HAL open science

Memoization for Unary Logic Programming: Characterizing PTIME

Clément Aubert, Marc Bagnol, Thomas Seiller

To cite this version:

Clément Aubert, Marc Bagnol, Thomas Seiller. Memoization for Unary Logic Programming: Characterizing PTIME. 2015. hal-01107377v1

HAL Id: hal-01107377

https://hal.science/hal-01107377v1

Preprint submitted on 20 Jan 2015 (v1), last revised 16 Oct 2015 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - ShareAlike 4.0 International License

Memoization for Unary Logic Programming: Characterizing PTIME

Clément Aubert
Inria
Université Paris-Est, LACL (EA 4219), UPEC,
F-94010 Créteil, France

Marc Bagnol
Aix Marseille Université,
CNRS, Centrale Marseille, I2M UMR 7373,
13453, Marseille, France

Thomas Seiller
IHÉS

***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.**

More precisely, we study the restriction of this framework to terms (and logic programs, rewriting rules) using only unary symbols. We prove it is complete for polynomial time computation, using an encoding of pushdown automata. We then introduce an algebraic counterpart of the memoization technique in order to show its PTIME soundness.

We finally relate our approach and complexity results to complexity of logic programming. As an application of our techniques, we show a PTIME-completeness result for a class of logic programming queries which use only unary function symbols.

***Index Terms*—Implicit Complexity, Resolution, Logic Programming, Polynomial Time, Proof Theory, Pushdown Automata, Geometry of Interaction.**

I. Introduction

Complexity theory focuses on questions related to 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 implementing an algorithm whose execution time is bounded by a polynomial in the size of its input.

However, these definitions depend on the notion of machine and cost-model considered, for the efficiency of an algorithm is sensible to these. The "invariance thesis" [1] is a way to bypass this limitation by defining what "a reasonable model" is: all the "reasonable" models (endowed with cost models) can simulate each other with a "reasonable" overhead. The bootstrap for this notion to apply largely was to remark that polynomial bounds on execution time are robust, as the class of problems captured by different models where this bound coincide. The definition is still machine-dependent, *but not dependent of a particular model of computation*.

One of the main motivations for an implicit computational complexity (ICC) theory is to find completely 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.*" [2, p. 90] This has been already achieved via different approaches, one of which is based on considering restricted programming languages or computational principles [3], [4], [5].

A number of results also arose from proof theory through the study of subsystems of linear logic [6]. More precisely, the Curry-Howard — or *proofs as programs* — correspondence expresses a deep relation between formal proofs and typed programs. For instance, one can define a formula `Nat` which corresponds to the type of binary integers, in the sense that a given (cut-free, i.e. normal, already evaluated) proof of this type represents a given natural number. A proof of the formula `Nat` $\Rightarrow$ `Nat` then corresponds to an algorithm computing a function from integers to integers, where the computation itself amounts to a rewriting on proofs: the cut-elimination procedure.

By restricting the rules of the logical system, one obtains a subsystem where *less* proofs of type `Nat` $\Rightarrow$ `Nat` can be written, hence *less* algorithms can be represented. In a number of such restricted systems the class of accepted proofs, i.e. of programs, corresponds[1] to some complexity class: elementary complexity [7], [8], polynomial time [9], [10], logarithmic [11] and polynomial [12] space.

More recently, new methods for obtaining implicit characterizations of complexity classes based on the *geometry of interaction* (GOI) research program [13] 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 [14], [15] or generalizations of graphs [16], bounded linear maps between Hilbert spaces (operators) [17], [18], [19], clauses (or "flows") [20], [21]. This set of objects is then considered together with an abstract notion of execution, seen as an interactive process: a function does not process a static input, but rather communicate with it, asking for values, reading its answers, asking for another

This work was partly supported by the ANR-10-BLAN-0213 Logoi, the ANR-11- BS02-0010 Récré and the ANR-11-INSE-0007 REVER.

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

value, etc.

Types can then be defined as sets of program representations sharing comparable behaviors. For instance the type $\mathtt{Nat} \Rightarrow \mathtt{Nat}$ is the set of untyped programs which, given an integer as input, produce an integer as output.

This approach based on the GoI differs from previous ICC works using linear logic in that they do not rely on a restriction of some type system, but rather on a restriction on the set of program representations considered. Still, they benefit from previous works in type theory: for instance the representation of integers used here comes from their representation in linear logic, translated in the GoI setting, whose interactive point of view on computation has proven crucial in characterizing logarithmic space computation [11].

The first results that used those innovative considerations were based on operator algebras [22], [23], [24]. Here we consider a more syntactic flavor of the GoI where untyped programs are represented in the so-called *resolution semiring* [21], a semiring based on the resolution rule [25] and a specific class of logic programs. This setting presents some advantages: it avoids the involvement of operator algebras theory, it eases the discussions in terms of complexity (we manipulate first-order terms, which have natural notions of size, height, etc.) and it offers a straightforward connection with complexity of logic programming [26].

Previous works in this direction led to characterizations of logarithmic space predicates LOGSPACE and CO-NLOGSPACE [27], [28], by considering for instance 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[2] are easily related to this simple restriction, for they can be represented as logical programs satisfying this "unarity" restriction. This will imply the completeness of the model under consideration for polynomial time predicates.

We then complete the characterization by showing that any such unary logic program can be decided in polynomial time. This part of the proof consists in an adaptation of S. Cook's memoization technique [29] to the context of logic programs.

The last part of the paper presents consequences of these results in terms of complexity of logic programming, namely that the corresponding class of queries are PTIME-complete, when considering combined complexity [26, p. 380].

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 complexity classes.

A byproduct of this work is to provide a method to test membership in PTIME: if one can rephrase a problem with clauses $H \dashv B$ using only unary function symbols, then our result ensures that the problem lies in PTIME. Conversely if a problem cannot be rephrased that way, it lies outside of PTIME.

[2]More precisely, 2-way k-head non-deterministic finite automata with pushdown stack. See Sect. III-A1.

A. Outline of the paper

We begin by giving in Sect. II-A the formal definition of the resolution semiring; then briefly explain how words can be represented in this structure (Sect. II-B) and recall the characterization of logarithmic space obtained in earlier work (Sect. II-C). In Sect. II-D 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 the fact that multi-head finite automata with pushdown stack characterize PTIME and review the memoization technique in this case (Sect. III-A), and then show how to represent them as elements built from the $\mathcal{S}tack$ semiring (Sect. III-B). The soundness result is then obtained by adapting memoization to the $\mathcal{S}tack$ semiring. This adaptation, which we call the *saturation* technique, is introduced in Sect. IV-A.

In the last section, we formulate our results in terms of complexity of logic programming. In particular, we explain how elements of the $\mathcal{S}tack$ semiring can be seen as a particular kind of unary logic programs to which the saturation technique can be applied. This allows us to show that the combined complexity problem for unary logic program is PTIME-complete.

As an illustration, we show in Sect. V-B that the circuit value problem can be solved with this method.

II. THE RESOLUTION SEMIRING

A. Flows and Wirings

Let us begin with some reminders and notations on first-order terms and unification theory.

Notation II.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 $h(\mathrm{t})$ *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 will be concerned with formal solving of equations of the form $\mathrm{t} = \mathrm{u}$ where t and u are terms. Let us introduce a precise formulation of this problem and some associated vocabulary.

Definition II.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* $\mathtt{var}(\mathrm{t}') \cap \mathtt{var}(\mathrm{u}') = \varnothing$*;*
- 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 [30, Theorem 1].

The notion of MGU allows to formulate the *resolution rule*, a key concept of logic programming that defines 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 on 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 II.3 (flow). *A* flow *is an ordered pair* f *of terms* $f := \mathrm{t} \leftharpoonup \mathrm{u}$*, with* $\mathtt{var}(\mathrm{t}) \subseteq \mathtt{var}(\mathrm{u})$*. Flows are considered up to renaming: for any renaming* α*,* $\mathrm{t} \leftharpoonup \mathrm{u} = \alpha\mathrm{t} \leftharpoonup \alpha\mathrm{u}$.

A flow $\mathrm{t} \leftharpoonup \mathrm{u}$ can also be understood as a rewriting rule over the set of first-order terms. For instance, the flow $\mathtt{g}(x) \leftharpoonup \mathtt{f}(x)$ corresponds to the following rewriting rule: terms of the form $\mathtt{f}(\mathrm{v})$ where v is a term are rewritten as $\mathtt{g}(\mathrm{v})$ and all other terms are left unchanged.

We will soon define the *product* of flows which provides a way of composing them; 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 $f_2 := \mathtt{g}(x) \leftharpoonup \mathtt{f}(x)$ to produce the flow $f_1 f_2 = \mathtt{h}(x) \leftharpoonup \mathtt{f}(x)$. 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 II.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* $(\mathrm{t} \leftharpoonup \mathrm{u})(\mathrm{v} \leftharpoonup \mathrm{w}) := \theta\mathrm{t} \leftharpoonup \theta\mathrm{w}$.

We now define wirings, which are simply 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 is naturally lifted to wirings.

Definition II.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.[3] 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: $\{ f_1, \ldots, f_n \} := f_1 + \cdots + f_n$.
- We denote by 0 the empty set, i.e. the unit of $+$.
- We have a unit for the product, the wiring $I := x \leftharpoonup x$.

As we will always be working within $\mathcal{R}$, the term "semiring" will be used instead of "subsemiring of $\mathcal{R}$".

Finally, let us recall the notion of nilpotency in a semiring and extend the notion of height (of terms) to flows and wirings.

Definition II.6 (height). *The* height $h(f)$ *of a flow* $f = \mathrm{t} \leftharpoonup \mathrm{u}$ *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 II.7 (nilpotency). *A wiring* F *is* nilpotent *— written* $\mathbf{Nil}(F)$ *— if and only if* $F^n = 0$ *for some* n.

The above 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 — note however that we consider rewriting that occur only at the root of terms, while the usual notion from term rewriting systems [31] allows in-context rewriting.

B. Representation of Words and Programs

This section explains and motivates the representation of words as flows. By studying their interactions with wirings from a specific semiring, notions of program and language are defined.

First, let us see how the binary function symbol $\bullet$ used to construct terms can be extended to build flows and then semirings.

[3] 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; the element 0 is absorbent: $0r = r0 = 0$ for all $r \in R$.

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

We define $(\mathtt{u} \leftharpoonup \mathtt{v}) \bullet (\mathtt{t} \leftharpoonup \mathtt{w}) := \mathtt{u} \bullet \mathtt{t} \leftharpoonup \mathtt{v} \bullet \mathtt{w}$. *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 II.9. *We write* $\mathtt{t} \leftrightharpoons \mathtt{u}$ *the sum* $\mathtt{t} \leftharpoonup \mathtt{u} + \mathtt{u} \leftharpoonup \mathtt{t}$.

Definition II.10 (word representations). *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:*

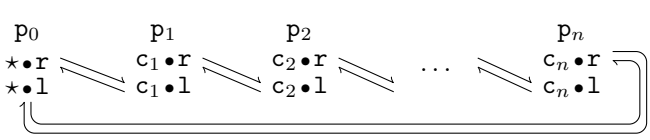
$$\bar{W}_p = \sum_{i=0}^{n} \mathtt{c}_i \bullet \mathtt{r} \bullet x \bullet y \bullet \textsc{head}(\mathtt{p}_i) \leftrightharpoons \mathtt{c}_{i+1} \bullet \mathtt{l} \bullet x \bullet y \bullet \textsc{head}(\mathtt{p}_{i+1})$$

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 will be given in the proof of Theorem III.3.

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:



On the other hand, 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 classes of observations that characterize specific complexity classes. The important point here is that we forbid an observation to use any position constant, in order to have it interact the same way with all the representations $\bar{W}_p$ of a word W.

Definition II.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 theorem is a consequence [21, Theorem IV.5] of the fact that observations cannot use position constants.

Theorem II.12 (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* $\mathbf{Nil}(O\bar{W}_q)$.

With this theorem, we can safely define how a word can be accepted by an observation: the notion is independent of the specific choice of a representation of position constants.

Definition II.13 (accepted language). *Let* O *be an observation. We define the* language accepted by O *as*

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

C. Balanced Flows and Logarithmic Space

In previous work [28], we investigated the semiring of *balanced wirings*, that are defined as sets of balanced — or "height-preserving"— flows.

Definition II.14 (balance). *A flow* $f = \mathtt{t} \leftharpoonup \mathtt{u}$ *is* balanced *if for any variable* $x \in \mathtt{var}(\mathtt{t}) \cup \mathtt{var}(\mathtt{u})$*, all occurrences of* x *in both* $\mathtt{t}$ *and* $\mathtt{u}$ *have the same height (recall notations p. 2). A* balanced *wiring* F *is a sum of balanced flows.*

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

Definition II.15 (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 logarithmic space computation [28, Theorems 34-35].

Theorem II.16 (balance and logarithmic space). *If* O *is a balanced observation, then* $\mathcal{L}(O) \in$ CO-NLOGSPACE. *If* $L \in$ CO-NLOGSPACE *then there exists a balanced observation such that* $\mathcal{L}(O) = L$.

It also appears that a natural subclass of balanced wirings characterizes DLOGSPACE, the class of *deterministic* logarithmic space computable predicates.

D. The $\mathcal{S}tack$ Semiring

This paper deals with another restriction on flows, namely the restriction to *unary flows*, i.e. flows defined from 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 characterize 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 [21].

Definition II.17 (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 $\sum_i \mathtt{t}_i \leftharpoonup \mathtt{u}_i$ where the $\mathtt{t}_i \leftharpoonup \mathtt{u}_i$ are unary flows.

Example II.18. The flows $\mathtt{f}(\mathtt{f}(x)) \leftharpoonup \mathtt{g}(x)$ and $x \leftharpoonup \mathtt{g}(x)$ are unary, while $x \bullet \mathtt{f}(x) \leftharpoonup \mathtt{g}(x)$ and $\mathtt{f}(\mathtt{c}) \leftharpoonup x$ are not.

Notation II.19 (stack operations). *If $\tau = \mathtt{g}_1, \ldots, \mathtt{g}_n$ is a finite sequence of unary function symbols and $\mathtt{t}$ is a term, we write $\tau(\mathtt{t}) := \mathtt{g}_1\big(\mathtt{g}_2(\cdots \mathtt{g}_n(\mathtt{t})\cdots\big)$. We will write $\tau\sigma$ the concatenation of the sequences τ and σ. Given two sequences τ and σ we define the flow $\mathtt{OP}_{\tau,\sigma} := \tau(x) \leftharpoonup \sigma(x)$ which we call a* stack operation.

Note that, by definition, an element of the $\mathcal{S}tack$ semiring must be a sum of stack operations.

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 $\mathcal{S}tack$, i.e. a wiring in $\mathcal{S}tack$ will be shown to be either cyclic or nilpotent.

Definition II.20 (cyclicity). *A flow $\mathtt{t} \leftharpoonup \mathtt{u}$ is a* cycle *if $\mathtt{t}$ and $\mathtt{u}$ are matchable (Definition II.2). A wiring F is* cyclic *if there is a k such that F^k contains a cycle.*

For $\vec{s} = f_1, \ldots, f_n$ a sequence of stack operations, define:

- *its* height *as $h(\vec{s}) := \max_i \{h(f_i)\}$*
- *its* cardinality[4] $\mathrm{Card}(\vec{s}) := \mathrm{Card}\{\, f_i \mid 1 \leq i \leq n \,\}$.
- *its product $p(\vec{s})$ as $f_1 \cdots f_n$.*

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

Remark II.21. A flow f is a cycle iff $f^2 \neq 0$.

To carry on the proof evoked above that cyclicity and nilpotency are complementary notions in $\mathcal{S}tack$, we borrow a result from an earlier work about GoI and complexity in the context of an algebra of Horn clauses.

Lemma II.22 (acyclic sequence [32, lemma 5.3]). *If $\vec{s}$ is an acyclic sequence of stack operations, then we have*

$$h\big(p(\vec{s})\big) \leq h(\vec{s})(\mathrm{Card}(\vec{s}) + 1)$$

The following property says that cycles in $\mathcal{S}tack$ can be iterated indefinitely, i.e. a stack operation $\mathtt{OP}_{\tau,\sigma}$ such that $(\mathtt{OP}_{\tau,\sigma})^2 \neq 0$ is never nilpotent.

Proposition II.23. *If a stack operation f is a cycle, then $f^n \neq 0$ for all n.*

Remark II.24. This does not hold for general flows. For instance, $f = x \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet x$ is a cycle because $f^2 = \mathtt{c} \bullet \mathtt{c} \leftharpoonup \mathtt{d} \bullet \mathtt{d} \neq 0$ (by Remark II.21), 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 II.25 (nilpotency in $\mathcal{S}tack$). *A wiring $F \in \mathcal{S}tack$ is nilpotent iff it is acyclic.*

Proof ▸ Suppose F is not nilpotent, so that there is at least one stack operation in F^n for any n, and let S be the number of different function symbols appearing in F. Set $k := (S^{h(F)(\mathrm{Card}(F)+1)} + S^{h(F)(\mathrm{Card}(F)+1)-1} + \cdots + 1)^2$, i.e. the total number of different flows of height at most $h(F)(\mathrm{Card}(F)+1)$ using the symbols appearing in F.

Let $f \neq 0$ be an element of F^{k+1}. It is the product $p(\vec{s})$ of a sequence $\vec{s} = f_1, \ldots, f_{k+1}$ of stack operations that belong to F. We show by contradiction that this sequence must be cyclic, so let us suppose it is not. By Lemma II.22, we know that for any $i > 0$, setting $\vec{s}_i := f_1, \ldots, f_i$ we have

$$h\big(p(\vec{s}_i)\big) \leq h(\vec{s}_i)(\mathrm{Card}(\vec{s}_i) + 1) \leq h(F)(\mathrm{Card}(F) + 1)$$

Therefore, for any $i > 0$ the flow $p(\vec{s}_i)$ is of height at most $h(F)(\mathrm{Card}(F)+1)$ and uses only symbols appearing in F, i.e. it wanders in a set of cardinal k, so there must be $1 \leq i < j \leq k+1$ such that $p(\vec{s}_i) = p(\vec{s}_j)$.

Now, setting $\vec{s}_{i+1,j} := f_{i+1}, \ldots, f_j$, we have that $p(\vec{s}_i)p(\vec{s}_{i+1,j}) = p(\vec{s}_j) = p(\vec{s}_i)$ hence $p(\vec{s}_i)p(\vec{s}_{i+1,j})^2 = p(\vec{s}_i) \neq 0$ and thus $p(\vec{s}_{i+1,j})^2 \neq 0$ i.e. $p(\vec{s}_{i+1,j})$ is a cycle. As $p(\vec{s}_{i+1,j}) \in F^{j-i}$ we can conclude that F is cyclic.

The converse is an immediate consequence of Proposition II.23. ◂

Example II.26. Consider the wiring

$$\begin{array}{rl} F := & \mathtt{f}_1(x) \leftharpoonup \mathtt{f}_0(x) \\ + & \mathtt{f}_0(\mathtt{f}_1(x)) \leftharpoonup \mathtt{f}_1(\mathtt{f}_0(x)) \\ + & \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_1(x))) \leftharpoonup \mathtt{f}_1(\mathtt{f}_1(\mathtt{f}_0(x))) \\ + & \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \leftharpoonup \mathtt{f}_1(\mathtt{f}_1(\mathtt{f}_1(x))) \end{array}$$

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 $\mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \leftharpoonup \mathtt{f}_0(\mathtt{f}_0(\mathtt{f}_0(x))) \in F^8$.

If we lift this example to the case of a counter from 0 to $2^n - 1$ that resets to 0 when it reaches 2^n, we obtain an example of a wiring F of cardinal n and height $n-1$ such that F^{2^n} contains a cycle, but F^{2^n-1} does not. This shows that the number of iterations needed to find a cycle may be exponential in the height and the cardinal 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 II.27. *A* balanced observation with stack *is an element of $\mathcal{O}^{\mathbf{b+s}} := \mathcal{O}[\,\mathcal{S}tack \bullet \mathcal{R}_{\mathbf{b}}\,]$.*

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

III. Pushdown Automata and Ptime Completeness

A. Characterization of Ptime by Pushdown Automata

The class of deterministic polynomial time computable predicates Ptime is the most studied complexity class, mainly because it supposedly contains all "tractable" problems.

Extending our approach to this class was a long-standing goal, whose completeness part is attained thanks to the connection with pushdown automata. In this subsection, we recall their definition, the Ptime characterization theorem we will rely on and the memoization technique.

1) Definition and classical results: Automata form a very basic 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.

Among possible extensions, our interest will focus on the addition of a "pushdown stack" (together with multiple heads), which we referred to as "pushdown automata" until now. We will see that this leads to a characterization of Ptime.

Let us give below the most general definition, for the non-deterministic case.

Definition III.1 (2MFA+S). *For $k \geqslant 1$, a 2-way k-head finite automaton with pushdown stack (2MFA+S(k)) is a tuple $M = \{\boldsymbol{S}, \boldsymbol{i}, A, B, \rhd, \lhd, \boxdot, \sigma\}$ 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*;*
- *$\rhd$ and $\lhd$ are the* left *and* right endmarkers*, $\rhd, \lhd \notin A$;*
- *$\boxdot$ is the* bottom symbol of the stack*, $\boxdot \notin B$;*
- *σ is the* transition relation*, i.e. a subset of the product $(\boldsymbol{S} \times (A_{\bowtie}\})^k \times B_{\boxdot}) \times (\boldsymbol{S} \times \{-1, 0, +1\}^k \times \{\mathtt{pop}, \mathtt{push}(b)\})$ where $A_{\bowtie}$ (resp. $B_{\boxdot}$) denotes $A \cup \{\rhd, \lhd\}$ (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 $\mathtt{pop}$ means "erase the top symbol", while, for all $b \in B$, $\mathtt{push}(b)$ 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 special state" [33, Theorem 2, p. 125]. Modulo another standard transformation, we restrict the transition relation so that at most one head moves at each transition.

Without pushdown stacks, 2-way k-head finite automata characterize Logspace and NLogspace, depending on the automata being deterministic or not.

This result, used in our previous work [28], [24], was first stated informally by Juris Hartmanis [34, pp. 338–339] and is often [35, p. 13], [34, pp. 338–339], attributed to Alan Cobham. However, a detailed proof can be found in a classical handbook [36, pp. 223–225]. The addition of a pushdown stack improves the expressivity of the machine model, as stated in the following theorem.

Theorem III.2. 2MFA+S *characterize* Ptime.

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

Simulating a polynomial-time Turing machine with a 2MFA+S 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 [35, pp. 9–11] as well as a precise algorithm [36, pp. 238–240] can be found in the literature.

Simulating a 2MFA+S 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 automaton, one can design an automaton that recognizes the same language but runs exponentially slower [37, p. 197]. That the automaton can accept its input after an exponential computation time is similar with the situation of the counter in Example II.26.

The technique invented by Alfred V. Aho et al. [37] and made popular by Stephen A. Cook consists in building a "memoization table" that allows the Turing machine to create shortcuts in the simulation of the automaton, decreasing drastically its computation time. In some cases, an automaton with an exponentially long run can even be simulated in linear time [29].

We give more details on this technique in the next subsection, as its adaptation to our context will be a key ingredient in the soundness proof in Sect. IV.

2) The memoization technique: Although the name comes from machine-learning [38], this technique is usually attributed to S. A. Cook and has provided fundamental as well as practical results. In the specific case of automata with stack, it can be condensed in the following remark: if at a given time you are in state $\mathbf{q}$ with b on top of a stack of height $h \geqslant 1$, and if you end up later on in the state $\mathbf{q}'$ with some symbol b' on top of a stack of height h, *without having popped a symbol at height inferior to h*, and if you are about to pop this symbol, then you can save this progression $(\mathbf{q}, b) \to (\mathbf{q}', b')$. If later on you find yourself in the same state $\mathbf{q}$, with b on top of your stack and with the heads in the same positions, you can directly skip to the saved progression, as there is no need to perform this part of the computation again. This "partial information", the description of your automaton without the contents of the stack, apart from its top symbol, is sometimes called "surface configuration" or "partial identifier".

The memoization technique consists in building and using the transitive closure of the relation between surface configuration. Differently expressed, memoization is a "*clever evaluation strategy*, applicable whenever the results of certain computations are needed more than once" [39, p. 348]. One looking for subtle refinements could look for a technique of memoization computed independently from the input, allowing to "compile" a stack program into equivalent online mem-

oizing program [40] that runs exponentially faster. A nice explanation in the case of single head automata can be found in a recent and short article by R. Glück [41].

We will be adapting this idea to our context in Sect. IV-A, which will amount to a form of *exponentiation by squaring*.

B. Encoding 2MFA+S *as Observations:* PTIME *Completeness*

The encoding proposed below is similar to the previously developed [28, 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 II.13). 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.

We turn now to the proof of PTIME-*completeness* for the set of balanced observations with stacks.

Theorem III.3. *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 ▸ The proof relies on encoding a 2MFA+S(k) M that recognizes $\mathcal{L}$ — whose existence is ensured by Theorem III.2 — as an observation of $\mathcal{O}^{\mathbf{b+s}}$. Taking $A = \Sigma$ the input alphabet, $k+1$ the number of heads of the automaton, we will encode the transition relation of M as a balanced observation with stack. 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

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

with

- $\mathtt{c}, \mathtt{c}' \in \Sigma \cup \{\star\}$,
- $\mathtt{d}, \mathtt{d}' \in \{\mathtt{l}, \mathtt{r}\}$,
- σ a finite sequence of unary function symbols,
- $\mathtt{s}$ a unary function symbol,
- $\mathtt{q}, \mathtt{q}'$ two constant symbols,
- $\textsc{aux}_k$, $\textsc{head}$ 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

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

where the $\mathtt{p}_i$ are position constants representing the positions of the *main pointer* ($\textsc{head}(\mathtt{p}_j)$) and of the *auxiliary pointers* ($\textsc{aux}_k(\mathtt{p}_{i_1}, \dots, \mathtt{p}_{i_k})$); 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 be encoding M as an observation O_M, and observe the iterations of $O_M \bar{W}_p$, its product with a word representation. Let us explain how the basic operations of M are encoded:

Moving the pointers. Looking back at the definition of the encoding of words (Definition II.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 II.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

$$\cdots \bullet \textsc{aux}_k(x, \dots, y_k) \bullet \textsc{head}(y_1) \leftharpoonup \\ \cdots \bullet \textsc{aux}_k(y_1, \dots, y_k) \bullet \textsc{head}(x)$$

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 $\textsc{aux}_k(\cdots)$ —, but this can be implemented easily using additional states.

One can see that it is the interaction between the observation O_M and the word representation $\bar{W}_p$ that simulates the behavior of the automaton, and not the observation on its own manipulating some passive data.

Handling the stack. Suppose we have a unary function symbol $\underline{\mathtt{b}}(\cdot)$ for each symbol b of the stack alphabet $B_{\boxdot}$.

A transition that reads b and pops it is simply written as

$$\cdots \bullet x \bullet \cdots \leftharpoonup \cdots \bullet \underline{\mathtt{b}}(x) \bullet \cdots$$

A transition that reads b and pushes a symbol c is written

$$\cdots \bullet \underline{\mathtt{c}}(\underline{\mathtt{b}}(x)) \bullet \cdots \leftharpoonup \cdots \bullet \underline{\mathtt{b}}(x) \bullet \cdots$$

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 [23, 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, as discussed in details in earlier work [23]. The reason for this is that another encoding may interfere with the representation of acceptation as termination: the existence of a loop in the observation O_M representing the automaton M, *even one that is not used in the computation with the input* W, prevents the wiring $O_M \bar{W}_p$ from being nilpotent.

Indeed, the "loop" in Definition III.1 of 2MFA+S is to be read as "perform forever the same computation". ◂

Notice that the encoding of pushdown automata as observations with stacks produces only specific observations, namely those that are sums of flows of a particular form (shown at the beginning of the preceding proof). This is due to the fact that one encodes the transitions directly, so that each flow corresponds to a transition step.

In particular, as the transition relation of automata depends only on the top of the stack, the body (i.e. the right-hand part) of the flows must be of the form $\cdots \bullet \underline{\mathtt{b}}(x) \bullet \cdots$. However, a general observation with stack is not constrained in this way, and allows a more compact representation of programs where one can read, pop and push several symbols of the stack simultaneously.

Nevertheless, this 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.

IV. NILPOTENCY IN $\mathcal{S}tack$ AND PTIME SOUNDNESS

A. The Saturation Technique

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 under certain conditions, the height of flows does not grow when computing their product. It adapts memoization to our setting: we repeatedly extend the wiring by adding pairwise products of flows, allowing for more and more "transitions" at each step.

Notation IV.1. *Let τ and σ be sequences of unary function symbols.*

If $h(\tau(x)) \geq h(\sigma(x))$ we say that $\mathtt{OP}_{\tau,\sigma}$ is increasing.

If $h(\tau(x)) \leq h(\sigma(x))$ we say that $\mathtt{OP}_{\tau,\sigma}$ is decreasing.

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

Lemma IV.2 (stability of height). *Let $f = \mathtt{OP}_{\tau,\sigma}$ and $g = \mathtt{OP}_{\rho,\chi}$ be stack operations. If f is decreasing and g is increasing, we have $h(fg) \leq \max\{h(f), h(g)\}$.*

Proof ▸ If $fg = 0$, the property holds because $h(0) = 0$. 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 $fg = \mathtt{OP}_{\tau,\chi\mu}$ and $h(\sigma) = h(\rho\mu)$.

As g is increasing, $h(\chi) \leq h(\rho)$ and therefore we have $h(\chi\mu) \leq h(\rho\mu) = h(\sigma) \leq h(f) \leq \max\{h(f), h(g)\}$. ◂

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.

We are designing in fact an *exponentiation by squaring* procedure for elements of $\mathcal{S}tack$, the algebraic reading of memoization.

Definition IV.3 (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 $\mathtt{short}$.

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 IV.4 (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 $h\big(\mathtt{short}^n(F)\big) = h(F)$.

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

Proof ▸ By Lemma IV.2 we have

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

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

For any n, the elements of $\mathtt{short}^n(F)$ are stack operations of height at most $h(F)$ built from the function symbols appearing in F, therefore $\mathtt{short}^n(F)$ is a subset of a set of cardinality $k := (S^{h(F)} + S^{h(F)-1} + \cdots + 1)^2$. As $G \subseteq \mathtt{short}(G)$ for all G, the iteration of $\mathtt{short}(\cdot)$ 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 IV.5 (size). *The* size *$|F|$ of a wiring F is defined as the total number of function symbol occurrences in it.*

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

Corollary IV.6 (computing the saturation). *Given any integer h, there is procedure $\textsc{Satur}_h(\cdot) \in$ FPTIME that, given*

an element $F \in \mathcal{S}tack$ such that $h(F) \leq h$ as an input, outputs $\mathtt{satur}(F)$.

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

Lemma IV.7 (rotation). *Let f and g be stack operations. Then fg is a cycle iff gf is a cycle.*

Proof ▸ If fg is a cycle, then $(fg)^n \neq 0$ for any n by Proposition II.23. In particular $(fg)^3 \neq 0$ and as we have $(fg)^3 = f(gf)(gf)g$ we get $(gf)^2 \neq 0$, i.e. gf is a cycle. ◂

Theorem IV.8 (cyclicity and saturation). *An element F of $\mathcal{S}tack$ is cyclic (Definition II.20) iff either $\mathtt{satur}(F)^{\uparrow}$ or $\mathtt{satur}(F)^{\downarrow}$ is.*

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

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

We are going to produce from $\vec{s}$ a sequence of elements of $\mathtt{satur}(F)^{\uparrow}$ or $\mathtt{satur}(F)^{\downarrow}$ 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 $\vec{s}$ as $f_1, \ldots, f_i f_{i+1}, \ldots, f_n$.
2) If step 1 does not apply and $\vec{s} = \vec{s}_1 \vec{s}_2$ ($\vec{s}_1$ and $\vec{s}_2$ both non-empty) with all elements of $\vec{s}_1$ increasing and all elements of $\vec{s}_2$ decreasing, then rewrite $\vec{s}$ as $\vec{s}_2 \vec{s}_1$.

This rewriting procedure preserves the following invariants:

- All elements of the sequence are in $\mathtt{satur}(F)$: step 2 does not affect the elements of the sequence (only their order) and step 1 replaces the flows $f_i \in \mathtt{satur}(F)^{\downarrow}$ and $f_{i+1} \in \mathtt{satur}(F)^{\uparrow}$ by $f_i f_{i+1} \in \mathtt{satur}(F)$.
- The product $p(\vec{s})$ of the sequence is a cycle: step 1 does not alter $p(\vec{s})$ and step 2 does not alter the fact that $p(\vec{s})$ is a cycle by Lemma IV.7.

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 $\mathtt{satur}(F)^{\downarrow}$ or $\mathtt{satur}(F)^{\uparrow}$ 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 IV.9. *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 $h(F) \leq h$ 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

$$T_{h,S} := \sum_{\tau = \mathtt{f}_1, \ldots, \mathtt{f}_h \in S} \tau(\star) \leftarrow \tau(x)$$

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

As it contains only flows of the form $\tau(\star) \leftarrow \sigma(x)$, 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 $F = \sum_i \sigma_i(x) \leftarrow \tau_i(x)$ is decreasing, we can consider $F^{\dagger} := \sum_i \tau_i(x) \leftarrow \sigma_i(x)$ which is increasing and nilpotent iff F is.

Then, as we know [28, p. 54], [21, Theorem IV.12] the nilpotency problem for balanced wirings to be in CO-NLOGSPACE $\subseteq$ PTIME, we are done. ◂

Theorem IV.10 (nilpotency is in PTIME). *Given any integer h, there is a procedure $\textsc{Nilp}_h(\cdot) \in \textsc{Ptime}$ that, given a $F \in \mathcal{S}tack$ such that $h(F) \leq h$ as an input, accepts iff F is nilpotent.*

Proof ▸ 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. ◂

Remark IV.11. All the results we gave in this section are parametrized by a height limit h, but this is only to ease the presentation. Indeed, it is possible to transform any element of $\mathcal{S}tack$ with an unspecified height into another element of comparable size but of height at most 2, preserving its nilpotency.

More precisely: consider a flow $l = \sigma(x) \leftarrow \tau(x)$, with $\sigma = \mathtt{f}_1, \ldots, \mathtt{f}_m$ and $\tau = \mathtt{g}_1, \ldots, \mathtt{g}_n$. Let us introduce new function symbols $\mathtt{l}_i^{\mathtt{pop}}(\cdot)$ and $\mathtt{l}_j^{\mathtt{push}}(\cdot)$ for $1 \leq i \leq m$ and $1 \leq j \leq n$. We can rewrite l as the sum

$$\mathtt{g}_1(x) \leftarrow \mathtt{l}_1^{\mathtt{push}}(x) + \mathtt{l}_1^{\mathtt{push}}(\mathtt{g}_2(x)) \leftarrow \mathtt{l}_2^{\mathtt{push}}(x) + \cdots$$
$$\cdots + \mathtt{l}_n^{\mathtt{push}}(x) \leftarrow \mathtt{l}_1^{\mathtt{pop}}(x) + \cdots$$
$$\cdots + \mathtt{l}_{m-1}^{\mathtt{pop}}(x) \leftarrow \mathtt{l}_m^{\mathtt{pop}}(\mathtt{f}_{m-1}(x)) + \mathtt{l}_m^{\mathtt{pop}}(x) \leftarrow \mathtt{f}_m(x)$$

with the idea that instead of popping and pushing several symbols at the same time, we do this step by step: we push/pop only one symbol and then leave a marker (either $\mathtt{l}_i^{\mathtt{pop}}(\cdot)$ or $\mathtt{l}_j^{\mathtt{push}}(\cdot)$) for the next operation to be performed. Moreover, we see that this flow of size (Definition IV.5) $|l| = m + n$ is transformed in a wiring containing three flows of size 2 (the central and two extremal ones) and $(m-1) + (n-1)$ flows of size 3; hence the sum has size $3|l|$.

When dealing with a wiring W, we can do the same by considering one family of $\mathtt{l}_i^{\mathtt{pop}}(\cdot)$ and $\mathtt{l}_j^{\mathtt{push}}(\cdot)$ symbols for each flow l of W. It is not hard to see that the resulting wiring W_{flat} has the same behavior as the original one in terms of nilpotency. It is also clear that the height of W_{flat} is indeed 2. Finally, if we started with a wiring W of size N, which is

the sum of the sizes of the flows in it, we get in the end that $|W_{\text{flat}}| = 3|W|$ using the one-flow case above.

This suggests by the way that the bound in Proposition IV.4 is probably too rough, but a way to sharpen it still needs to be found.

B. PTIME Soundness

We will now use the saturation technique to prove that the language recognized by an observation with stack 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 $O\bar{W}_p$. One can then decide the nilpotency of this element thanks to the procedure described in the previous section.

Proposition IV.12. *Let $O \in \mathcal{O}^{\mathbf{b+s}}$ be an observation with stack. There is a procedure $\text{RED}_O(\cdot) \in \text{FPTIME}$ that, given a word W as an input, outputs a wiring $F \in \mathcal{S}tack$ with $h(F) \leq h(O)$ such that F is nilpotent iff $O\bar{W}_p$ is for any choice of $\vec{\mathtt{p}}$.*

Proof (sketch [21, 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 $\mathcal{R}_{\mathbf{b}}$ [28].

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

Theorem IV.13 (soundness). *If $O \in \mathcal{O}^{\mathbf{b+s}}$ is an observation with stack, then $\mathcal{L}(O) \in \text{PTIME}$.*

Proof ▸ We have, using the compositionality of PTIME and FPTIME again, that $\text{NILP}_{h(O)}(\text{RED}_O(\cdot))$ is a decision procedure in PTIME for $\mathcal{L}(O)$. ◂

V. 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.

We cover in here the consequences in terms of logic programming of the results and techniques introduced so far. The basic definitions and a list of previously known results — that highlight the novelty of our result — regarding logic programming can be found in an extensive survey [26].

As an illustration, we show in Sect. V-B how the classical boolean circuit value problem (CVP) [42] can be encoded as a unary logic program, thus providing an alternative proof of its inclusion in PTIME.

A. Unary Queries

Definition V.1 (data, goal, query). *A* unary query *is a triple* $\mathbf{Q} = (D, P, G)$*, 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 $d \dashv$ for $d \in D$ and the elements of P by the resolution rule exposed in Sect. II-A, 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 an encoding.

Definition V.2 (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 $\mathcal{S}tack$:

$$[D] := \{\, \tau(\underline{\mathtt{c}}(x)) \leftharpoonup \mathtt{START}(x) \mid \tau(\mathtt{c}) \in D \,\}$$

and to any unary goal $G = \tau(\mathtt{c})$ we associate

$$\langle G \rangle := \mathtt{ACCEPT}(x) \leftharpoonup \tau(\underline{\mathtt{c}}(x))$$

Remark V.3. 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 V.4 (success). *A unary query $\mathbf{Q} = (D, P, G)$ succeeds if and only if*

$$\mathtt{ACCEPT}(x) \leftharpoonup \mathtt{START}(x) \in \langle G \rangle P^n[D] \quad \textit{for some } n$$

Then, we can show that the saturation technique applies to the problem of deciding whether a unary query accepts. The proof uses the saturation technique (Sect. IV-A) to rewrite a sequence of flows, adding to them "pre-computed" rewriting rules.

Lemma V.5 (saturation of unary queries). *A unary query $\mathbf{Q} = (D, P, G)$ succeeds if and only if*

$$\mathtt{ACCEPT}(x) \leftharpoonup \mathtt{START}(x) \in \mathtt{satur}\big([D] + P + \langle G \rangle\big)$$

Theorem V.6 (PTIME-completeness). *The* UQUERY *problem (given a unary query, is it successful?) is* PTIME*-complete.*

Proof ▸ The lemma above, combined with Corollary IV.6, ensures that the problem lies indeed in the class PTIME, *modulo* the considerations on the height of Remark IV.11.

The hardness part follows from a variation on the encoding presented in Sect. III-B and the reduction derived from Proposition IV.12. ◂

Remark V.7. 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;
- that in $\mathtt{t} \leftharpoonup \mathtt{u}$ the variables of $\mathtt{t}$ does not appear in $\mathtt{u}$;
- constants in the program part of the query.

A harder question is whether everything scales up to logic programs of the form $H \dashv B_1, \ldots, B_n$, with more than one formula on the right of $\dashv$. Indeed we would no longer have obvious notions of *increasing* or *decreasing* (Notation IV.1) clause anymore, and these are crucial to the saturation technique. It is already known [26, pp. 386–387] that in the case of *propositional* (i.e. with no variables) logic programming, allowing more than one B_i makes the combined complexity (see Remark V.8 below) switch from LOGSPACE to PTIME: one can expect by analogy a higher complexity than PTIME in our unary case, but nothing has been proven yet.

Remark V.8. In terms of complexity of logic programs, we are considering the *combined complexity* [26, 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. III-B 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.

B. Circuit Value Problem

To illustrate our point in the introduction about rephrasing a problem with unary symbols to tell whether it lies in PTIME, we present an encoding of the classical PTIME-complete *circuit value problem* (CVP) [26] as a unary query.

An instance of CVP is a boolean circuit composed of $\mathtt{and}$, $\mathtt{or}$, $\mathtt{not}$, $\mathtt{0}$ and $\mathtt{1}$ gates and is *accepted* if the circuit computes the value $\mathtt{1}$ at its output gate.

More formally, we can see an instance of CVP as (G, o) with G an acyclic directed hypergraph[5] with a distinguished output vertex o built with edges among

$$a, b \rhd^{\mathtt{and}} c \qquad a, b \rhd^{\mathtt{or}} c \qquad a \rhd^{\mathtt{not}} b \qquad \rhd^{0} a \qquad \rhd^{1} a$$

such that any vertex is the target of exactly one edge.

First, we associate to each vertex v of the graph a pair $\mathtt{v}(\cdot), \overline{\mathtt{v}}(\cdot)$, of unary function symbols. Then to each edge e we associate a flow $[e]$ as follows:

$$\begin{aligned}
[a, b \rhd^{\mathtt{and}} c] &:= \mathtt{a}(\mathtt{b}(x)) \leftharpoonup \mathtt{c}(x) \\
&\quad + \overline{\mathtt{a}}(x) \leftharpoonup \overline{\mathtt{c}}(x) \ + \ \overline{\mathtt{b}}(x) \leftharpoonup \overline{\mathtt{c}}(x) \\
[a, b \rhd^{\mathtt{or}} c] &:= \mathtt{a}(x) \leftharpoonup \mathtt{c}(x) \ + \ \mathtt{b}(x) \leftharpoonup \mathtt{c}(x) \\
&\quad + \overline{\mathtt{a}}(\overline{\mathtt{b}}(x)) \leftharpoonup \overline{\mathtt{c}}(x) \\
[a \rhd^{\mathtt{not}} b] &:= \overline{\mathtt{a}}(x) \leftharpoonup \mathtt{b}(x) \ + \ \mathtt{a}(x) \leftharpoonup \overline{\mathtt{b}}(x) \\
[\rhd^{0} a] &:= x \leftharpoonup \overline{\mathtt{a}}(x) \\
[\rhd^{1} a] &:= x \leftharpoonup \mathtt{a}(x)
\end{aligned}$$

The intuition behind this encoding is that we are handling a stack of needed values, $\mathtt{v}(\cdot)$ (resp. $\overline{\mathtt{v}}(\cdot)$) meaning "we need the value $\mathtt{1}$ (resp. $\mathtt{0}$) at v". The flows associated to gates are then meant to handle this stack, popping and pushing needed values.

Then, to a circuit (G, o) we associate the unary query

$$\Big(\mathtt{o}(\star) \,,\ \sum_{e \text{ vertex of } G} [e] \,,\ \star \Big)$$

This query succeeds iff the circuit computes the value $\mathtt{1}$ at the gate o: the data $\mathtt{o}(\star)$ initiates a stack with the intuitive meaning "we need the value $\mathtt{1}$ at o", the encodings of edges propagate the needed values to the point where they can be "popped" if the correct $\rhd^{0} x$ or $\rhd^{1} x$ is available. The query succeeds if we can derive the goal $\star$ — i.e. the empty stack — with the intuitive meaning "all the needed values have been provided".

Note the parallel nature of this way of solving the problem: when we compute the saturation of (the encoding of) the query, we unify the terms that match at any point of the circuit without having to worry in which order we perform the operations.

For instance, the two elements $\mathtt{a}(x) \leftharpoonup \mathtt{o}(\star)$ and $\mathtt{b}(x) \leftharpoonup \mathtt{o}(\star)$ of $[a, b \rhd^{\mathtt{or}} o(\star)]$ would be unified with $o(\star) \leftharpoonup \star$, providing two flows $\mathtt{a}(x) \leftharpoonup \star$ and $\mathtt{b}(x) \leftharpoonup \star$. Those flows would be, at the next execution step, tested for unification against all (provided we respect the increasing/decreasing discipline at work in Definition IV.3) the other flows and so on, without having to wait to know whether a or b will hold the value $\mathtt{1}$. A partial evaluation happens at any point of the graph, independently of the input: $[a, b \rhd^{\mathtt{and}} c]$ and $[c \rhd^{\mathtt{not}} d]$ will give after one step of evaluation the flows $\mathtt{a}(\mathtt{b}(x)) \leftharpoonup \overline{\mathtt{d}}(x)$, $\overline{\mathtt{a}}(x) \leftharpoonup \mathtt{d}(x)$ and $\overline{\mathtt{b}}(x) \leftharpoonup \overline{\mathtt{d}}(x)$. The execution does not have to sequentially wait for the propagation of the needed values.

Finally, let us say a word about the stabilization time of $\mathtt{satur}(\cdot)$ (Definition IV.3) in this case. Given a circuit with S gates, we are dealing with flows of height at most 2, written with at most S different symbols. In view of Proposition IV.4 we have that the iterations of $\mathtt{short}(\cdot)$ will stabilize in at most $(S^2 + S + 1)^2$ steps. A bound that is rough, due the absence of optimization and fine-grained analysis of the procedure.

[5] A *directed hypergraph* is given as a set of vertices V and a set of edges $E \subseteq \mathcal{P}(V) \times \mathcal{P}(V)$. We say that $(S, T) \in E$ is an edge *from* S *to* T.
We consider labeled edges and write $x_1, \ldots, x_n \rhd^{\mathtt{k}} y_1, \ldots, y_m$ an edge labeled by $\mathtt{k}$ from $\{ x_1, \ldots, x_n \}$ to $\{ y_1, \ldots, y_m \}$.

VI. Perspectives

This article extends modularly on our previous approaches [23], [24], [27], [28] 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.

We saw that to a *qualitative* constraint on the way memory is handled by automata corresponds a *syntactical* 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 were able to adapt the mechanism of pre-computation of transitions, known as memoization, in a setting where logic programs are represented as algebraic objects. This technique — that we called the saturation technique — computes shortcuts in a logic program in order to decide its nilpotency faster.

This approach to complexity was earlier based on von Neumann algebras [22], [23], [24] and now explore unification theory [21], [27], [28]: it is emerging as a meeting point for computer science, logic and mathematics. This raises multiple questions and perspectives.

A number of interrogations come from the relations of this work 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 is their computational nature. In the distance, one may hope for a connection between our approach and ongoing work on higher order trees and model checking; all alike, one could study the interaction between observations and one-way integers — briefly discussed in earlier work [28] — 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 [43] that already developed and used — although with a different, much more logical, meaning — the notion of shortcut in the evaluation.

Finally, we also aim at representing functional computation, by considering a more general notion of observation that would allow for expressing the notion of output.

References

[1] P. van Emde Boas, "Machine models and simulation," in *Handbook of Theoretical Computer Science. volume A: Algorithms and Complexity*, J. Van Leeuwen, Ed. Elsevier, 1990, pp. 1–66.

[2] U. Dal Lago, "A short introduction to implicit computational complexity," in *ESSLLI*, ser. LNCS, N. Bezhanishvili and V. Goranko, Eds., vol. 7388. Springer, 2011, pp. 89–109.

[3] S. J. Bellantoni and S. A. Cook, "A new recursion-theoretic characterization of the polytime functions," *Comput. Complex.*, vol. 2, pp. 97–110, 1992.

[4] D. Leivant, "Stratified functional programs and computational complexity," in *POPL*, M. S. Van Deusen and B. Lang, Eds. ACM Press, 1993, pp. 325–333.

[5] P. Neergaard, "A functional language for logarithmic space," in *APLAS*, ser. LNCS, W.-N. Chin, Ed. Springer, 2004, vol. 3302, pp. 311–326.

[6] J.-Y. Girard, "Linear logic," *Theoret. Comput. Sci.*, vol. 50, no. 1, pp. 1–101, 1987.

[7] ——, "Light linear logic," in *LCC*, ser. LNCS, D. Leivant, Ed. Springer, 1995, vol. 960, pp. 145–176.

[8] V. Danos and J.-B. Joinet, "Linear logic & elementary time," *Inf. Comput.*, vol. 183, no. 1, pp. 123–137, 2003.

[9] Y. Lafont, "Soft linear logic and polynomial time," *Theoret. Comput. Sci.*, vol. 318, no. 1, pp. 163–180, 2004.

[10] P. Baillot and K. Terui, "Light types for polynomial time computation in lambda-calculus," in *LICS*. IEEE Computer Society, 2004, pp. 266–275.

[11] U. Dal Lago and U. Schöpp, "Functional programming in sublinear space," in *ESOP*, ser. LNCS, A. D. Gordon, Ed., vol. 6012. Springer, 2010, pp. 205–225.

[12] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca, "An implicit characterization of pspace," *ACM Trans. Comput. Log.*, vol. 13, no. 2, pp. 18:1–18:36, 2012.

[13] J.-Y. Girard, "Towards a geometry of interaction," in *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference held June 14-20, 1987*, ser. Categories in Computer Science and Logic, J. W. Gray and A. Ščedrov, Eds., vol. 92. AMS, 1989, pp. 69–108.

[14] V. Danos, "La logique linéaire appliquée à l'étude de divers processus de normalisation (principalement du λ-calcul)," Ph.D. dissertation, Université Paris VII, 1990.

[15] T. Seiller, "Logique dans le facteur hyperfini : géometrie de l'interaction et complexité," Ph.D. dissertation, Université de la Méditerranée, 2012. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00768403/

[16] ——, "Interaction graphs: Graphings," *Submitted*, 2014. [Online]. Available: http://arxiv.org/pdf/1405.6331

[17] J.-Y. Girard, "Geometry of interaction 1: Interpretation of system F," *Studies in Logic and the Foundations of Mathematics*, vol. 127, pp. 221–260, 1989.

[18] ——, "Geometry of interaction V: logic in the hyperfinite factor," *Theoret. Comput. Sci.*, vol. 412, no. 20, pp. 1860–1883, Apr. 2011.

[19] T. Seiller, "A correspondence between maximal abelian sub-algebras and linear logic fragments," *Submitted*, 2014. [Online]. Available: http://arxiv.org/pdf/1408.2125

[20] J.-Y. Girard, "Geometry of interaction III: accommodating the additives," in *Advances in Linear Logic*, ser. London Math. Soc. Lecture Note Ser., J.-Y. Girard, Y. Lafont, and L. Regnier, Eds. CUP, Jun. 1995, no. 222, pp. 329–389.

[21] M. Bagnol, "On the resolution semiring," Ph.D. dissertation, Aix-Marseille Université – Institut de Mathématiques de Marseille, 2014.

[22] J.-Y. Girard, "Normativity in logic," in *Epistemology versus Ontology*, ser. Logic, Epistemology, and the Unity of Science, P. Dybjer, S. Lindström, E. Palmgren, and G. Sundholm, Eds. Springer, 2012, vol. 27, pp. 243–263.

[23] C. Aubert and T. Seiller, "Characterizing co-nl by a group action," *MSCS (FirstView)*, pp. 1–33, 12 2014.

[24] ——, "Logarithmic space and permutations," *Inf. Comput.*, 2015, to appear.

[25] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *J. ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965.

[26] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov, "Complexity and expressive power of logic programming," *ACM Comput. Surv.*, vol. 33, no. 3, pp. 374–425, 2001.

[27] C. Aubert and M. Bagnol, "Unification and logarithmic space," in *RTA-TLCA*, ser. LNCS, G. Dowek, Ed., vol. 8650. Springer, 2014, pp. 77–92.

[28] C. Aubert, M. Bagnol, P. Pistone, and T. Seiller, "Logic programming and logarithmic space," in *APLAS*, ser. LNCS, J. Garrigue, Ed., vol. 8858. Springer, 2014, pp. 39–57.

[29] S. A. Cook, "Linear time simulation of deterministic two-way pushdown automata," in *IFIP Congress (1)*. North-Holland, 1971, pp. 75–80.

[30] C. Dwork, P. C. Kanellakis, and J. C. Mitchell, "On the sequential nature of unification," *J. Log. Program.*, vol. 1, no. 1, pp. 35–50, 1984.

[31] F. Baader and T. Nipkow, *Term rewriting and all that.* CUP, 1998.

[32] P. Baillot and M. Pedicini, "Elementary complexity and geometry of interaction," *Fund. Inform.*, vol. 45, no. 1–2, pp. 1–31, 2001.

[33] M. Ladermann and H. Petersen, "Notes on looping deterministic two-way pushdown automata," *Inf. Process. Lett.*, vol. 49, no. 3, pp. 123–127, 1994.

[34] J. Hartmanis, "On non-determinancy in simple computing devices," *Acta Inform.*, vol. 1, no. 4, pp. 336–344, 1972.

[35] S. A. Cook, "Characterizations of pushdown machines in terms of time-bounded computers," *J. ACM*, vol. 18, no. 1, pp. 4–18, 1971.

[36] K. W. Wagner and G. Wechsung, *Computational Complexity*, ser. Mathematics and its Applications. Springer, 1986, vol. 21.

[37] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Time and tape complexity of pushdown automaton languages," *Inform. Control*, vol. 13, no. 3, pp. 186–206, 1968.

[38] D. Michie, ""Memo" functions and machine learning," *Nature*, vol. 218, pp. 19–22, Apr. 1968.

[39] T. Amtoft and J. L. Träff, "Partial memoization for obtaining linear time behavior of a 2dpda," *Theoret. Comput. Sci.*, vol. 98, no. 2, pp. 347–356, 1992.

[40] N. Andersen and N. D. Jones, "Generalizing cook's transformation to imperative stack programs," in *Results and Trends in Theoretical Computer Science*, ser. LNCS, J. Karhumäki, H. A. Maurer, and G. Rozenberg, Eds., vol. 812. Springer, 1994, pp. 1–18.

[41] R. Glück, "Simulation of two-way pushdown automata revisited," in *Festschrift for Dave Schmidt*, ser. EPTCS, A. Banerjee, O. Danvy, K.-G. Doh, and J. Hatcliff, Eds., vol. 129, 2013, pp. 250–258.

[42] R. E. Ladner, "The circuit value problem is log space complete for p," *ACM SIGACT News*, vol. 7, no. 1, pp. 18–20, 1975.

[43] V. Danos and L. Regnier, "Reversible, irreversible and optimal lambda-machines," *Theoret. Comput. Sci.*, vol. 227, no. 1-2, pp. 79–97, 1999.