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] Proofs, Programs, Systems. CNRS and Paris Diderot University

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), a construction stemming from an interactive interpretation of the cut-elimination procedure of linear logic known as the *geometry of interaction*.
We restrict this framework to terms (logic programs, rewriting rules) using only unary symbols and prove the restriction complete for polynomial time computation using an encoding of pushdown automata. Soundness w.r.t. PTIME is proven thanks to a saturation method close the one used for pushdown systems and inspired by the memoization technique.
As a direct consequence, we get a PTIME-completeness result for a class of logic programming queries that uses only unary function symbols.

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

1 Introduction

Complexity theory focuses on resource usage of computer programs, such as the amount of time or memory a given program will need to solve a problem. 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.*" [16, p. 90] This has been already achieved via different approaches, for instance by considering restricted programming languages or computational principles [10,36,37].

* 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 and the ANR 12 JS02 006 01 project COQUAS.

A number of results in this area also arose from proof theory, through the study of subsystems of linear logic [24]. More precisely, the Curry-Howard —or *proofs as programs*—correspondence expresses a close relation between formal proofs and typed programs. In this approach, one sets a formula `Nat` which corresponds to the type of binary integers, and proofs of the formula `Nat` ⇒ `Nat` then correspond to algorithms computing functions from integers to integers, via the cut-elimination procedure. By considering restricted subsystems, one allows *less* proofs of type `Nat` ⇒ `Nat`, hence *less* algorithms can be implemented, and the class of accepted proofs/programs may correspond[4] to some complexity class: elementary complexity [28,19], polynomial time [35,9], logarithmic [17] and polynomial [23] space.

More recently, new methods for obtaining implicit characterizations of complexity classes based on the *geometry of interaction* (GoI) research program [26] 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 [18,39] or generalizations of graphs [41], bounded linear maps between Hilbert spaces (operators) [25,29,40], clauses (or "flows") [27,7]. 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 [17]).

The idea is not to rely on a restriction of some type system, but rather on conditions on the shape of program representations considered. 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 [30,4,5]. We consider a more syntactic flavor of the GoI interpretation where untyped programs are represented in the so-called *resolution semiring* [7], a semiring based on the resolution rule [38] and a specific class of logic programs. This setting presents some advantages: it avoids the involvement of operator algebras theory, eases the discussions in terms of complexity (we manipulate first-order terms, which have natural notions of size, height, etc.) and offers a straightforward connection with complexity of logic programming [21]. 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.

Our main contribution here is a characterization of the class Ptime by studying a natural restriction, namely that one is allowed to use exclusively *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. Soundness follows from a variation of the saturation algorithm for pushdown systems [11], inspired by S. Cook's memoization technique [15]

[4] We mean *extensional* correspondence: they can compute the same functions.

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, our results have a simple formulation and provide an original point of view on this complexity class. They also constitute the first characterization of a time-complexity class directly on the semantic side of the GOI interpretation.

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 of the paper

We begin by giving in Sect. 2.1 the formal definition of the resolution semiring; then briefly explain how words and programs can be represented in this structure (Sect. 2.2). In Sect. 2.3 we introduce the restricted semiring that will be under study in this paper: the $\mathcal{S}tack$ semiring.

The next two sections are respectively devoted to the completeness and soundness results for PTIME. For completeness, we first review multi-head finite automata with pushdown stack, that characterize PTIME, and then show how 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).

We finally show how this results implies the PTIME-completeness of unary logic programming queries (Sect. 5).

Sketched proofs are detailed in the Appendix.

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 $\theta\mathrm{t}$ 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 are concerned with formal solving of equalities $\mathrm{t} \doteq \mathrm{u}$ between terms. Let us make this precise and introduce some vocabulary.

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 an instance of* 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* $\mathit{var}(\mathrm{t}') \cap \mathit{var}(\mathrm{u}') = \varnothing$*;*
- 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 [22, 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 $H \dashv B_1, \dots, B_n$):

$$\dfrac{V \dashv T_1, \dots, T_n \qquad \mathtt{var}(U) \cap \mathtt{var}(V) = \varnothing \qquad H \dashv B_1, \dots, B_m, U \qquad \theta \text{ is a MGU of } U \text{ and } V}{\theta H \dashv \theta B_1, \dots, \theta B_m, \theta T_1, \dots, \theta T_n} \ \mathtt{Res}$$

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 appearing 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* $f = \mathrm{t} \leftharpoonup \mathrm{u}$*, with* $\mathit{var}(\mathrm{t}) \subseteq \mathit{var}(\mathrm{u})$*. 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 $f_1 = \mathtt{h}(x) \leftharpoonup \mathtt{g}(x)$ and $f_2 = \mathtt{g}(\mathtt{f}(x)) \leftharpoonup \mathtt{f}(x)$ 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* $\mathit{var}(\mathrm{u}) \cap \mathit{var}(\mathrm{v}) = \varnothing$*.*

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* $(\mathrm{t} \leftharpoonup \mathrm{u})(\mathrm{v} \leftharpoonup \mathrm{w}) = \theta\mathrm{t} \leftharpoonup \theta\mathrm{w}$*.*

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* $FG = \{ fg \mid f \in F, g \in G, fg \text{ defined} \}$. *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: $\{ f_1, \dots, f_n \} = f_1 + \cdots + f_n$.
- We denote by 0 the empty set, i.e. the unit of $+$.
- The unit for the product is the wiring $I = x \leftharpoondown x$.

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

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

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

This classical notion from abstract algebra has a specific reading in our case of study. 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 the point of view of rewriting, it means that the set of rewriting rules we consider is terminating with a uniform bound on the length of rewriting chains —again, we consider rewriting at the root of terms, while the usual notion from term rewriting systems [6] allows in-context rewriting.

2.2 Representation of Words and Programs

This section explains and motivates the representation of words as flows. By studying their interaction with wirings from a specific semiring, we will be able to define notions of program and accepted language. First, let us see how the binary function symbol $\bullet$ used to construct terms can be extended to build flows and then semirings.

[5] A *semiring* is a set R equipped with two operations $+$ (the sum) and $\times$ (the product, whose symbol is usually omitted), and an element $0 \in R$ such that $(R, +, 0)$ is a commutative monoid; $(R, \times)$ is a *semigroup*, i.e. a monoid which may not have a neutral element; the product distributes over the sum; and the element 0 is absorbent: $0r = r0 = 0$ for all $r \in R$.

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, 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 $\mathrm{t} \leftharpoonup \mathrm{u} + \mathrm{u} \leftharpoonup \mathrm{t}$.

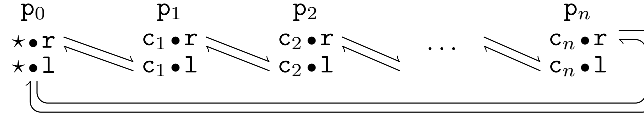
Definition 10 (word representation). *From now on, we suppose fixed an infinite set of constant symbols* $\mathtt{P}$ *(the* position constants*) and a finite alphabet* Σ *disjoint from* $\mathtt{P}$ *with* $\star \notin \Sigma$ *(we write* Σ^* *the set of words over* Σ*).*

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:*

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

In this definition, the position constants 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* from a symbol to the next or the previous (hence the choice of symbols $\mathtt{l}$ and $\mathtt{r}$ for "left/previous" and "right/next") for some kind of automaton reading the input. More details on this point of view will be given in the proof of Theorem 4. Hence, for a given position constant $\mathtt{p}_i$, we use terms $\mathtt{c}_i \bullet \mathtt{r}$ and $\mathtt{c}_i \bullet \mathtt{l}$ which will be linked (by flows of the representation) to elements $\mathtt{c}_{i+1} \bullet \mathtt{l}$ at position $\mathtt{p}_{i+1}$ and $\mathtt{c}_{i-1} \bullet \mathtt{r}$ at position $\mathtt{p}_{i-1}$ respectively.

Note moreover that the representation of the input is circular (this is a consequence of using the Church encoding of words), as we take $\mathtt{c}_{n+1} = \mathtt{c}_0 = \star$. 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 here is that an observation is forbidden to use any position constant, so that it interacts the same way with all the representations $\bar{W}_p$ 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* $\mathcal{O} = (\Sigma_{\mathtt{lr}} \bullet \mathcal{R}) \cap \mathtt{P}^{\perp}$, *and the semiring of* observations over the semiring $\mathcal{A}$ *as* $\mathcal{O}[\mathcal{A}] = (\Sigma_{\mathtt{lr}} \bullet \mathcal{A}) \cap \mathtt{P}^{\perp}$.

The following property is a consequence of the fact that observations cannot use position constants [7, 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* $\mathbf{Nil}(O\bar{W}_q)$.

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

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

In previous work [3], we 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* $x \in \boldsymbol{var}(\mathrm{t}) \cup \boldsymbol{var}(\mathrm{u})$*, all occurrences of* x *in both* t *and* u *have the same height (recall notations p. 3). A* balanced *wiring* F *is a sum of balanced flows.*

We write $\mathcal{R}_{\mathbf{b}}$ *for the set of balanced wirings.*

Definition 14 (balanced observation). *A* balanced observation *is an element of* $\mathcal{O}[\,\mathcal{R}_{\mathbf{b}} \bullet \mathcal{R}_{\mathbf{b}}\,]$.

This natural restriction was shown to characterize (non-deterministic) logarithmic space computation [3, Theorems 34-35], with a natural subclass of balanced wirings corresponding to deterministic logarithmic space computable predicates.

2.3 The *Stack* Semiring

This paper deals with another restriction on flows, namely the restriction to *unary flows*, defined with unary function symbols only. The semiring of wirings composed only of unary flows is called the *Stack* semiring, and will be shown to give a characterization of polynomial time computation. Here we briefly give the definitions and results about this semiring that will be needed in this paper. A more complete picture can be found in the second author's Ph.D. thesis [7].

Definition 15 (unary flows). *A* unary flow *is a flow built using only unary function symbols and a variable.*

The semiring Stack *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 in the proof of the characterization of polynomial time computation. As we will see, it is complementary to the nilpotency property for elements of *Stack*, i.e. a wiring in *Stack* will be either cyclic or nilpotent.

Definition 16 (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 $f^2 = \mathtt{c} \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet \mathtt{d} \neq 0$ (by Remark 1), 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 [7, theorem II.52]). The fact that when F not nilpotent is acyclic is a consequence of a result bounding the height of elements of F^n when F is acyclic, from an earlier work on GoI and complexity [8]. From this result, 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.

The converse is an immediate consequence of Remark 1. ☐

Example 2. Let us have a look at a nilpotent element of $\mathcal{S}tack$, displaying how the nilpotency problem can be tricky to solve efficiently.

Consider the wiring

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

which implements a sort of counter from 0 to 7 in binary notation that resets to 0 when it reaches 8 (we see the sequence $\mathtt{f}_x\mathtt{f}_y\mathtt{f}_z$ as the integer $x + 2y + 4z$). 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 $2^n - 1$, we obtain an example where the number of iterations needed to find a cycle is exponential in the size of F, which rules out a polynomial time decision procedure for the nilpotency problem that would simply compute the iterations of F until it finds a cycle in it.

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

Definition 17. *A* balanced observation with stack *is an element of the semiring* $\mathcal{O}^{\mathbf{b+s}} = \mathcal{O}[\,\mathcal{S}tack \bullet \mathcal{R}_{\mathbf{b}}\,]$.

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, one gets 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 they can be represented as observations.

Definition 18 (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 $b \in B$,* **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" [34, 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 [3,5] the characterization of LOGSPACE and NLOGSPACE by 2-way k-head finite automata *without* pushdown stacks [42, 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 [14, pp. 9–11] as well as a precise algorithm [42, 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 [15]. □

We turn now to 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 [3, Sect. 4.1] encoding of 2-way k-head finite automata (*without* pushdown stack) by flows. The only difference is the addition of a "plug-in" that allows for a representation of stacks in observations.

Remember that acceptance by observations is phrased in terms of nilpotency of the product $O\bar{W}_p$ 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 $O\bar{W}_p, (O\bar{W}_p)^2, (O\bar{W}_p)^3, \dots$ and the computation either ends at some point (i.e. accepts)—that is $(O\bar{W}_p)^n = 0$ for some integer n—or loops (i.e. rejects). One can think of this iteration as representing a dialogue, or a game, between the observation and its input: the *interaction* between the observation and the word representation is what simulates the behavior of the automaton, and not the observation on its own manipulating some passive data.

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 we will encode its transition relation as a balanced observation with stack (Definition 17). 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

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

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,
- AUX_k and 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

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

where the $\mathtt{p}_i$ are position constants representing the positions of the *main pointer* ($\mathrm{HEAD}(\mathtt{p}_j)$) and of the *auxiliary pointers* ($\mathrm{AUX}_k(\mathtt{p}_{i_1}, \dots, \mathtt{p}_{i_k})$); 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, we will observe the iterations of $O_M \bar{W}_p$, the product of the encoding of M with a word representation. 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), we see that we can have a new reading of what the representation of a word does: 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

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

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. Suppose we have 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. We suppose that we have a constant $\mathtt{q}$ for each state $\mathbf{q}$ of M. Then, 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 [4, 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. The point we need to be careful about is the encoding of loops: those 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 18 of pushdown automata is to be read as "perform forever the same computation". □

As we see, 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 [31] to our setting: we repeatedly extend the wiring by adding pairwise products of flows, allowing for more and more "transitions" at each step.

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) [11], a problem known to lie in Ptime [13]. 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" [33]. This problem, equivalent to deciding the *nœthériennité* of a finite system rewriting suffix words, is known to be decidable [12, p. 10], but whether it was already proven to be in Ptime is unknown to us.

Notation 19. Let τ and σ be sequences of unary function symbols.

If $\hbar\big(\tau(x)\big) \geq \hbar\big(\sigma(x)\big)$ we say that $\tau(x) \leftharpoonup \sigma(x)$ is *increasing*.

If $\hbar\big(\tau(x)\big) \leq \hbar\big(\sigma(x)\big)$ we say that $\tau(x) \leftharpoonup \sigma(x)$ is *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 $f = \tau(x) \leftharpoonup \sigma(x)$ and $g = \rho(x) \leftharpoonup \chi(x)$ be stack operations. If f is decreasing and g is increasing, we have $\hbar(fg) \leq \max\{\hbar(f), \hbar(g)\}$.*

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 20 (saturation). *If $F \in \mathcal{S}tack$ we define its increasing $F^{\uparrow} = \{ f \in F \mid f \text{ is increasing} \}$ and decreasing $F^{\downarrow} = \{ f \in F \mid f \text{ is decreasing} \}$ subsets. We set the* shortcut *operation $\mathtt{short}(F) = F + F^{\downarrow}F^{\uparrow}$ and its least fixpoint, which we call the* saturation *of F: $\mathtt{satur}(F) = \sum_{n\in\mathbb{N}} \mathtt{short}^n(F)$ (where $\mathtt{short}^n$ denotes the nth 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 the stability of height lemma just above. 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 $\text{NILP}_h(\cdot) \in$ 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 [7, 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 $O \in \mathcal{O}^{\mathbf{b+s}}$ be an observation with stack. There is a procedure $\text{RED}_O(\cdot) \in$ 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 $O\bar{W}_p$ 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 $O \in \mathcal{O}^{\mathbf{b+s}}$ is an observation with stack, 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 [21] which we detail now.

Definition 21 (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 22 (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* $\mathcal{S}tack$*:* $[D] = \{\ \tau(\underline{\mathtt{c}}(x)) \leftharpoondown \mathtt{START}(x) \mid \tau(\mathtt{c}) \in D\ \}$ *and to any unary goal* $G = \tau(\mathtt{c})$ *we associate* $\langle G \rangle = \mathtt{ACCEPT}(x) \leftharpoondown \tau(\underline{\mathtt{c}}(x))$*.*

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* $\mathtt{ACCEPT}(x) \leftharpoondown \mathtt{START}(x) \in \boldsymbol{\mathit{satur}}\big([D] + P + \langle G \rangle\big)$*.*

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 $\boldsymbol{\mathit{satur}}(\cdot)$ 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 this construction 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* [21, 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 $\boldsymbol{\mathit{satur}}(\cdot)$ 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

This article extends modularly on our previous work [2,3,4,5] to obtain a characterization of Ptime, by adding a sort of "stack plugin" to observations. This enhancement was guided by the intuition of a stack added to an automaton, allowing to move from Logspace to Ptime and providing a decisive proof technique: memoization (or exponentiation by squaring in our context) implemented as saturation.

We saw that to a qualitative constraint on the way memory is handled by automata corresponds a syntactic restriction on flows. These flows are evaluated in a setting inspired by the representation of inputs in the interactive approach to the Curry-Howard correspondence—geometry of interaction—, which makes the complexity parametric in the program *and* the input. However, despite the evaluation being highly parallel and different from the step-by-step evaluation performed by automata, a precise simulation of pushdown automata by unary logic program is given, leading to complexity results.

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

The approach to complexity under study here can be based either on operator algebra [30,4,5] or unification theory [7,2,3]: it is emerging as a meeting point for computer science, logic and mathematics. This raises multiple questions and 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 [32]; 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 an account of observations in terms of a proof-system.

One could also investigate possible relations with other models of computation, such as the interaction abstract machine [20] or pushdown systems, that already developed and used the notion of shortcut in the evaluation.

Finally, it should be possible to represent functional computation (and not only decision problems), by considering a more general notion of observation that would allow for expressing the notion of output.

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., Seiller, T.: Characterizing co-NL by a group action. MSCS (FirstView) pp. 1–33 (Dec 2014)
5. Aubert, C., Seiller, T.: Logarithmic space and permutations. Inf. Comput., Special Issue on Implicit Computational Complexity (2015)
6. Baader, F., Nipkow, T.: Term rewriting and all that. CUP (1998)
7. 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
8. Baillot, P., Pedicini, M.: Elementary complexity and geometry of interaction. Fund. Inform. 45(1–2), 1–31 (2001)
9. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: LICS. pp. 266–275. IEEE Computer Society (2004)
10. Bellantoni, S.J., Cook, S.A.: A new recursion-theoretic characterization of the polytime functions. Comput. Complex. 2, 97–110 (1992)
11. 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)
12. Caucal, D.: Récritures suffixes de mots. Research Report RR-0871, INRIA (1988), https://hal.inria.fr/inria-00075683
13. Caucal, D.: On the regular structure of prefix rewriting. In: Arnold, A. (ed.) CAAP. LNCS, vol. 431, pp. 87–102. Springer (1990)
14. Cook, S.A.: Characterizations of pushdown machines in terms of time-bounded computers. J. ACM 18(1), 4–18 (1971)
15. Cook, S.A.: Linear time simulation of deterministic two-way pushdown automata. In: IFIP Congress (1). pp. 75–80. North-Holland (1971)
16. 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)
17. Dal Lago, U., Schöpp, U.: Functional programming in sublinear space. In: Gordon, A.D. (ed.) ESOP. LNCS, vol. 6012, pp. 205–225. Springer (2010)
18. 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)
19. Danos, V., Joinet, J.B.: Linear logic & elementary time. Inf. Comput. 183(1), 123–137 (2003)
20. Danos, V., Regnier, L.: Reversible, irreversible and optimal lambda-machines. Theoret. Comput. Sci. 227(1-2), 79–97 (1999)
21. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Comput. Surv. 33(3), 374–425 (2001)
22. Dwork, C., Kanellakis, P.C., Mitchell, J.C.: On the sequential nature of unification. J. Log. Program. 1(1), 35–50 (1984)

23. 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)
24. Girard, J.Y.: Linear logic. Theoret. Comput. Sci. 50(1), 1–101 (1987)
25. Girard, J.Y.: Geometry of interaction 1: Interpretation of system F. Studies in Logic and the Foundations of Mathematics 127, 221–260 (1989)
26. 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)
27. 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)
28. Girard, J.Y.: Light linear logic. In: Leivant, D. (ed.) LCC, LNCS, vol. 960, pp. 145–176. Springer (1995)
29. Girard, J.Y.: Geometry of interaction V: logic in the hyperfinite factor. Theoret. Comput. Sci. 412(20), 1860–1883 (Apr 2011)
30. 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)
31. 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)
32. 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
33. 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
34. Ladermann, M., Petersen, H.: Notes on looping deterministic two-way pushdown automata. Inf. Process. Lett. 49(3), 123–127 (1994)
35. Lafont, Y.: Soft linear logic and polynomial time. Theoret. Comput. Sci. 318(1), 163–180 (2004)
36. Leivant, D.: Stratified functional programs and computational complexity. In: Van Deusen, M.S., Lang, B. (eds.) POPL. pp. 325–333. ACM Press (1993)
37. Neergaard, P.M.: A functional language for logarithmic space. In: Chin, W.N. (ed.) APLAS, LNCS, vol. 3302, pp. 311–326. Springer (2004)
38. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM 12(1), 23–41 (Jan 1965)
39. 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
40. Seiller, T.: A correspondence between maximal abelian sub-algebras and linear logic fragments. ArXiv preprint abs/1408.2125 (2014)
41. Seiller, T.: Interaction graphs: Graphings. ArXiv preprint abs/1405.6331 (2014)
42. Wagner, K.W., Wechsung, G.: Computational Complexity, Mathematics and its Applications, vol. 21. Springer (1986)

A Proof of Soundness

We borrow a result of P. Baillot and M. Pedicini which requires the concept of *acyclic sequence* of unary flows.

Definition 23 (sequences). *For $\boldsymbol{s} = f_1, \dots, f_n$ a sequence of unary flows, define:*

- *its* height *as* $\boldsymbol{h}(\boldsymbol{s}) = \max_i \{\boldsymbol{h}(f_i)\}$
- *its* cardinality[7] $\mathrm{Card}(\boldsymbol{s}) = \mathrm{Card}\{ f_i \mid 1 \leq i \leq n \}$.
- *its product* $\boldsymbol{p}(\boldsymbol{s})$ *as* $f_1 \cdots f_n$.

We say the sequence $\boldsymbol{s}$ *is* cyclic *if there is a sub-sequence* $\boldsymbol{s}_{i,j} = f_i, \dots, f_j$ $(1 \leq i \leq j \leq n)$ *such that* $\boldsymbol{p}(\boldsymbol{s}_{i,j})$ *is a cycle.*

Lemma 4 (acyclic sequence [8, lemma 5.3]). *If* $\boldsymbol{s}$ *is an acyclic sequence of unary flows, then we have* $\boldsymbol{h}\big(\boldsymbol{p}(\boldsymbol{s})\big) \leq \boldsymbol{h}(\boldsymbol{s})(\mathrm{Card}(\boldsymbol{s}) + 1)$.

Proof of Lemma 1. If $fg = 0$, the property holds because $\boldsymbol{h}(0) = 0$. Otherwise, we have either $\sigma = \rho\mu$ or $\sigma\mu = \rho$. Suppose we are in the first case (the second being symmetric). Then we have $fg = \tau(x) \leftharpoonup \chi\mu(x)$ and $\boldsymbol{h}(\sigma) = \boldsymbol{h}(\rho\mu)$. As g is increasing, $\boldsymbol{h}(\chi) \leq \boldsymbol{h}(\rho)$ and therefore we have $\boldsymbol{h}(\chi\mu) \leq \boldsymbol{h}(\rho\mu) = \boldsymbol{h}(\sigma) \leq \boldsymbol{h}(f) \leq \max\{\boldsymbol{h}(f), \boldsymbol{h}(g)\}$. □

Now, as we are only manipulating flows with a limited height, the iteration of the shortcut operation is bound to stabilize at some point.

Proposition 2 (stability of saturation). *Let* $F \in \mathcal{S}tack$ *be a wiring and* S *the number of distinct function symbols appearing in* F.

For any n, *we have* $\boldsymbol{h}\big(\boldsymbol{short}^n(F)\big) = \boldsymbol{h}(F)$.

Moreover if $n \geq (S^{\boldsymbol{h}(F)} + S^{\boldsymbol{h}(F)-1} + \dots + 1)^2$ *then* $\boldsymbol{short}^n(F) = \boldsymbol{satur}(F)$.

Proof. By Lemma 1 we have

$$\boldsymbol{h}(F^{\downarrow}F^{\uparrow}) \leq \max\{\boldsymbol{h}(F^{\downarrow}), \boldsymbol{h}(F^{\uparrow})\} = \boldsymbol{h}(F)$$

Therefore $\boldsymbol{h}\left(\boldsymbol{short}(F)\right) = \boldsymbol{h}(F)$, and we get the first property by induction.

For any n, the elements of $\boldsymbol{short}^n(F)$ are unary flows of height at most $\boldsymbol{h}(F)$ built from the function symbols appearing in F, therefore $\boldsymbol{short}^n(F)$ is a subset of a set of cardinality $k = (S^{\boldsymbol{h}(F)} + S^{\boldsymbol{h}(F)-1} + \dots + 1)^2$. As $G \subseteq \boldsymbol{short}(G)$ for all G, the iteration of $\boldsymbol{short}(\cdot)$ on F must be stable after at most k steps. □

In the following, we let FPTIME be the class of functions computable by Turing machine in polynomial time. Here we need to specify how the size of a wiring is measured.

Definition 24 (size). *The* size $|F|$ *of a wiring* F *is defined as the total number of function symbol occurrences in it.*

[7] Note that the cardinality of $\boldsymbol{s}$ is not necessarily equal to the length of $\boldsymbol{s}$. For instance, if $\boldsymbol{s} = f_1, f_1, f_2$ with $f_1 \neq f_2$ then $\mathrm{Card}(\boldsymbol{s}) = 2$.

By computing the fixpoint of $\textit{\textbf{short}}\,(\cdot)$ we have first a FPTIME procedure computing the saturation.

Corollary 1 (computing the saturation). *Given any integer h, there is procedure* $\mathrm{SATUR}_h(\cdot) \in$ FPTIME *that, given an element $F \in \mathcal{S}tack$ such that $\textit{\textbf{h}}(F) \leq h$ as an input, outputs $\textit{\textbf{satur}}(F)$.*

Moreover, we can obtain a further reduction of the nilpotency problem in $\mathcal{S}tack$ related to saturation.

Lemma 5 (rotation). *Let f and g be unary flows. Then fg is a cycle iff gf is a cycle.*

Proof. One should first remark that if a unary flows f is a cycle, then $f^n \neq 0$ for all n. As a particular case, if fg is a cycle, then $(fg)^3 \neq 0$ and as we have $(fg)^3 = f(gf)(gf)g$ we get $(gf)^2 \neq 0$, i.e. gf is a cycle. □

Theorem 8 (cyclicity and saturation). *An element F of $\mathcal{S}tack$ is cyclic (Definition 16) iff either $\textit{\textbf{satur}}(F)^{\uparrow}$ or $\textit{\textbf{satur}}(F)^{\downarrow}$ is.*

Proof. The cyclicity of $\textit{\textbf{satur}}\,(F)^{\uparrow}$ or $\textit{\textbf{satur}}\,(F)^{\downarrow}$ obviously implies that of F because $\textit{\textbf{short}}\,(F) \subseteq F + F^2$, hence $\textit{\textbf{satur}}\,(F) \subseteq \sum_{n\in\mathbb{N}} F^n$.

Conversely, suppose F is cyclic and let $\textit{\textbf{s}} = f_1, \ldots, f_n \in F$ be such that the product $\textit{\textbf{p}}\,(\textit{\textbf{s}}) \in F^n$ is a cycle.

We are going to produce from $\textit{\textbf{s}}$ a sequence of elements of $\textit{\textbf{satur}}\,(F)^{\uparrow}$ or $\textit{\textbf{satur}}\,(F)^{\downarrow}$ whose product is a cycle. For this we apply to the sequence the following rewriting procedure:

1. If there are f_i and f_{i+1} such that f_i is decreasing and f_{i+1} is increasing, then rewrite $\textit{\textbf{s}}$ as $f_1, \ldots, f_i f_{i+1}, \ldots, f_n$.
2. If step 1 does not apply and $\textit{\textbf{s}} = \textit{\textbf{s}}_1\textit{\textbf{s}}_2$ ($\textit{\textbf{s}}_1$ and $\textit{\textbf{s}}_2$ both non-empty) with all elements of $\textit{\textbf{s}}_1$ increasing and all elements of $\textit{\textbf{s}}_2$ decreasing, then rewrite $\textit{\textbf{s}}$ as $\textit{\textbf{s}}_2\textit{\textbf{s}}_1$.

This rewriting procedure preserves the following invariants:

– All elements of the sequence are in $\textit{\textbf{satur}}\,(F)$: step 2 does not affect the elements of the sequence (only their order) and step 1 replaces the flows $f_i \in \textit{\textbf{satur}}\,(F)^{\downarrow}$ and $f_{i+1} \in \textit{\textbf{satur}}\,(F)^{\uparrow}$ by $f_i f_{i+1} \in \textit{\textbf{satur}}\,(F)$.
– The product $\textit{\textbf{p}}\,(\textit{\textbf{s}})$ of the sequence is a cycle: step 1 does not alter $\textit{\textbf{p}}\,(\textit{\textbf{s}})$ and step 2 does not alter the fact that $\textit{\textbf{p}}\,(\textit{\textbf{s}})$ is a cycle by Lemma 5.

The rewriting terminates as step 1 strictly reduces the length of the sequence and step 2 can never be applied twice in a row (it can be applied only when step 1 is impossible and its application makes step 1 possible). Let $g_1, \ldots, g_n$ be the resulting sequence, as it cannot be reduced, the g_i must be either all increasing or all decreasing.

Therefore, by the invariants above $g_1, \ldots, g_n$ is either a sequence of elements of $\textit{\textbf{satur}}\,(F)^{\downarrow}$ or $\textit{\textbf{satur}}\,(F)^{\uparrow}$ such that the product $g_1 \cdots g_n$ is a cycle. □

Finally, we need a way to decide cyclicity of elements of $\mathcal{S}tack$ that are either increasing or decreasing.

Lemma 6. *Given any integer h, there is a procedure* $\textsc{Incr}_h(\cdot) \in \textsc{Ptime}$ *that, given an element $F \in \mathcal{S}tack$ which is either increasing or decreasing and satisfying $\hbar(F) \leq h$ as an input, accepts iff F is nilpotent.*

Proof. Let S be a set of function symbols and h an integer. We define the *truncation wiring* associated to S and h

$$T_{h,S} = \sum_{\tau = \mathrm{f}_1, \dots, \mathrm{f}_h \in S} \tau(\star) \leftharpoondown \tau(x)$$

and set for the rest of the proof $T = T_{\hbar(F),E}$ where E is the set of function symbols occurring in F.

As it contains only flows of the form $\tau(\star) \leftharpoondown \sigma(x)$, i.e. with only one variable, TF is balanced and can be computed in polynomial time since T is of polynomial size in $|F|$.

If F is increasing, an easy computation shows that we have $(TF)^n = TF^n$. From this, we deduce that F is nilpotent iff TF is. If $F = \sum_i \sigma_i(x) \leftharpoondown \tau_i(x)$ is decreasing, we can consider $F^\dagger = \sum_i \tau_i(x) \leftharpoondown \sigma_i(x)$ which is increasing and nilpotent iff F is.

Then, as we know [3, p. 54], [7, Theorem IV.12] the nilpotency problem for balanced wirings to be in co-NLogspace $\subseteq$Ptime, we are done. ☐

Proof of Theorem 5. Simply take $\textsc{Nilp}_h(\cdot) = \textsc{Incr}_h(\textsc{Satur}_h(\cdot))$. By compositionality of Ptime and FPtime algorithms, this procedure is in Ptime. ☐

Proof of Proposition 1. We only provide a sketch of the proof, a complete version is available in Bagnol's PhD thesis [7, proposition IV.21].

The idea is that the product $O\bar{W}_p$ can be seen as an element of $\mathcal{R}_{\mathbf{b}} \bullet \mathcal{S}tack$. Then, its balanced part can be replaced in polynomial time by closed terms without altering the nilpotency in a way similar to what is done to treat the nilpotency of elements of $\mathcal{R}_{\mathbf{b}}$ [3].

We are left with a flow $\sum_i \mathrm{t}_i \bullet \sigma_i(x) \leftharpoondown \mathrm{u}_i \bullet \tau_i(x)$ such that $\mathrm{t}_i \leftharpoondown \mathrm{u}_i$ is balanced and $\sigma_i(x) \leftharpoondown \tau_i(x)$ is a unary flow, and we can associate to each closed $\mathrm{t}_i, \mathrm{u}_i$, unary function symbols $\underline{\mathtt{t}}_i(\cdot)$, $\underline{\mathtt{u}}_i(\cdot)$, and rewrite our flow as $\sum_i \underline{\mathtt{t}}_i(\sigma_i(x)) \leftharpoondown \underline{\mathtt{u}}_i(\tau_i(x)) \in \mathcal{S}tack$. ☐

Proof of Theorem 6. We have, using the compositionality of Ptime and FPtime again, that $\textsc{Nilp}_{\hbar(O)}(\textsc{Red}_O(\cdot))$ is a decision procedure in Ptime for $\mathcal{L}(O)$. ☐