is. Finally one has to see that the case of increasing or decreasing wirings is easy to treat by discarding the bottom of large stacks, which is harmless in that case. □

The saturation technique can then be used to show that the language recognized by an observation with stack always belongs to the class Ptime. The important point in the proof is that, given an observation O and a representation $\bar{W}_p$ of a word W, one can produce in polynomial time an element of $\mathcal{S}tack$ whose nilpotency is equivalent to the nilpotency of $O\bar{W}_p$.

Proposition 1. *Let $O \in \mathcal{O}^{\mathbf{b}+\mathbf{s}}$ be an observation with stack. There is a procedure $\textsc{Red}_O(\cdot) \in$ FPtime that, given a word W as an input, outputs a wiring $F \in \mathcal{S}tack$ with $\hbar(F) \leq \hbar(O)$ such that F is nilpotent iff $O\bar{W}_{\mathtt{p}}$ is for any choice of $\mathtt{p}$.*

This is done essentially by remarking that the "balanced" part of the computation can never step outside a finite computation space, so that one can associate to each configuration a unary function symbol that is put on top of the stack.

Theorem 6 (soundness). *If $O \in \mathcal{O}^{\mathbf{b}+\mathbf{s}}$, then $\mathcal{L}(O) \in$ Ptime.*

5 Unary Logic Programming

In previous sections, we showed how the $\mathcal{S}tack$ semiring captures polynomial time computation. As we already mentioned, the elements of this semiring correspond to a specific class of logic programs, so that our results have a reading in terms of complexity of logic programming [22] which we detail now.

Definition 20 (data, goal, query). *A* unary query *is $\mathbf{Q} = (D, P, G)$, where:*
- *D is a set of closed unary terms (a* unary data*),*
- *P is a an element of $\mathcal{S}tack$ (a* unary program*),*
- *G is a closed unary term (a* unary goal*).*

We say that the query $\mathbf{Q}$ succeeds *if* $G \dashv$ *can be derived combining* $d \dashv$ *with* $d \in D$ *and the elements of* P *by the resolution rule presented in Sect. 2.1, otherwise we say the query* fails. *The* size $|\mathbf{Q}|$ *of the query is defined as the total number of occurrences of symbols in it.*

To apply the saturation technique directly, we need to represent all the elements of the unary query (data, program, goal) as elements of $\mathcal{S}tack$. This requires a simple encoding.

Definition 21 (encoding unary queries). *We suppose that for any constant symbol* $\mathtt{c}$*, we have a unary function symbol* $\underline{\mathtt{c}}(\cdot)$*. We also need two unary functions,* $\underline{\mathtt{START}}(\cdot)$ *and* $\underline{\mathtt{ACCEPT}}(\cdot)$*. To any unary data* D *we associate an element of* $\mathcal{S}tack$*:* $[D] \coloneqq \{\, \tau(\underline{\mathtt{c}}(x)) \leftharpoondown \mathtt{START}(x) \mid \tau(\mathtt{c}) \in D \,\}$ *and to any unary goal* $G = \tau(\mathtt{c})$ *we associate* $\langle G \rangle \coloneqq \mathtt{ACCEPT}(x) \leftharpoondown \tau(\underline{\mathtt{c}}(x))$*.*

Note that the program part P of the query needs not to be encoded as it is already an element of $\mathcal{S}tack$. Once a query is encoded, we can tell if it is successful or not using the language of the resolution semiring.

Lemma 2 (success). *A unary query* $\mathbf{Q} = (D, P, G)$ *succeeds if and only if* $\mathtt{ACCEPT}(x) \leftharpoondown \mathtt{START}(x) \in \langle G \rangle P^n[D]$ *for some* n*.*

The saturation technique then can be applied to unary queries adding to new shortcut rules which eventually allow to decide acceptance.

Lemma 3 (saturation of unary queries). *A unary query* $\mathbf{Q} = (D, P, G)$ *succeeds if and only if* $\mathtt{ACCEPT}(x) \leftharpoondown \mathtt{START}(x) \in \mathit{satur}\big([D] + P + \langle G \rangle\big)$*.*

Theorem 7 (Ptime-completeness). *The* UQuery *problem (given a unary query, is it successful?) is* Ptime*-complete.*

Proof. The lemma above, combined with the fact that $\mathit{satur}(\cdot)$ is computable in polynomial time,[6] ensures that the problem lies indeed in the class Ptime. The hardness part follows from a variation on the encoding presented in Sect. 3 and the reduction derived from Proposition 1. □

Remark 3. We presented the result in a restricted form to stay in line with the previous sections. However, it should be clear to the reader that it would not be impacted if we allowed: non-closed goals and data; programs with no restriction on variables, e.g. $\mathtt{f}(x) \leftharpoondown \mathtt{g}(y)$; constants in the program part of the query.

Remark 4. In terms of complexity of logic programs, we are considering the *combined complexity* [22, p. 380]: every part of the query $\mathbf{Q} = (D, P, G)$ is variable. If for instance we fixed P and G (thus considering *data complexity*), we would have a problem that is still in Ptime, but it is unclear to us if it would be complete. Indeed, the encoding of Sect. 3 relies on a representation of inputs as plain programs, and on the fact that the evaluation process is a matter of interaction between programs rather than mere data processing.

[6] The bound on the running time of the procedure computing $\mathit{satur}(\cdot)$ being exponential in the height, one needs to first process the query into an equivalent one using only terms of bounded height, which can easily be done in polynomial time

6 Perspectives

Adding a "stack plugin" to observations extends modularly previous works [2,3,5,6] and gives the perfect tool to characterize Ptime. This modularity was inspired by the classical addition of a stack to an automaton, allowing to switch from Logspace to Ptime, and providing a decisive proof technique: memoization—or exponentiation by squaring in our context—implemented as saturation. The automata's qualitative constraint on memory is directly represented as a syntactic restriction on flows.

In this setting, evaluation is inspired by the interactive approach to the Curry-Howard correspondence—geometry of interaction—, which makes the complexity parametric in the program *and* the input. This mechanism of computation differs from automata's step-by-step evaluation, but that does not prevent the simulation of pushdown automata by unary logic program.

The mechanism of pre-computation of transitions, known as memoization, was adapted in a setting where logic programs are represented as algebraic objects. This *saturation technique* computes shortcuts in a logic program to decide its nilpotency in polynomial time. As it turns out, this is similar to the techniques employed to solve efficiently the problem of termination of pushdown systems.

More generally, this approach to complexity is based either on operator algebra [31,5,6] or unification theory [8,2,3]: it is emerging as a meeting point for computer science, logic and mathematics, and raises a number of perspectives.

A number of interrogations emerges naturally when considering the relations to proof theory. First, we could consider the Church encoding of other data types—trees for instance—and define "orthogonally" set of programs interacting with them, wondering what their computational nature is. In the distance, one may hope for a connection between our approach and ongoing work on higher order trees and model checking [33]; all alike, one could study the interaction between observations and one-way integers—briefly discussed in earlier work [3]—or non-deterministic data. Second, a still unanswered question of interest is to give proof-terms representation of captured programs, i.e. observations.

Finally, it should be possible to represent functional computation (and not only decision problems, i.e. to switch from Ptime to FPtime), by considering a more general notion of observation that could express what an output is. In that perspective, a good place to start should be to show that light logics characterization results [9] can be recovered via our methods, which seems very likely but remains to be precisely investigated.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Time and tape complexity of pushdown automaton languages. Inform. Control 13(3), 186–206 (1968)
2. Aubert, C., Bagnol, M.: Unification and logarithmic space. In: Dowek, G. (ed.) RTA-TLCA. LNCS, vol. 8650, pp. 77–92. Springer (2014)
3. Aubert, C., Bagnol, M., Pistone, P., Seiller, T.: Logic programming and logarithmic space. In: Garrigue, J. (ed.) APLAS. LNCS, vol. 8858, pp. 39–57. Springer (2014)
4. Aubert, C., Bagnol, M., Seiller, T.: Memoization for unary logic programming: Characterizing ptime. Research Report RR-8796, INRIA (2015), `https://hal.archives-ouvertes.fr/hal-01107377`
5. Aubert, C., Seiller, T.: Characterizing co-NL by a group action. MSCS (FirstView) pp. 1–33 (Dec 2014)
6. Aubert, C., Seiller, T.: Logarithmic space and permutations. Inf. Comput., Special Issue on Implicit Computational Complexity (2015)
7. Baader, F., Nipkow, T.: Term rewriting and all that. CUP (1998)
8. Bagnol, M.: On the Resolution Semiring. Ph.D. thesis, Aix-Marseille Université – Institut de Mathématiques de Marseille (2014), `https://hal.archives-ouvertes.fr/tel-01215334`
9. Baillot, P., Mazza, D.: Linear logic by levels and bounded time complexity. Theoret. Comput. Sci. 411(2), 470–503 (2010), `http://arxiv.org/pdf/0801.1253v3`
10. Baillot, P., Pedicini, M.: Elementary complexity and geometry of interaction. Fund. Inform. 45(1–2), 1–31 (2001)
11. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: LICS. pp. 266–275. IEEE Computer Society (2004)
12. Bellantoni, S.J., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Comput. Complex. 2, 97–110 (1992)
13. Carayol, A., Hague, M.: Saturation algorithms for model-checking pushdown systems. In: Ésik, Z., Fülöp, Z. (eds.) Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014. EPTCS, vol. 151, pp. 1–24 (2014)
14. Caucal, D.: Récritures suffixes de mots. Research Report RR-0871, INRIA (1988), `https://hal.inria.fr/inria-00075683`
15. Caucal, D.: On the regular structure of prefix rewriting. In: Arnold, A. (ed.) CAAP. LNCS, vol. 431, pp. 87–102. Springer (1990)
16. Cook, S.A.: Characterizations of pushdown machines in terms of time-bounded computers. J. ACM 18(1), 4–18 (1971)
17. Cook, S.A.: Linear time simulation of deterministic two-way pushdown automata. In: IFIP Congress (1). pp. 75–80. North-Holland (1971)
18. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI. LNCS, vol. 7388, pp. 89–109. Springer (2011)
19. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In: Gordon, A.D. (ed.) ESOP. LNCS, vol. 6012, pp. 205–225. Springer (2010)
20. Danos, V.: La Logique Linéaire appliquée à l'étude de divers processus de normalisation (principalement du λ-calcul). Ph.D. thesis, Université Paris VII (1990)
21. Danos, V., Joinet, J.B.: Linear logic & elementary time. Inf. Comput. 183(1), 123–137 (2003)
22. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. 33(3), 374–425 (2001)

23. Dwork, C., Kanellakis, P.C., Mitchell, J.C.: On the sequential nature of unification. J. Log. Program. 1(1), 35–50 (1984)
24. 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)
25. Girard, J.Y.: Linear logic. Theoret. Comput. Sci. 50(1), 1–101 (1987)
26. Girard, J.Y.: Geometry of interaction 1: Interpretation of system F. Studies in Logic and the Foundations of Mathematics 127, 221–260 (1989)
27. Girard, J.Y.: Towards a geometry of interaction. In: Gray, J.W., Scedrov, A. (eds.) Proceedings of the AMS Conference on Categories, Logic and Computer Science. Categories in Computer Science and Logic, vol. 92, pp. 69–108. AMS (1989)
28. 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)
29. Girard, J.Y.: Light linear logic. In: Leivant, D. (ed.) LCC, LNCS, vol. 960, pp. 145–176. Springer (1995)
30. Girard, J.Y.: Geometry of interaction V: logic in the hyperfinite factor. Theoret. Comput. Sci. 412(20), 1860–1883 (Apr 2011)
31. 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)
32. Glück, R.: Simulation of two-way pushdown automata revisited. In: Banerjee, A., Danvy, O., Doh, K.G., Hatcliff, J. (eds.) Festschrift for Dave Schmidt. EPTCS, vol. 129, pp. 250–258 (2013)
33. Grellois, C., Melliès, P.A.: Relational semantics of linear logic and higher-order model checking. In: Kreutzer, S. (ed.) CSL. LIPIcs, vol. 41, pp. 260–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015), http://www.dagstuhl.de/dagpub/978-3-939897-90-3
34. Huet, G., Lankford, D.: On the uniform halting problem for term rewriting systems. Research Report RR-283, INRIA (1978), http://www.ens-lyon.fr/LIP/REWRITING/TERMINATION/Huet_Lankford.pdf
35. Ladermann, M., Petersen, H.: Notes on looping deterministic two-way pushdown automata. Inf. Process. Lett. 49(3), 123–127 (1994)
36. Lafont, Y.: Soft linear logic and polynomial time. Theoret. Comput. Sci. 318(1), 163–180 (2004)
37. Leivant, D.: Stratified functional programs and computational complexity. In: Van Deusen, M.S., Lang, B. (eds.) POPL. pp. 325–333. ACM Press (1993)
38. Neergaard, P.M.: A functional language for logarithmic space. In: Chin, W.N. (ed.) APLAS, LNCS, vol. 3302, pp. 311–326. Springer (2004)
39. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (Jan 1965)
40. Seiller, T.: Logique dans le facteur hyperfini : géometrie de l'interaction et complexité. Ph.D. thesis, Université de la Méditerranée (2012), https://hal.archives-ouvertes.fr/tel-00768403
41. Seiller, T.: A correspondence between maximal abelian sub-algebras and linear logic fragments. ArXiv preprint abs/1408.2125 (2014), to appear in MSCS
42. Seiller, T.: Interaction graphs: Graphings. ArXiv preprint abs/1405.6331 (2014)
43. Wagner, K.W., Wechsung, G.: Computational Complexity, Mathematics and its Applications, vol. 21. Springer (1986)