Promise Algebra: a Functional Approach to Non-Deterministic Computations

Eugenia Ternovska[1][0000−0003−0751−4031]

Simon Fraser University, Canada ter@sfu.ca

Abstract. We introduce an algebra that formalizes non-deterministic computations in a semi-deterministic way. The algebra is an algebra of partial functions. It has in an equivalent form of a linear-time (as opposite to branching-time) Dynamic Logic. A crucial part in our construction is played by a new kind of Choice functions that are history-dependent. We show that, under a restriction on the length of Choice functions, a fragment of the logic captures precisely the complexity class NP with respect to its data complexity, while, for the unrestricted language, the upper bound, again, with restricted length, is PSPACE.

Keywords: Algebra of partial functions · Choice functions · Dynamic logic

1 Introduction

Algebras connected to Dynamic Logics are most known from the seminal work of Kozen, e.g., [28], who studied Kleene algebra extensively. The properties of such algebras are still an active area of research. For example, recently, Kleene algebras were studied in, e.g., [27], [38], [29]. Kleene algebras, and associated Dynamic logics [14, 20], include constructs of Regular Expressions $\cup$ and *. Such contracts are, by their nature, non-deterministic. That is, in an algebra of actions, even if all atomic actions are deterministic, algebraic expressions with these operations express non-deterministic computations.

However, a different formalization of non-deterministic computations is possible. One can allow non-determinism in atomic actions only, and require all algebraic operations to be *function-preserving*. That means, *if* atomic expressions are functional binary relations, then so are all algebraic expressions. Such an approach provides an alternative perspective by containing non-determinism at the atomic level. Instead of general binary relations specifying what is true (before, after) an action, here we work with *partial functions*. This is similar to the work of Jackson and Stokes [25] and McLean [32]. In fact, the work [25] is particularly relevant, because it introduces some connectives we use.

The idea of a functional approach to non-determinism seems very restrictive. So, the natural questions are: *Is this approach viable? Can we express interesting computational problems? Can we capture complexity classes?* We answer these questions affirmatively. In this paper, we introduce an algebra that formalizes non-deterministic computations in a semi-deterministic way. It is an algebra of

string-to-string transductions. Specifically, it is an algebra of partial functions on words over the alphabet of relational τ-structures over the same domain. The algebra has a two-level syntax, and thus, two parameters to control its expressive power. The top level defines algebraic expressions, and the bottom level specifies atomic transitions. We define a simple secondary logic for representing atomic transitions, which is a modification of conjunctive queries. The algebra has in an equivalent form of a linear-time (as opposite to branching-time such as PDL) Dynamic Logic, where terms describing computational processes or programs appear inside modalities. We believe that this is the first linear-time logic that is also an *algebra of partial functions*.

A crucial part is our construction is a new kind of Choice functions that are history-dependent. The functions resolve atomic non-determinism, and serve as certificates for computational problems specified by algebraic terms. We show that, under a restriction on the length of Choice functions, a fragment of the logic captures precisely the complexity class NP with respect to its data complexity. We believe that ours is the first such capturing result by an algebraic function-preserving logic.[1] The upper bound, under the length restriction, is PSPACE. With this logic, we can represent both reachability and counting examples, which is not possible in Datalog. Proofs and details of the related work are given in the Appendix. We assume familiarity with the basic notions of first-order (FO) and second-order (SO) logic (see, e.g., [12]), and use ':=' to mean "denotes" or "is by definition".

2 Algebra: Syntax

In this section, we define the syntax of our algebra, and introduce some initial intuitions regarding the meaning of its operations. The purpose of the algebra is to talk about Choice functions of a certain kind. The Choice functions will later be considered as certificates for computational problems specified by algebraic terms, or *promises*. If such a promise is given for a term (intuitively, a non-deterministic program), the computation is guaranteed to proceed successfully. With this intuition in mind, we call this algebra a *Promise Algebra*. A promise is like an elephant in a room – it is there, but is never mentioned explicitly.[2]

Formally, the algebra is an algebra of (functional) binary relations on strings of relational structures. It has a two-level syntax. The top level (defined in this section) specifies algebraic expressions. The algebraic operations are *dynamic counterparts* of classical logic connectives – negation, conjunction, disjunction – and iteration (the Kleene star, or reflexive transitive closure). Unlike classical connectives, these operations are *function-preserving* in the sense that if atomic elements are functional binary relations, then so are all algebraic expressions

[1] The first, and much celebrated, result on capturing NP is Fagin's [13]. It is formulated for Second-Order Existential logic.

[2] We use ε to denote the presence of choice, but no concrete certificate is present in the language. Epsilon can be thought of as a free *function variable*.

built up from them. The bottom level of the formalism specifies atomic expressions (intuitively, actions) in a separate logic (explained later).

We now remind the reader the notion of a (relational) structure. Let τ be a relational vocabulary, which is finite (but is of an unlimited size). Let $\tau := \{S_1, \ldots, S_n\}$, each S_i has an associated arity r_i, and A be a non-empty set. A *τ-structure* $\mathfrak{A}$ over domain $\mathrm{dom}(\mathfrak{A}) := A$ is $\mathfrak{A} := (A;\ S_1^{\mathfrak{A}}, \ldots, S_n^{\mathfrak{A}})$, where $S_i^{\mathfrak{A}}$ is an r_i-ary relation called the *interpretation* of S_i. In this paper, all structures are finite. If $\mathfrak{A}$ is a τ-structure, $\mathfrak{A}|_\sigma$ is its restriction to a sub-vocabulary σ. We now fix a relational vocabulary τ, and assume it is partitioned into "inputs" (or EDB relations, in the database terminology), and unary "register" symbols, interpreted as singleton-set relations:[3]

$$\tau \ := \ \tau_{\mathrm{EDB}} \uplus \tau_{\mathrm{reg}}. \tag{1}$$

The details and the formal requirements on these two vocabularies will become clear when a particular logic of the bottom level is explained later in the paper. We only mention now that, intuitively, the interpretations of EDB (or input) relations never change, while the interpretations of the registers (singleton sets) are updated by applications of atomic modules.

Let a set $\mathcal{M}$ of atomic module symbols, denoting non-deterministic atomic actions, be fixed. Intuitively, the actions are atomic updates of relational structures. The atomic updates are combined into complex updates (formally, algebraic expressions, or terms) using algebraic operations. The set *Terms* of the well-formed terms of the algebra are defined as:

$$t ::= \varepsilon a \mid \mathrm{id} \mid \ \sim t \mid t\,;t \mid t \sqcup t \mid t^{\uparrow} \mid P = Q \mid \ \textsc{bg}(P_{now} \neq Q), \tag{2}$$

where $a \in \mathcal{M} \neq \varnothing$ and ε is a free function variable ranging over Choice functions. The syntax allows only *one* such (free) variable per an algebraic expression.[4] We require that $P, Q \in \tau_{\mathrm{reg}}$. The subscript "*now*" in P_{now} is not a part of the syntax. It is added for the reader temporarily, to memorize that the "register" P, shown on the left, refers to the current position in the computation. The content of P must be different from the values in Q ever before (denoted '$\textsc{bg}$', for Back Globally).

In the table, we list the variables first, and then the partial functions of the algebra that are split into three groups – nullary (constant), unary and binary partial mappings. Unary functions take one other function as an argument, binary – two. The column in the middle represent the range of the variable or the type of the corresponding partial function. Our notations are: $\mathbf{U}$ is the set of all relational τ-structures over the same domain. In this paper, the domain is finite. $\mathsf{CH}(\mathbf{Tr})$ is a set of Choice functions depends on a transition system $\mathbf{Tr}$, to be defined later in the paper, and $\mathcal{F}$ denotes partial functions on $\mathbf{U}^+$ (non-empty strings over the alphabet $\mathbf{U}$).

[3] EDB is common term in Database theory. It stands for Extensional Database, which is a relational structure.

[4] Syntactically, the variable ε, that occurs free in the algebraic expressions, is not really necessary. We use it to emphasize the implicit presence of a concrete Choice function.

x	$\mathbf{U}^+$	string variable
ε	$\mathsf{CH}(\mathbf{Tr})$	Choice function variable
f, g	$\mathcal{F}$	partial function variables on $\mathbf{U}^+$
$\mathrm{id}(x)$		Identity on $\mathbf{U}^+$
$\varepsilon a(h/\varepsilon)(x)$		Action, for all $a \in \mathcal{M}$, $h \in \mathsf{CH}(\mathbf{Tr})$
$\textsc{bg}(P \neq Q)(x)$	$\mathcal{F}$	Back Globally Non-Equal
$(P = Q)(x)$		Equal
$\sim f(x)$	$\mathcal{F} \to \mathcal{F}$	Anti-Domain (Unary Negation)
$f^{\uparrow}(x)$		Maximum Iterate
$(f \sqcup g)(x)$	$\mathcal{F}^2 \to \mathcal{F}$	Preferential Union
$(f\,;g)(x)$		Sequential Composition

Intuitively, partial functions $\mathcal{F}$ on strings in $\mathbf{U}^+$ represent programs, and $\mathbf{U}^+$ contains traces of their executions. A program takes a trace of a previous program, and either produces a longer trace, or performs no changes, if it is a successful test. Applying a term on a string of length one corresponds to applying a program on an input structure, e.g., a graph in 3-Colourability.

The set $\mathcal{F}$ of partial functions on $\mathbf{U}^+$ contains all functional constant (i.e., nullary) operations (the rows marked with $\mathcal{F}$), and is closed under unary (marked $\mathcal{F} \to \mathcal{F}$) and binary (marked $\mathcal{F}^2 \to \mathcal{F}$) operations.

Nullary (constant mappings on strings in $\mathbf{U}^+$): $\mathrm{id} \in \mathcal{F}$, $\varepsilon a(h/\varepsilon) \in \mathcal{F}$ for all a in $\mathcal{M}$ and $h \in \mathsf{CH}(\mathbf{Tr})$, and $\{(P = Q), \textsc{bg}(P \neq Q)\} \subset \mathcal{F}$, for all $P, Q \in \tau_{\mathrm{reg}}$. Here, id is the Identity function (Diagonal relation) on strings in $\mathbf{U}^+$. Each $a \in \mathcal{M}$ is an atomic action, which we call a "module", a binary relation, that, intuitively, updates the interpretations of some of the symbols in τ_{reg} non-deterministically, i.e., multiple outcomes of a are possible. Epsilon ε, inspired by Hilbert's Choice operator [22], indicates that, at each particular point in time, only one outcome of a, out of those possible, is selected. With a *semantic* instantiation of a concrete Choice function h for ε in εa, denoted $\varepsilon a(h/\varepsilon)$, the relation a becomes a partial function in $\mathcal{F}$. Back Globally Non-Equal, denoted $\textsc{bg}(P \neq Q)(x)$, checks if the value stored in register Q at any point earlier in string x is *different* from the value in P now.[5] Equality check $(P = Q)$ compares current interpretations of P and Q. Thus, access to domain elements is allowed only via atomic updates (modules in $\mathcal{M}$ specified in a secondary logic) and (in)equality checks only.

Unary $\mathbf{op}_1 : \mathcal{F} \to \mathcal{F}$ (partial mappings on strings in $\mathbf{U}^+$ that depend on one partial function): $\mathbf{op}_1 \in \{\sim, ^{\uparrow}\}$. Each operation $\mathbf{op}_1$ in this set takes a function in $\mathcal{F}$ and modifies it according to the semantics of $\mathbf{op}_1$. The modified operation is applied on strings in $\mathbf{U}^+$. Anti-Domain $\sim f(x)$ checks if there is no outgoing f-transition from the state represented by the string x; Maximum Iterate $f^{\uparrow}(x)$ is a "determinization" of the Kleene star $f^*(x)$ (reflexive transitive closure). It outputs only the longest, in terms of the number of f-steps, transition out of all possible transitions from the same state produced by the Kleene star.[6]

[5] The $\textsc{bg}$ construct is related to the modality "H" used in temporal logics.

[6] For readers familiar with the μ operator, we mention that $f^{\uparrow} := \mu Z.(\sim f \cup f\,; Z)$, although these constructs are not in our language.

Binary $\mathbf{op}_2 : \mathcal{F}^2 \to \mathcal{F}$ (partial mappings on strings in $\mathbf{U}^+$ that depend on two functions): $\mathbf{op}_2 \in \{;, \sqcup\}$ They take two functions as arguments, and combine them to obtain a (partial) mapping on $\mathbf{U}^+$. Sequential Composition $(f\,;g)(x)$ is the standard function composition $g(f(x))$. Preferential Union $(f \sqcup g)(x)$ applies f and returns its result; but, if f is not defined, it applies g. We routinely omit the string variable x and work entirely with partial functions in $\mathcal{F}$, as is typical in algebraic settings such as groups, algebras of functions and binary relations.

3 Semantics

Let $\mathcal{M} \neq \varnothing$ be a set of atomic actions, and $\mathbf{U}$ be the set of all relational τ-structures over the same domain. In this paper, the domain is finite. The semantics of the algebra makes use of an underlying labelled transition system (a labelled graph): $\mathbf{Tr}[\![\cdot]\!] : \mathcal{M} \to \mathbf{U} \times \mathbf{U}$. The relation $\mathbf{Tr}[\![\cdot]\!]$ can be specified in a (secondary) logic that constitutes the bottom level of the algebra. In Section 5, we give an example of such a logic. For now, we view $\mathbf{Tr}[\![a]\!] \subseteq \mathbf{U} \times \mathbf{U}$, for $a \in \mathcal{M}$, as an arbitrary binary relation, not necessarily functional. In this sense, atomic transitions are non-deterministic. Choice functions (to be defined) semantically resolve atomic non-determinism, while taking the history into account.[7]

A *tree* over alphabet $\mathbf{U}$ is a (finite or infinite) nonempty set $\mathcal{T} \subseteq \mathbf{U}^+$ such that for all $x \cdot c \in \mathcal{T}$, with $x \in \mathbf{U}^+$ and $c \in \mathbf{U}$, we have $x \in \mathcal{T}$. The elements of $\mathcal{T}$ are called *nodes*, and a letter, which is an element of $\mathbf{U}$, e.g., $\mathfrak{A} \in \mathbf{U}$ is the *root* of $\mathcal{T}$. For every $x \in \mathcal{T}$, the nodes $x \cdot c \in \mathcal{T}$ where $c \in \mathbf{U}$ are the *children* of x. A node with no children is a *leaf*.

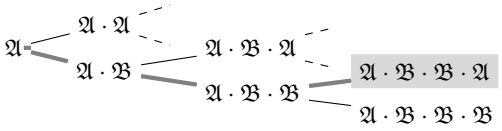

Fig. 1: A tree over $\mathbf{U}$. Let $a_{17} \in \mathcal{M}$ and $(\mathfrak{A}, \mathfrak{B}) \in \mathbf{Tr}[\![a_{17}]\!]$, $(\mathfrak{B}, \mathfrak{A}) \in \mathbf{Tr}[\![a_{17}]\!]$, $(\mathfrak{B}, \mathfrak{B}) \in \mathbf{Tr}[\![a_{17}]\!]$. A concrete Choice function $h : \mathcal{M} \to (\mathbf{U}^+ \rightharpoonup \mathbf{U}^+)$ maps a_{17} to a set of mapping containing, for example, $\mathfrak{A} \cdot \mathfrak{B} \cdot \mathfrak{B} \mapsto \mathfrak{A} \cdot \mathfrak{B} \cdot \mathfrak{B} \cdot \mathfrak{A}$, and other such mappings shown by the thicker edges in the tree. Term $t := \varepsilon a_{17}\,;\varepsilon a_{17}\,;\varepsilon a_{17}(h/\varepsilon)$, with this specific h, makes transitions along the branch in this tree shown by the thicker edges. There is a one-to-one correspondence between that branch and the string shown by the shaded area, which is a *trace* of t from the input structure $\mathfrak{A}$ that corresponds to h. However, not all strings are traces.

Notations for Strings and Partial Functions We use $\mathbf{e}$ to denote the empty string; $v(\mathit{first})$ and $v(\mathit{last})$ to denote the first and the last letter in string v, respectively; and $v \sqsubseteq w$ to denote that v is a prefix of string w. We use the following notation for partial functions: $A \rightharpoonup B := \bigcup_{C \subseteq A}(C \to B)$.

[7] Notice that concrete Choice functions are not a part of the syntax. The syntax, summarized in (2) and detailed in the table, has only a Choice function *variable* ε, one per term, a "place holder" for a Choice function.

Choice Functions Intuitively, a string v that starts from an input structure $\mathfrak{A}$ represents a history, i.e., a sequence of states – elements of $\mathbf{U}$ – that is a trace of some program (a term). If an action a is executatable at the last structure in string v (which is indicated by the presence of the corresponding pair of structures in the relation $\mathbf{Tr}[\![a]\!] \subseteq \mathbf{U} \times \mathbf{U}$), then Choice function h selects one possible outcome of the atomic module a, out of those possible according to $\mathbf{Tr}[\![a]\!]$, at that time point, see Figure 1. The outcome is history-dependent, that is, for the same module a, we may have different outcomes at different time points. Let $\mathcal{M}$ be a finite non-empty set of module symbols, and $\mathbf{U}$ be an alphabet. A *Choice function* is a partial function $h : \mathcal{M} \to (\mathbf{U}^+ \rightharpoonup \mathbf{U}^+)$ defined by the functional binary relation $h(\varepsilon a) := \{(v, w) \mid v \sqsubseteq w \wedge (v(last), w(last)) \in \mathbf{Tr}[\![a]\!]\big)\}$. The set of all Choice functions is denoted $\mathsf{CH}(\mathbf{Tr})$. Since, by definition, $h(\varepsilon a)$ is a *functional* binary relation, at an application of a, each Choice function resolves atomic non-determinism, in its own way. Note that a "no-change" transition is allowed – it extends a string v by repeating the same letter $v(last) \in \mathbf{U}$.

Extension of Choice Functions to All Terms So far, Choice functions were defined on the (non-empty) domain $\mathcal{M}$ of atomic module symbols. The extension $\bar{h}$ of h, where $h \in \mathsf{CH}(\mathbf{Tr})$, provides semantics to all algebraic terms *Terms* in (2) as string-to-string transductions, or partial functions on non-empty strings of relational structures $\mathbf{U}^+$:

$$\bar{h} : \mathit{Terms} \to (\mathbf{U}^+ \rightharpoonup \mathbf{U}^+). \tag{3}$$

The transductions can also be viewed as (functional) binary relations, where for each term t, we have $\bar{h}(t) \subseteq \mathbf{U}^+ \times \mathbf{U}^+$. Below, we assume that all strings belong to $\mathbf{U}^+$, and $h, h' \in \mathsf{CH}(\mathbf{Tr})$.

1. $\bar{h}(\varepsilon a) := h(\varepsilon a)$ for $a \in \mathcal{M}$.
2. $\bar{h}(\mathrm{id}) := \{(v, v)\}$, i.e., it is the Identity (Diagonal) relation on $\mathbf{U}^+$.
3. $\bar{h}(\sim t) := \{(v, v) \mid \neg\big(\exists h' \exists w\ ((v, w) \in \bar{h}'(t))\big)\}$, i.e., $\sim t$ is a test (a subset of Identity) that is true for those strings v where t is not executable. We'll discuss tests more, shortly.
4. $\bar{h}(t\,;g) := \{(v, w) \mid \exists\ u\ \big((v, u) \in \bar{h}(t)\ \wedge\ (u, w) \in \bar{h}(g)\big)\}$.
5. $\bar{h}(t \sqcup g) := \begin{cases} \bar{h}(t) & \text{if } \bar{h}(t) \neq \varnothing, \\ \bar{h}(g) & \text{if } \bar{h}(t) = \varnothing. \end{cases}$
6. $\bar{h}(t^{\uparrow}) := \{(v, w) \mid (v, v) \in \bar{h}(\sim t) \wedge v = w$
   $\vee\ \exists u\ \big(v \sqsubseteq u \sqsubseteq w\ \wedge\ (v, u) \in \bar{h}(t)\ \wedge\ (u, w) \in \bar{h}(t^{\uparrow})\big)\}$.
7. $\bar{h}(P = Q) := \{(v, v) \mid Q^{v(last)} = P^{v(last)}\}$. Recall that $v(last)$ refers to the last letter in string v, so the condition says that the interpretations of P and Q at the end of v coincide.
8. $\bar{h}(\ \textsc{bg}(P \neq Q)) := \{(v, v) \mid\ \neg\exists w\ \big(w \sqsubseteq v\ \wedge\ P^{v(last)} = Q^{w(last)}\big)\}$. [8] [9]

[8] Note that the intended use of the algebra is to evaluate algebraic expressions with respect to an input structure, that is a one-letter string. In this use, the $\textsc{bg}$ construct never looks into the "pre-history", since, in that use, all strings start from an input structure and are traces of some computations.

[9] An alternative definition of $\textsc{bg}(P \neq Q)$ is possible, where the scope of the condition is specified explicitly (by e.g., adding parentheses). But, it is not needed since the

Note that, for a given Choice function h, its extension $\bar{h}$ to a particular term may not exist. For example, two atomic modules may not compose because of a clash of their pre- and post-conditions. In those cases, $\bar{h}(t) = \varnothing$, and we say that h *is not extendable to* t. Observe that, for any t and h, if $(v, w) \in \bar{h}(t)$ then $v \sqsubseteq w$. The result of cutting v from the left end of w is called a *delta (or trace) of* t, and is denoted $v \backslash w$, also known as the right residual.

Tests vs Processes Tests play a special role in our formalisation of non-deterministic computations as they specify computational decision problems. While tests are, essentially, "yes" or "no" questions, processes specify the actual content of these questions. A term t is called a *test* if, for all h, the extension $\bar{h}$ of h is a subset of the Identity (Diagonal) relation on $\mathbf{U}^+$. Otherwise, t is called a *process*. The cases for tests in the definition of an extension of Choice function are 2,3,7 and 8. It is possible that an atomic action does not make a transition to another state in the transition system $\mathbf{Tr}[\![\cdot]\!]$. However, such an action would not be considered a test because it still extends the current string by repeating the same letter. The delta of a test is the empty string $\mathbf{e}$.

Maximum Iterate vs the Kleene Star To give a comparison of Maximum Iterate with the Kleene star (i.e., the reflexive transitive closure, a construct known from Regular Languages and Kleene algebras), we show how to define the two operators side-by-side, inductively.

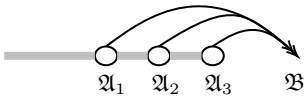

Fig. 2: Maximum Iterate:
$t^{\uparrow}_0 := \sim t, \quad t^{\uparrow}_{n+1} := t^{\uparrow}_n \,;\, t, \quad t^{\uparrow} := \bigcup_{n\in\mathbb{N}} t^{\uparrow}_n$.
It is a deterministic operator (a partial function).

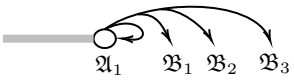

Fig. 3: The Kleene Star:
$t^{*}_0 := \mathrm{id}, \quad t^{*}_{n+1} := t \,;\, t^{*}_n, \quad t^{*} := \bigcup_{n\in\mathbb{N}} t^{*}_n$.
It is a non-deterministic operator (not a function).

4 Dynamic Logic

We now provide an alternative (and equivalent) two-sorted version of the syntax in the form of a Dynamic logic. While the two formalizations are equivalent, in many ways, it is easier to work with the logic. The syntax is:

$$\begin{array}{c} t ::= \varepsilon a \,|\mathrm{id}|\, \sim t \mid t \,;\, t \mid t \sqcup t \mid t^{\uparrow} \mid P = Q \mid \textsc{bg}(P \neq Q) \mid \phi? \\ \phi ::= \; |t\rangle\phi \mid \top \mid \neg\phi \;\mid\; \phi \wedge \phi \;\mid\; \phi \vee \phi \end{array} \tag{4}$$

The expressions in the first line of (4) are typically called *process* terms, and those in the second line *state* formulae or *tests*. State formulae ϕ are "unary" in the same sense as $P(x)$ is a unary notation for $P(x, x)$. Semantically, they are

scope can be controlled by a clever use of fresh "registers", i.e., unary predicate symbols in τ_{reg}.

subsets of the identity relation on $\mathbf{U}^+$. The state formulae are shorthands that use the operations in the first line: $|t\rangle\,\phi := \sim\sim(t;\phi)$ is the "diamond" modality claiming the existence of a successful execution of t leading to ϕ. In addition, we define more familiar operations $\top := \mathrm{id}$, $\neg\phi := \sim\phi$, $\phi \wedge \psi := \phi\,;\psi$, $\phi \vee \psi := \phi \sqcup \psi$, and also $\phi? := \sim\sim\phi$, appearing in the top line of (4). In addition, we define $\mathrm{Dom}(t) := \sim\sim t$.

Semantics of State Formulae To interpret the Dynamic Logic (4), we need to provide semantics to both processes and state formulae. Processes are interpreted as before – as partial mappings (parameterized with h) on strings in $\mathbf{U}^+$, that can be viewed as string-to-string transductions. For state formulas we define, for all concrete Choice functions h and strings v,

$$\mathbf{Tr}, v \models \phi(h/\varepsilon) \quad \text{iff} \quad (v, v) \in \bar{h}(\phi). \tag{5}$$

In this paper, the relation $\mathbf{Tr}[\![\cdot]\!]$ (that is specified by a secondary logic at the bottom level of the algebra) is fixed, and is omitted from (5).

Implicit Quantification State formulae exhibit implicit quantification over the Choice function variable ε. Indeed, tests of the form $\sim\sim t$ represent the domain of t, and, intuitively, mean that there is a Choice function h such that $v \models \sim\sim t(h/\varepsilon)$. On the other hand, tests of the form $\sim t$ are universal, because they claim that there is no Choice function that corresponds to a successful execution of t. To summarize, we have the following implicit quantification over Choice functions:

$$\begin{array}{rl} \sim\sim t(x) - & \text{implicitly, } \exists\varepsilon, \\ \sim t(x) - & \text{implicitly, } \forall\varepsilon. \end{array} \tag{6}$$

These are *second-order* quantifiers, since we quantify over functions in $\mathsf{CH}(\mathbf{Tr})$. The quantifiers can interleave, since expressions of the form (6) may appear within the term t. The two kinds of tests (6) are of special interest in the study of computational problems specified by a term.

Programming Constructs It is well-known that in Propositional Dynamic Logic [14], imperative programming constructs are definable using a fragment of regular languages, see the Dynamic Logic book [20]. The corresponding language is called *Deterministic Regular (While) Programs* in [20].[10] In our case, imperative constructs are definable (cf. [25]) by:

$$\begin{array}{ll} \textbf{skip} := \mathrm{id}, & \textbf{fail} := \sim\mathrm{id}, \qquad \textbf{while } \phi \textbf{ do } t := (\phi?\,;t)^{\uparrow}\,;(\sim\phi?), \\ \textbf{if } \phi \textbf{ then } t \textbf{ else } t' := (\phi?\,;t) \sqcup t', & \textbf{repeat } t \textbf{ until } \phi := t\,;((\sim\phi?)\,;t)^{\uparrow}\,;\phi?. \end{array}$$

Thus, importantly, the non-determinism of operations $*$ and $\cup$ of regular languages is not needed to formalize these programming constructs.

We now formulate the main computational task we study in this paper. The task amounts to satisfiability, with respect to a one-letter string, of a state

[10] Please note that Deterministic Regular expressions and the corresponding Glushkov automata are unrelated to what we study here. In those terms, expression $a\,;a^*$ is Deterministic Regular, while $a^*\,;a$ is not. Both expressions are not in our language.

formula in certain form. That is, the task is a particular case of satisfying a state formula, cf. (5). Importantly, we define a computational problem specified by an algebraic term, as a class of relational structures.

Problem: **Main Computational Task**		
Given: A τ-structure $\mathfrak{A}$ and term t.		
Question:	$\mathfrak{A} \models \|t\rangle\top$?	(7)

Intuitively, when the answer is yes, (7) says that there is a successful execution of t at the input structure $\mathfrak{A}$, witnessed by a Choice function h. The study of the computational task (7) is the main goal of this paper. In particular, we are interested in the data complexity of this task, where the formula is fixed, and the input structures vary [46].[11]

The modality in the main task is existential, see (6), since $|t\rangle\top = \mathrm{Dom}(t) = \sim\sim t$. Dually to this task, the complement of the problem (7) can be defined, using the fact that $\sim t$ is the complement of $\sim\sim t$. For instance, if a problem in the complexity class NP is specified using the, implicitly, existential, term $\sim\sim t$, then its complement, a problem in co-NP, is specified by the term $\sim t$, with an implicit universal quantifier. The reader who is familiar with some elements of Descriptive Complexity (see, e.g., Immerman's book [24]), might notice an analogy with the connection between second-order quantifier alternations and the polynomial time hierarchy. In this paper, we are mostly interested in the existential side, specifically, NP.

A *computational problem specified by a process-term* t is an isomorphism-closed class $\mathcal{P}_t$ of τ-structures $\mathfrak{A}$ such that a structure $\mathfrak{A}$ is in this class if and only if (7) holds, that is, there exists h that witnesses $\mathfrak{A} \models |t\rangle\top$.[12] Here, we assume that, in $\mathfrak{A}$, all relational "register" symbols in τ_{reg} are interpreted by a special "blank" element, and the interpretation of τ_{EDB} describes the input to the problem. Intuitively, the class $\mathcal{P}_t$ contains all structures $\mathfrak{A}$ such that a successful execution of t on input $\mathfrak{A}$ is possible, or t is *defined* on input $\mathfrak{A}$.

5 A Logic for Atomic Modules

In this section, first, we discuss the Law of Inertia, that must hold regardless of what secondary logic is used for specifying atomic modules. Second, we define a specific secondary logic used further in this paper. The logic is based on a modification of unary Conjunctive Queries (CQs). Finally, we discuss the machine model.

Recall that we have used the relation $\mathbf{Tr}[\![a]\!]$ that represents a transition system in the semantics of atomic modules, via Choice functions. We have, temporarily, treated it as an arbitrary binary relation. However, this relation has

[11] Notice that fixing the relation **Tr** means fixing a *formula* of the secondary logic. The size of the transition graph depends on the size of the input structure and the module specifications of the secondary logic.

[12] For natural secondary logics such as the one given in Section 5, isomorphism-closure holds for all structures satisfying the Main Task 7.

an important property that always holds for any update – *everything that is not explicitly modified in the update, must remain the same.* This property is commonly called the Law of Inertia, starting from[31].

We restrict atomic transductions so that the only relations that change from structure to structure are *unary* and *contain one domain element at a time*, similarly to registers in a Register Machine [40], as explained shortly.[13] EDB relations given on the input are never updated. All computational work happens in the registers.

We now specify a particular secondary logic for axiomatizing atomic transductions, that is similar to Conjunctive Queries (CQs). Intuitively, we take a *monadic* (unary) primitive positive (M-PP) relation, i.e., a unary relation definable by a unary CQ, and output, arbitrarily, only a *single* element contained in the relation, instead of the whole relation. There could be several such CQs in the same modules, applied ot once. A formal definition will be given shortly.

For readers familiar with Datalog (such a reader is invited to jump ahead to see the general form (12)), we mention informally that, at the bottom level of the algebra, we have a set of Datalog-like "programs", one per each atomic module symbol $a \in \mathcal{M}$. The "programs" are similar to Conjunctive Queries (non-recursive Datalog programs). The rules in such programs have a unary predicate symbol in the head of each rule. Each (simultaneous) application of the rules puts only *one* domain element into each unary IDB relation in the head, out of several possible. This creates non-determinism in the atomic module applications. The applications of the modules are controlled by algebraic expressions in (2) at the top level of the algebra. This logic is the second parameter that affects the expressive power of the language, in addition to the selection of the algebraic operations presented earlier. The logic is carefully constructed to ensure complexity bounds presented towards the end of the paper.

We start our formal exposition by reviewing queries, Conjunctive Queries, and PP-definable relations. Let C be a class of relational structures of some vocabulary τ. Following Gurevich [19], we say that an r-ary *C-global relation* q (a *query*) assigns to each structure $\mathfrak{A}$ in C an r-ary relation $q(\mathfrak{A})$ on $\mathfrak{A}$; the relation $q(\mathfrak{A})$ is the specialization of q to $\mathfrak{A}$. The vocabulary τ is the vocabulary of q. If C is the class of all τ-structures, we say that q is *τ-global*. Let $\mathcal{L}$ be a logic. A k-ary query q on C is $\mathcal{L}$-definable if there is an $\mathcal{L}$-formula $\psi(x_1, \ldots, x_k)$ with $x_1, \ldots, x_k$ as free variables such that for every $\mathfrak{A} \in C$,

$$q(\mathfrak{A}) = \{(a_1, \ldots, a_k) \in \mathcal{D}^k \mid \mathfrak{A} \models \psi(a_1, \ldots, a_k)\}. \tag{8}$$

In this case, $q(\mathfrak{A})$ is called the *output of q on $\mathfrak{A}$*. Query q is *unary* if $k = 1$. A *conjunctive query* (CQ) is a query definable by a FO formula in prenex normal form built from atomic formulas, $\wedge$, and $\exists$ only. A relation is *Primitive Positive*

[13] Updating the registers can be thought of re-interpreting a fixed set (a vector) of k constants. However, using singleton-set relations, that represent registers, is more convenient in Datalog-like rules.

(PP) if it is definable by a CQ:

$$\forall x_1 \ldots \forall x_k \ \big(R(x_1, \ldots, x_k) \leftrightarrow \underbrace{\exists z_1 \ldots \exists z_m \ (B_1 \wedge \cdots \wedge B_m)}_{\Phi(x_1,\ldots,x_k)}\big),$$

and each atomic formula B_i has object variables from $x_1, \ldots, x_k, z_1, \ldots, z_m$. We will use $\bar{x}$, $\bar{u}$, etc., to denote tuples of variables, and use the broadly used Datalog notation for the sentence above:

$$R(\bar{x}) \leftarrow \underbrace{B_1(\bar{u}_1), \ldots, B_n(\bar{u}_n)}_{\Phi_{\text{body}}}, \tag{9}$$

where the part on the right of $\leftarrow$ is called the *body*, and the part to the left of $\leftarrow$ the *head* of the rule or the *answer*. Notice that an answer symbol is always different from the symbols in the body (9), since, by definition, CQs are not recursive. We say that $R(x_1, \ldots, x_k)$ is *monadic* if $k = 1$, and apply this term to the corresponding PP-relations as well.

Example 1. Relation **Path of Length Two** is PP-definable, but not monadic: $P(x_1, x_2) \leftarrow E(x_1, z), E(z, x_2)$.

Example 2. Relation **At Distance Two from** X is monadic and PP-definable: $D(x_2) \leftarrow X(x_1), E(x_1, z), E(z, x_2)$.

From now on, we assume that all queries are conjunctive, and their answers are monadic, as in Example 2. We now introduce a *modification of CQs* we use as the secondary logic, the bottom level of our algebra. Relational vocabulary $\tau := \tau_{\text{EDB}} \uplus \tau_{\text{reg}}$, consists of two disjoint sets. The symbols in τ_{reg} are monadic, and reg abbreviates "registers". Their interpretations change during a computation formalized by an algebraic term. The arities of the symbols of τ_{EDB} are arbitrary. Their interpretations are given by the input structure and never change.

SM-PP Atomic Modules Let $R'_i(x) \leftarrow \Phi_{\text{body}}$ be a unary CQ where $R_i \in \tau_{\text{reg}}$ and Φ_{body} is in τ. The reason for the renaming of R_i by R'_i is to specify R_i's value in a successor state, without introducing recursion over R_i, which is not allowed in CQs. *Singleton-set-Monadic Primitive Positive (*SM-PP*) relation* is a singleton-set relation R' implicitly definable by:

$$\forall x \ \big((\Phi_{\text{body}}(x) \rightarrow R'_i(x)) \ \wedge \ \forall x \forall y (R'_i(x) \wedge R'_i(y) \rightarrow x = y)\big). \tag{10}$$

We use a rule-based notation for (10):

$$R_i(x) \leftharpoondown \Phi_{\text{body}}. \tag{11}$$

Notation "$\leftharpoondown$" in (11), unlike Datalog's "$\leftarrow$" in (9), is used to emphasize that only one domain element is put into the relation in the head of the rule.

Example 3. Suppose we want to put just one (arbitrary) element in extension of D denoting "At Distance Two" from Example 2. In the rule-based syntax (11):

$$D(x_2) \leftharpoondown X(x_1), E(x_1, z), E(z, x_2).$$

The defined relation is SM-PP. Since only one domain element, out of those at distance two from X, is put into D, there could be multiple outcomes, up to the size of the input domain.

Example 4. A non-example of SM-PP relation is Path of Length Two (Example 1). The relation P on the output of the atomic module is neither monadic, nor singleton-set.

An *SM-PP module* is a set of rules of the form (11):

$$\varepsilon a \ := \ \left\{ \begin{array}{l} R_1(x) \leftharpoondown \Phi^1_{\text{body}} \\ \cdots \\ R_k(x) \leftharpoondown \Phi^k_{\text{body}} \end{array} \right\}, \tag{12}$$

where εa is a symbolic notation for the module, where $a \in \mathcal{M}$, and ε is a free function variable ranging over Choice functions.
In the rest of this paper, the modules are SM-PP, that is, they define SM-PP relations. We use modules to specify atomic transitions or *actions*. Each module may update the interpretations of several registers simultaneously. Thus, we allow limited parallel computations. The parallelism is essential – this way we avoid using relations of arity greater than one, such as in the Same Generation example, explained in Section 6.

Inertia We now formulate the Law of Inertia for SM-PP modules. Consider a module specification in the form (12). We say that a value b in a successor state $\mathfrak{B}$ of a register R_i (where b is a domain element) is *forced by transition* $(\mathfrak{A}, \mathfrak{B}) \in \mathbf{Tr}[\![a]\!]$ if, whenever $\mathfrak{A} \models \Phi^i[b/x]$, we have that $b \in R_i^{\mathfrak{B}}$. Here, $\Phi^i[b/x]$ is a formula in the secondary logic (here, a unary conjunctive query with free variable x). These queries must be applied to $\mathfrak{A}$ simultaneously, in order for the update $(\mathfrak{A}, \mathfrak{B}) \in \mathbf{Tr}[\![a]\!]$ to happen. The *transitions specified by* a is a binary relation $\mathbf{Tr}[\![a]\!] \subseteq \mathbf{U} \times \mathbf{U}$ such that, for all transitions $(\mathfrak{A}, \mathfrak{B}) \in \mathbf{Tr}[\![a]\!]$, the values of all registers in $\mathfrak{B}$, forced by the transition, are as specified by the rules (12), and all other relations remain unchanged in the transition from $\mathfrak{A}$ to $\mathfrak{B}$.

Definition 1. *By $\mathbb{L}$ we denote the algebra with a two-level syntax that consists of the algebra defined in Section 2, with SM-PP modules of the form (12).*

Machine Model We can think of evaluations of algebraic expressions as computations of Register machines starting from input $\mathfrak{A}$. The machines are reminiscent those of Shepherdson and Sturgis [40]. Importantly, we are interested in *isomorphism-invariant* computations, i.e., those that do not distinguish between isomorphic input structures. We have: (1) monadic "registers" – predicates used during the computation, each containing only one domain element at a time; (2) the "real" inputs, e.g., the edge relation $E(x, y)$ of an input graph, are of any arity; (3) atomic transitions correspond to conditional assignments with a non-deterministic outcome; (4) in each atomic step, only the registers of the previous state or the input structure are accessible; (5) a concrete Choice function, depending on the history, chooses one of the possible outputs; (6) computations are controlled by algebraic terms that, intuitively, represent programs. In Section 4, we saw that the main programming constructs are definable. In Section 6, we give examples of "programming" in this model.

6 Examples

We now give some cardinality and reachability examples, and an of an NP-complete problem.[14] We assume that the input structure $\mathfrak{A}$ is of combined vocabulary $\tau := \tau_{\text{EDB}} \uplus \tau_{\text{reg}}$, where $\mathfrak{A}|_{\tau_{\text{EDB}}}$ is an input to a computational problem, e.g., a graph, and the "registers" in $\mathfrak{A}|_{\tau_{\text{reg}}}$ are interpreted by a "blank" symbol. We use α with subscripts to denote algebraic terms, and add τ_{EDB} symbols, e.g., P and Q in $\alpha_{\text{eq\_size}}(P, Q)$, to emphasize what is given on the input.

Problem: **Size Four** α_4
Given: A structure $\mathfrak{A}$ with a vocabulary symbol *adom* denoting its active domain. Question: Is $|adom^{\mathfrak{A}}|$ equal to 4?

$$\varepsilon GuessP := \left\{ P(x) \leftarrowtail adom(x) \right\}.$$

Here, we put an arbitrary element of the active domain into P. We specify guessing a new element by checking that the interpretation of P now has never appeared in the trace of the program before: $GuessNewP := GuessP\,;\ {}_{\text{BG}}(P_{now} \neq P)$. The problem Size Four is now specified as: $\alpha_4 := GuessNewP^4\,;\ {\sim}GuessNewP$, where the power four means that we execute the guessing procedure four times sequentially. The answer to the question $\mathfrak{A} \models |\alpha_4\rangle\top$, is "yes", i.e., it is possible to find a concrete Choice function to semantically instantiate ε, if and only if the input domain is of size four. Obviously, such a program can be written for any natural number.

Problem: **EVEN** α_E
Given: A structure $\mathfrak{A}$ with a vocabulary symbol *adom* denoting its active domain. Question: Is $|adom^{\mathfrak{A}}|$ even?

EVEN is PTIME computable, but is not expressible in Datalog, or any fixed point logic, unless a linear order on the domain elements is given. It is also known that Monadic Second-Order (MSO) logic over the empty vocabulary cannot express EVEN. In our logic, we construct a 2-coloured path in the transition system by guessing new domain elements one-by-one, and using E and O as labels.[15] We make sure that the elements never repeat. We define three atomic modules:

$$\begin{aligned} \varepsilon GuessP &:= \left\{ P(x) \leftarrowtail adom(x) \right\}, \\ \varepsilon CopyPO &:= \left\{ O(x) \leftarrowtail P(x) \right\}, \\ \varepsilon CopyPE &:= \left\{ E(x) \leftarrowtail P(x) \right\}. \end{aligned}$$

$$\begin{aligned} GuessNewO &:= \big(GuessP\,;\ {}_{\text{BG}}(P \neq E)\ {}_{\text{BG}}(P \neq O)\big)\ ;\ CopyPO, \\ GuessNewE &:= \big(GuessP\,;\ {}_{\text{BG}}(P \neq E)\ {}_{\text{BG}}(P \neq O)\big)\ ;\ CopyPE. \end{aligned}$$

The problem EVEN is now formalized as: $\alpha_E := (GuessNewO\,;GuessNewE)^{\uparrow}\,;\ {\sim}GuessNewO$. The program is successfully executed if each chosen element

[14] We have also developed other examples, including those with mixed propagations, connected to the construction by Cai et al. [10], and not expressible in infinitary counting logic, as shown by Atserias, Bulatov and Dawar [6].

[15] It is possible to use fewer register symbols, but we are not trying to be concise here.

is different from any elements selected so far in the current information flow, and if E and O are guessed in alternation. Given a structure $\mathfrak{A}$ over an empty vocabulary, the result of the query $\mathfrak{A} \models |\alpha_E\rangle\top$ is "yes" whenever there is a successful execution of α_E, that is, the size of the input domain is even.

Problem: **s-t-Connectivity** $\alpha(E, S, T)$
Given: Binary relation E, two constants s and t, as singleton-set relations S and T. Question: Is t reachable from s by following the edges?

We use the definable constructs of imperative programming:

$$\alpha(E,S,T) := \quad M_{base\_case}\,;\mathbf{repeat}\,\big(M_{ind\_case};\ \textsc{bg}(\mathit{Reach}' \neq \mathit{Reach})\big)^{-};\,\mathit{Copy}\ \ \mathbf{until}\ \mathit{Reach} = T.$$

Here, we use a unary relational symbol (a register) *Reach*. Initially, the corresponding relation contains the same node as S. The execution is terminated when *Reach* equals T. Register Reach' is used as a temporary storage. To avoid guessing the same element multiple times, we use the $\textsc{bg}$ construct. The atomic modules used in this program are:

$$\begin{aligned} \varepsilon M_{base\_case} &:= \ \big\{\, \mathit{Reach}(x) \leftharpoondown S(x) \,\big\}\,, \\ \varepsilon M_{ind\_case} &:= \big\{\, \mathit{Reach}'(y) \leftharpoondown \mathit{Reach}(x), E(x,y) \,\big\}\,, \\ \varepsilon \mathit{Copy} &:= \quad \big\{\ \mathit{Reach}(x) \leftharpoondown \mathit{Reach}'(x) \,\big\}\,. \end{aligned}$$

Here, module M_{ind_case} is the only non-deterministic module. The other two modules are deterministic (i.e., the corresponding binary relation is a partial function). Given structure $\mathfrak{A}$ over a vocabulary that matches the input (EDB) predicate symbols E, S and T, including matching the arities, by checking $\mathfrak{A} \models |\alpha\rangle\top$, we verify that there is a successful execution of α. That is, t is reachable from s by following the edges of the input graph.

Problem: **Same Generation** $\alpha_{\mathrm{SG}}(E, \mathit{Root}, A, B)$
Given: Tree – edge relation: E; root: *Root*; two nodes represented by unary singleton-set relations: A and B. Question: Do A and B belong to the same generation in the tree?

Note that, since we do not allow binary "register" relations (binary EDB relations are allowed), we need to capture the notion of being in the same generation through coexistence in the same structure.

$$\varepsilon M_{base\_case} := \big\{\, \mathit{Reach}_A(x) \leftharpoondown A(x),\ \mathit{Reach}_B(x) \leftharpoondown B(x) \,\big\}\,.$$

Simultaneous propagation starting from the two nodes:

$$\varepsilon M_{ind\_case} := \left\{ \begin{array}{l} \mathit{Reach}'_A(x) \leftharpoondown \mathit{Reach}_A(y), E(x,y), \\ \mathit{Reach}'_B(v) \leftharpoondown \mathit{Reach}_B(w), E(v,w) \end{array} \right\}.$$

This atomic module specifies that, if elements y and w, stored in the interpretations of Reach_A and Reach_B respectively, coexisted in the previous state, then

x and v will coexist in the successor state. We copy the reached elements into "buffer" registers:

$$\varepsilon Copy := \left\{ \begin{array}{l} Reach_A(x) \leftharpoondown Reach'_A(x), \\ Reach_B(x) \leftharpoondown Reach'_B(x) \end{array} \right\}.$$

The resulting interpretation of $Reach_A$ and $Reach_B$ coexist in one structure, which is a state in a transition system. The algebraic expression, using the definable imperative constructs, is:

$$\begin{array}{l} \alpha_{\mathrm{SG}}(E, Root, A, B) := M_{base\_case}; \mathbf{repeat} M_{ind\_case}\,;\, Copy; \\ \quad \mathbf{until}\ (Reach_A = Root\,;\, Reach_B = Root). \end{array}$$

While this expression looks like an imperative program, it really is a *constraint* on all possible Choice functions, each specifying a particular sequence of choices. The answer to the question $\mathfrak{A} \models |\alpha_{\mathrm{SG}}\rangle \top$ is "yes" if and only if A and B belong to the same generation in the tree.

Finally, we give an example of an NP-complete problem by a MSc student Yi Chen. While other formalizations are possible, this one especially elegant.

Problem: **3-Colourability** $\alpha_{3\mathrm{Col}}(E)$ Given: graph $G = (V, E)$, which is a relational structure. Question: Is G 3-Colourable?

$$\begin{array}{l} \alpha_{pickNew} := \{\, W(x) \leftharpoondown \}\,;\ \textsc{bg}(W \neq V)\,;\, \{\, V(x) \leftharpoondown W(x)\,\} \\ \alpha_{color} := \alpha_R \sqcup \alpha_B \sqcup \alpha_G \\ \alpha_R := \ \sim(\{\, W(y) \leftharpoondown V(x) \wedge E(x,y)\,\}\,;\ \sim \textsc{bg}(W \neq R))\,;\, \{\, R(x) \leftharpoondown V(x)\,\} \\ \alpha_B := \ \sim(\{\, W(y) \leftharpoondown V(x) \wedge E(x,y)\,\}\,;\ \sim \textsc{bg}(W \neq B))\,;\, \{\, B(x) \leftharpoondown V(x)\,\} \\ \alpha_G := \ \sim(\{\, W(y) \leftharpoondown V(x) \wedge E(x,y)\,\}\,;\ \sim \textsc{bg}(W \neq G))\,;\, \{\, G(x) \leftharpoondown V(x)\,\} \\ \alpha := (\alpha_{pickNew}\,;\, \alpha_{color})^{\uparrow}\,;\, \alpha_{pickNew} \end{array}$$

7 Complexity of Query Evaluation

In this section, we discuss the complexity of query evaluation in logic $\mathbb{L}$ (Definition 1), specifically focusing on its fragment that captures precisely the complexity class NP. *Data complexity* refers to the case where the formula is fixed, and input structures $\mathfrak{A}$ vary [46]. Here, we assume that the description of the transition system, specified in the secondary logic SM-PP (10), is a part of the formula in the main task (7), and, therefore, is fixed. Proofs are in the Appendix.

The Length of a Choice Function By the *length* of a Choice function we mean the maximal length of an "extension" string $v \backslash w$, for $v \sqsubseteq w$ i.e., a delta (see Section 3) of a term: $length(\bar{h}(t)) := \max\{|v \backslash w| \mid (v, w) \in \bar{h}(t)\}$. Intuitively, it corresponds to the height of the "unwinding" tree, starting from v, of a term, as in Figure 1. We say that a Choice function h is *polynomial* if $length(\bar{h}(t)) \in O(n^k)$, where n is the size of the domain of $\mathfrak{A} \in \mathbf{U}$, and k is some constant. Since Choice functions are certificates, their length has to be restricted.

Upper Bound To analyze the data complexity for all $\mathbb{L}$-terms, under *polynomial* Choice functions, we use the algorithm based on the structural operational

semantics presented in Section 9. The complexity depends on the nesting of the implicit quantifiers on Choice functions (cf. 6), i.e., how exactly $\sim$ is applied in the term, including as part of the evaluation of Maximum Iterate. Since the implicit (existential and universal) quantifiers can alternate, a problem at any level of the Polynomial Time Hierarchy can be expressed. Thus, the upper bound is PSPACE.

Capturing NP For a *restricted* language, where, in the term inside the modality of the main task, $\sim$ is applied to atomic modules only, including in Maximum Iterate, the data complexity of the main task is in NP. Intuitively, this is because, in that case, the size of the certificate (Choice function) we guess is polynomial, and it can verified in deterministic polynomial time. Moreover, NP is captured precisely.[16]

Theorem 1. *The fragment of logic* $\mathbb{L}$*, where negation applies to atomic modules only, and Choice functions are polynomial, captures precisely NP with respect to its data complexity.*

8 Discussion and Conclusion

We have defined an algebra of partial functions on strings of relational structures. The algebra has a two-level syntax, where propagations on the bottom level are separated from algebraic control. The operations of the top level are obtained by taming (dynamic versions of) classical connectives and a fixed-point construct, that is, by making them *function-preserving*. The algebra allows for complex nested tests, and has an iterator construct. A particular example of a logic of the bottom level is a singleton-set restriction of Monadic Conjuctive Queries that, intuitively, represent non-deterministic conditional assignments. The algebra has an equivalent form of a Dynamic logic. Typical programming constructs, such as while loops and if-then-else, are definable. We have defined a machine model, and considered a restricted fragment where the length of the Choice functions is limited to a polynomial in the size of the input structure. Since the logic can implicitly mimic quantification over certificates, it can express problems at any level of the Polynomial Time Hierarchy. With further restriction, where negations are applied to atomic modules only, the logic captures precisely the complexity class NP. The logic expresses *both* reachability and counting type of examples, *without* a special counting construct. This is known to be challenging for logics with restricted expressiveness [30].

As the next step, we want to understand *under what general conditions on the terms* t *of the logic* $\mathbb{L}$, evaluating the main query $\mathfrak{A} \models |t\rangle\top$ can be done by simply following one (arbitrary) sequence of atomic choices. When such an evaluation is possible, the query is choice-invariant. Work is well under way on a proof system (and a quasi-equational theory) for the algebra, with the goal of performing automated reasoning. Current work also includes an application to deontic reasoning.

[16] This theorem is proven, as Corollary 1, in the Appendix.

Acknowledgements The author is grateful to Leonid Libkin, Ramyaa, Brett McLean and Balder ten Cate for useful discussions. The author's research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). Part of the research presented in this paper was carried out while the author participated in the program on Propositional Satisfiability and Beyond of the Simons Institute for the Theory of Computing during the spring of 2021, and its extended reunion.

References

1. Heba Aamer. *Logical Analysis of Input and Output Sensitivity in the Logic of Information Flows*. PhD thesis, Hasselt University, Belgium, 2023.
2. Heba Aamer, Bart Bogaerts, Dimitri Surinx, Eugenia Ternovska, and Jan Van den Bussche. Inputs, outputs, and composition in the logic of information flows. *ACM Transactions on Computational Logic (TOCL)*, 24:1–44.
3. Heba Aamer, Bart Bogaerts, Dimitri Surinx, Eugenia Ternovska, and Jan Van den Bussche. Executable first-order queries in the logic of information flows. In *Proceedings 23rd International Conference on Database Theory*, volume 155 of *Leibniz International Proceedings in Informatics*, pages 4:1–4:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.
4. Heba Aamer, Bart Bogaerts, Dimitri Surinx, Eugenia Ternovska, and Jan Van den Bussche. Inputs, outputs, and composition in the logic of information flows. In *KR'2020*, 2020.
5. Vikraman Arvind and Somenath Biswas. Expressibility of first order logic with a nondeterministic inductive operator. In *STACS*, volume 247 of *Lecture Notes in Computer Science*, pages 323–335. Springer, 1987.
6. Albert Atserias, Andrei A. Bulatov, and Anuj Dawar. Affine systems of equations and counting infinitary logic. *Theor. Comput. Sci.*, 410(18):1666–1683, 2009.
7. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
8. Andreas Blass and Yuri Gurevich. The logic of choice. *J. Symb. Log.*, 65(3):1264–1310, 2000.
9. Andreas Blass, Yuri Gurevich, and Saharon Shelah. Choiceless polynomial time. *Ann. Pure Appl. Logic*, 100(1-3):141–187, 1999.
10. Jin-yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identifications. *Combinatorica*, 12(4):389–410, 1992.
11. Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860, 2013.
12. Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.
13. Ronald Fagin. Generalized first-order spectra and polynomial-time recognizable sets. *Complexity of computation, SIAM-AMC proceedings*, 7:43–73, 1974.
14. Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.
15. George H. L. Fletcher, Marc Gyssens, Dirk Leinders, Dimitri Surinx, Jan Van den Bussche, Dirk Van Gucht, Stijn Vansummeren, and Yuqing Wu. Relative expressive power of navigational querying on graphs. *Inf. Sci.*, 298:390–406, 2015.

16. Françoise Gire and H. Khanh Hoang. An extension of fixpoint logic with a symmetry-based choice construct. *Inf. Comput.*, 144(1):40–65, 1998.
17. Erich Grädel. Capturing complexity classes by fragments of second order logic. In *Computational Complexity Conference*, pages 341–352. IEEE Computer Society, 1991.
18. Erich Grädel. Why are modal logics so robustly decidable? In *Current Trends in Theoretical Computer Science*, pages 393–408. 2001.
19. Yuri Gurevich. Logic and the challenge of computer science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.
20. David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic (Foundations of Computing)*. MIT Press, 2000.
21. Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
22. David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 2. Springer Verlag, 1939.
23. Marco Hollenberg and Albert Visser. Dynamic negation, the one and only. *J. Log. Lang. Inf.*, 8(2):137–141, 1999.
24. Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.
25. Marcel Jackson and Timothy Stokes. Modal restriction semigroups: towards an algebra of functions. *IJAC*, 21(7):1053–1095, 2011.
26. Bjarni Jónsson and Alfred Tarski. Representation problems for relation algebras. *Bull. Amer. Math. Soc.*, 74:127–162, 1952.
27. Tobias Kappé. Completeness and the finite model property for kleene algebra, reconsidered. In *RAMiCS*, volume 13896 of *Lecture Notes in Computer Science*, pages 158–175. Springer, 2023.
28. Dexter Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.
29. Stepan L. Kuznetsov. On the complexity of reasoning in kleene algebra with commutativity conditions. In *ICTAC*, volume 14446 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2023.
30. Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.
31. John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.
32. Brett McLean. Complete representation by partial functions for composition, intersection and anti-domain. *J. Log. Comput.*, 27(4):1143–1156, 2017.
33. D. G. Mitchell and E. Ternovska. A framework for representing and solving NP search problems. In *Proc. AAAI'05*, pages 430–435, 2005.
34. Martin Otto. Epsilon-logic is more expressive than first-order logic over finite structures. *J. Symb. Log.*, 65(4):1749–1757, 2000.
35. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981. Also published in: Journal of Logic and Algebraic Programming, 60-61:17-140, 2004.
36. Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science, Houston, Texas, USA, 25-27 October 1976*, pages 109–121, 1976.

37. Vaughan R. Pratt. Origins of the calculus of binary relations. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*, pages 248–254, 1992.
38. Todd Schmid, Tobias Kappé, and Alexandra Silva. A complete inference system for skip-free guarded kleene algebra with tests. In *ESOP*, volume 13990 of *Lecture Notes in Computer Science*, pages 309–336. Springer, 2023.
39. Luc Segoufin and Balder ten Cate. Unary negation. *Log. Methods Comput. Sci.*, 9(3), 2013.
40. John C. Shepherdson and Howard E. Sturgis. Computability of recursive functions. *J. ACM*, 10(2):217–255, 1963.
41. Sergei Soloviev. Studies of Hilbert's epsilon-operator in the USSR. *FLAP*, 4(2), 2017.
42. Dimitri Surinx, Jan Van den Bussche, and Dirk Van Gucht. The primitivity of operators in the algebra of binary relations under conjunctions of containments. In *LICS '17*, 2017.
43. E. Ternovska. An algebra of modular systems: Static and dynamic perspectives. In *Proceedings of the 12th International Symposium on Frontiers of Combining Systems (FroCoS)*, September 2019.
44. Eugenia Ternovska. A logic of information flows (two page abstract). In *Proc. KR2020*, 2020.
45. Moshe Vardi. On the complexity of bounded-variable queries. In *PODS*, pages 266–276. ACM Press, 1995.
46. Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146. ACM, 1982.
47. Moshe Y. Vardi. Why is modal logic so robustly decidable? In *Descriptive Complexity and Finite Models, Proceedings of a DIMACS Workshop 1996, Princeton, New Jersey, USA, January 14-17, 1996*, pages 149–184, 1996.
48. Claus-Peter Wirth. The explicit definition of quantifiers via hilbert's epsilon is confluent and terminating. *Journal of Applied Logics – IFCoLog Journal of Logics and their Applications, Special Issue on Hilbert's epsilon and tau in Logic, Informatics and Linguistics*, 4(2), 2017.
49. Thomas Zeume and Frederik Harwath. Order-invariance of two-variable logic is decidable. In *LICS*, pages 807–816. ACM, 2016.

9 Appendix: Structural Operational Semantics

Our goal is to develop an algorithm that, given a Choice function h, finds an answer to the main task $\mathfrak{A} \models |\alpha\rangle$. We represent the algorithm as a set of rules in the style of Plotkin's Structural Operational Semantics [35]. The transitions describe "single steps" of the computation, as in the computational semantics [21].

Identity (Diagonal) id:

$$\frac{true}{(\mathrm{id}, v) \longrightarrow (\mathrm{id}, v)}.$$

Atomic Modules εa:

$$\frac{true}{(\varepsilon a, v \cdot \mathfrak{A}) \longrightarrow (\mathrm{id}, v \cdot \mathfrak{A} \cdot \mathfrak{B})} \text{ if } (v \cdot \mathfrak{A} \mapsto v \cdot \mathfrak{A} \cdot \mathfrak{B}) \in h(\varepsilon a).$$

Sequential Composition $\alpha\,;\beta$:

$$\frac{(\alpha, v) \longrightarrow (\alpha', v')}{(\alpha\,;\beta, v) \longrightarrow (\alpha'\,;\beta, v')}, \qquad \frac{(\beta, v) \longrightarrow (\beta', v')}{(\mathrm{id}\,;\beta, v) \longrightarrow (\mathrm{id}\,;\beta', v')}.$$

Preferential Union $\alpha \sqcup \beta$:

$$\frac{(\alpha, v) \longrightarrow (\alpha', v')}{(\alpha \sqcup \beta, v) \longrightarrow (\alpha', v')}.$$

That is, $\alpha \sqcup \beta$ evolves according to the instructions of α, if α can successfully evolve to α'.

$$\frac{(\beta, v) \longrightarrow (\beta', v') \text{ and } (\sim\alpha, v) \longrightarrow (\sim\alpha, v)}{(\alpha \sqcup \beta, v) \longrightarrow (\beta', v')}.$$

The rule says that $\alpha \sqcup \beta$ evolves according to the instructions of β, if β can successfully evolve, while α cannot.

Unary Negation (Anti-Domain) $\sim\alpha$: There are no one-step derivation rules for $\sim\alpha$. Instead, we try, step-by-step, to derive α, and, if not derivable, make the step.

$\dfrac{true}{(\sim\alpha, v) \longrightarrow (\mathrm{id}, v)}$ if there is no Choice function h' such that derivation of α in v succeeds.

Equality Check $(P = Q)$:

$$\frac{true}{((P = Q), v \cdot \mathfrak{B}) \longrightarrow (\mathrm{id}, v \cdot \mathfrak{B})} \text{ if } P^{\mathfrak{B}} = Q^{\mathfrak{B}}.$$

Back Globally $\textsc{bg}(P \neq Q)$:

$$\frac{true}{(\textsc{bg}(P \neq Q), v \cdot \mathfrak{B}) \longrightarrow (\mathrm{id}, v \cdot \mathfrak{B})} \text{ if for all } i,\ P^{v(i)} \neq Q^{\mathfrak{B}}.$$

Here, $v(i)$ is the i'th letter of v, $\mathit{first}(v) \leq i \leq \mathit{last}(v)$.

Maximum Iterate $\alpha^{\uparrow}$:

$$\frac{(\alpha, v) \longrightarrow (\alpha', v')}{(\alpha^{\uparrow}, v) \longrightarrow (\alpha'\,;\alpha^{\uparrow}, v')}, \qquad \frac{true}{(\alpha^{\uparrow}, v) \longrightarrow (\mathrm{id}, v)} \text{ if } \sim\alpha \text{ succeeds in } v.$$

Thus, $\alpha^{\uparrow}$ evolves according to α, if α can evolve successfully, or if $\sim\alpha$ succeeds in v.

The execution of the algorithm consists of "unwinding" the term α, starting with an input structure $\mathfrak{A}$. The derivation process is deterministic: whenever there are two rules for a connective, only one of them is applicable. The goal of evaluation is to apply the rules of the structural operational semantics starting from $(\alpha, \mathfrak{A})$ and then tracing the evolution of α to the "empty" program id step-by-step, by completing the branch upwards that justifies the step. In that case, we say that the derivation *succeeds*. Otherwise, we say that it *fails*.

Proposition 1. *The evaluation algorithm based on the structural operational semantics, that finds an answer to the main task* $\mathfrak{A} \models |\alpha\rangle\top$*, is correct with respect to the semantics of* $\mathbb{L}$*.*

Proof. (outline) The correctness of the algorithm follows by induction on the structure of the algebraic expression, since the rules simply implement the semantics given earlier.

10 Appendix: Complexity: Proofs

Theorem 2. *The data complexity of checking* $\mathfrak{A} \models |\alpha\rangle\top$ *for the restriction of logic* $\mathbb{L}$*, where negation applies to atomic modules only and Choice functions are polynomial, is in* NP*.*

Proof. We guess a certificate, which is a trace of the computation of α, which is of a polynomial length in the size of $\mathfrak{A}$, to instantiate the free function variable ε. With such an instantiation of a *fixed* Choice function h, term α becomes deterministic. By our assumption, h of polynomial length.

We argue, by induction on the structure of algebraic terms, that all operations, including negation, can be evaluated in polynomial time. The base case, SM-PP atomic modules (essentially, conjunctive queries) can be checked, given this certificate, in P-time. For the inductive case, we argue, using Structural Operational Semantics from Section 9, that each step of applying a rule can be done in P-time.

Thus, we return "yes" in polynomial time if there is a witness that proves that the answer to $\mathfrak{A} \models |\alpha\rangle\top$ is "yes"; or "no" in polynomial time otherwise.

Theorem 3. *For every NP-recognizable class* $\mathcal{K}$ *of structures, there is a sentence of logic* $\mathbb{L}$ *(where negation applies to atomic modules only) whose models are exactly* $\mathcal{K}$*.*

Proof. We focus on the query $\mathfrak{A} \models |\alpha_{\mathrm{TM}}\rangle\top$ from Section 4 and outline such a construction. The main idea is that a linear order on the domain elements of $\mathfrak{A}$ is *induced* by a path in a transition system that corresponds to a guessed Choice function h. In this path, we *guess* new elements one by one, as in the examples. The linear order corresponds to an order on the tape of a Turing machine. After such an order is guessed, a deterministic computation, following the path, proceeds for that specific order. We assume, without loss of generality that the deterministic machine runs for n^k steps, where n is the size of the domain. The program is of the form:

$$\alpha_{\mathrm{TM}} := \text{ORDER} \ ; \ \text{START} \ ; \ \textbf{repeat} \ \text{STEP} \ \textbf{until} \ \text{END}.$$

Procedure ORDER: Guessing an order is perhaps the most important part of our construction. We use a secondary numeric domain with a linear ordering, and guess elements one-by-one, using a concrete Choice function h. We associate an

element of the primary domain with an element of the secondary one, using coexistence in the same structure. Each Choice function corresponds to a possible linear ordering.

Procedure START: This procedure creates an encoding of the input τ_{EDB}-structure $\mathfrak{A}$ (say, a graph) in a sequence of structures in the transition system, to mimic an encoding $enc(\mathfrak{A})$ on a tape of a Turing machine. We use structures to represent cells of the tape of the Turing machine (one τ-structure = one cell). The procedure follows a specific path, and thus a specific order generated by the procedure START. Subprocedure $Encode(vocab(\mathfrak{A}), \ldots, S_\sigma, \ldots, \bar{P}, \ldots)$ operates as follows. In every state (= cell), it keeps the input structure $\mathfrak{A}$ itself, and adds the part of the encoding of $\mathfrak{A}$ that belongs to that cell. The interpretations of $\bar{P}$ over the secondary domain of labels provide cell positions on the simulated input tape. Each particular encoding is done for a specific induced order on domain elements, in the path that is being followed.

In addition to producing an encoding, the procedure START sets the state of the Turing machine to be the initial state Q_0. It also sets initial values for the variables used to encode the instructions of the Turing machine.

Expression START is similar to the first-order formula $\beta_\sigma(\bar{a})$ used by Grädel in his proof of capturing P-time using SO-HORN logic on ordered structures [17]. The main difference is that instead of tuples of domain elements $\bar{a}$ used to refer to the addresses of the cells on a tape, we use tuples $\bar{P}$, also of length k. Grädel's formula $\beta_\sigma(\bar{a})$ for encoding input structures has the following property: $(\mathfrak{A}, <) \models \beta_\sigma(\bar{a}) \quad \Leftrightarrow \quad$ the $\bar{a}$-th symbol of $enc(\mathfrak{A})$ is σ. Here, we have:

$$\begin{array}{l} \mathfrak{A} \models Encode(\ldots, S_\sigma, \ldots, P_1(a_1), \ldots, P_k(a_k), \ldots)(h/\varepsilon) \\ \Leftrightarrow \quad \text{the } P_1(a_1), \ldots, P_k(a_k)\text{-th symbol of } enc(\mathfrak{A}) \text{ is } \sigma, \end{array}$$

where $\bar{a}$ is a tuple of elements of the secondary domain, h is a Choice function that guesses a linear order on the input domain through an order on structures (states in the transition system), starting in the input structure $\mathfrak{A}$. That specific generated order is used in the encoding of the input structure. Another path produces a different order, and constructs an encoding of the input structure for that order.

Procedure STEP: This procedure encodes the instructions of the deterministic Turing machine. SM-PP modules are well-suited for this purpose. Instead of time and tape positions as arguments of binary predicates as in Fagin's [13] and Grädel's [17] proofs, we use coexistence in the same structure with k-tuples of domain elements, as well as lexicographic successor and predecessor on such tuples. Polynomial time of the computation is guaranteed because time, in the repeat-until part, is tracked with k-tuples of domain elements.

Procedure END: It checks if the accepting state of the Turing machine is reached.

We have that, for any P-time Turing machine, we can construct term α_{TM} in the logic $\mathbb{L}$ such that the answer to $\mathfrak{A} \models |\alpha_{\mathrm{TM}}\rangle \top$ is "yes" if and only if the Turing machine accepts an encoding of $\mathfrak{A}$ for some specific but arbitrary order of domain elements on its tape.

Combining the theorems, we obtain the following corollary.

Corollary 1. *The fragment of logic $\mathbb{L}$, where negation applies to atomic modules only, and Choice functions are polynomial, captures precisely NP with respect to its data complexity.*

11 Appendix: Related Work in More Detail

Logic of Information Flows The work in the current paper is a continuation of research initiated by the author, who introduced the Logic of Information Flows (LIF) in [43]. The goal of introducing LIF was to understand how information propagates, in order to make such propagations efficient. A version of LIF, [43], was published, initially, in the context of reasoning about modular systems, and was based on classical logic. In subsequent work, with a group of coauthors, we studied the notions of input, output and composition [4], [2] in LIF, and applications to data access in database research [3]. A short description of that work can be found in [44]. A lot of development in the papers [4], [3], and beyond, was done in the excellent PhD work by Heba Aamer [1].

In parallel with the work on [4], [2], the author continued working towards the main goal, on studying how to make information flows efficient. Defining LIF based on classical logic, as was done in the early versions, was clearly not sufficient. One of the main observations of the author, very early in the development of LIF, was that it was necessary to make operations *function-preserving*, and to introduce the notion of *history-dependent Choice function*. The author conducted an extensive analysis, going through numerous versions and combinations of algebraic operations, eventually coming up with a minimal, but sufficiently expressive set of the operations presented here.

Choice Operator Choice occurs in many high-level descriptions of polynomial-time algorithms, e.g., in Gaussian elimination: *choose an element and continue.* A big challenge, in our goal of formalizing non-deterministic computations algebraically, in an algebra of partial functions, was in how to deal with binary relations. Such relations are not necessarily functional. To deal with this challenge, the author invented history-dependent Choice functions. The dependence on the history, to the best of our knowledge, has not been used in formalizing choice before. The first use of a Choice operator ε in logic goes back to Hilbert and Bernays in 1939 [22] for proof-theoretic purposes, without giving a model-theoretic semantics. Early approaches to Choice, in connection to Finite Model Theory, include the work by Arvind and Biswas [5], Gire and Hoang [16], Blass and Gurevich [8] and by Otto [34], among others. Richerby and Dawar [**?**] survey and study logics with various forms of choice. Outside of Descriptive Complexity, Hilbert's ε has been studied extensively by Soviet logicians Mints, Smirnov and Dragalin in 1970's, 80's and 90's, see [41]. The semantics of this operator is still an active research area, see, e.g., [48]. Unlike the earlier approaches, our Choice operator formalizes a *strategy*, i.e., what to do next, given the history, under the given constraint given by a term. An example of using strategies can be seen in

building proofs in a Gentzen-style proof system. There, while selecting an element witnessing an existential quantifier, we ensure that the element is "new", i.e., has not appeared earlier in the proof.

A problem with a set-theoretic Choice operator, for unstructured sets, is that for first-order (FO) logic, and thus for its fixed-point extensions such as FO(FP), choice-invariance is undecidable [8]. Therefore, in using FO, there is a danger of obtaining an undecidable syntax, which violates a basic principle behind a well-defined logic. Choiceless Polynomial Time [9] is an attempt to characterize what can be done *without* introducing choice. But, as a critical step, we *restrict FO connectives*, similarly to Description logics [7], and in strong connection to modal logics, that are robustly decidable [47, 18].

Two-Variable Fragments The step towards binary relations in LIF [43], where we partitioned the variables of atomic symbols into input and outputs, was inspired by our own work on Model Expansion [33], with its before-after-a-computation perspective, and also by bounded-variable fragments of first-order logic. Such fragments have been shown to have good algorithmic properties by Vardi [45]. Two-variable fragments have been offered as an initial explanation of the robust decidability of modal logics [47, 18]. In addition, two-variable fragments have order-invariance [49]. Unfortunately, such fragments of FO are not expressive enough to encode Turing machines. To overcome this obstacle, we lifted the algebra to operate on (functional) *binary relations on strings of relational structures*. Such relations, intuitively, encode state transitions.

Algebras of Binary Relations Such an algebra was first introduced by De Morgan. It has been extensively developed by Peirce and then Schröder. It was abstracted to relation algebra RA by Jónsson and Tarski in [26]. For earlier uses of the operations and a historic perspective please see Pratt's historic and informative overview paper [37]. More recently, relation algebras were studied by Fletcher, Van den Bussche, Surinx and their collaborators in a series of paper, see, e.g. [42, 15]. The algebras of relations consider various subsets of operations on binary relations as primitive, and other as derivable. Our algebra is an algebra of binary relations, similar to Jónsson and Tarski [26], however our relations are on more complex entities – *strings* of relational structures. Moreover, our binary relations are *functional*, which is achieved by making all operations function-preserving, and introducing Choice functions.

Algebras of Functions In another direction, Jackson and Stokes and then McLean [25, 32] study partial functions and their algebraic equational axiomatizations. The work of Jackson and Stokes [25] is particularly relevant, because it introduces some connectives we use. However, they do not study algebras on strings and Choice functions. Also, we had to eliminate intersection, which they, and many other researchers, use. We had to do it because intersection wastes computational power due to confluence, and makes information flows less efficient. We came up with an example where an intersection of (the representations of) two NP-complete problems produce a problem in P-time. We have selected a minimal set of operations, for our purposes. The operations correspond to

dynamic and function-preserving versions of conjunction, disjunction, negation and iteration. However, other operations, such as many of those from McLean [32], can be studied as well.

Restricting Connectives Our algebraic operations are a restriction of first-order connectives, similarly to restrictions of such connectives in Description logics [7]. Classical connectives, such as negation, disjunction, and also the the iterator in the form of the Kleene star (reflexive transitive closure) are incompatible with an algebraic setting of *functions*. We require the connectives to be function-preserving. We traced the origins of the operations we use as follows. The Unary Negation operation is from Hollenberg and Visser [23]. It is also studied, among other operations on functions, by McLean in, e.g., [32] in the form of the Antidomain operation. Restricting negation (full complementation) to its unary counterpart is already known to imply good computational properties [39].[17] However, in our case, *all* connectives and the fixed point construct of first-order with least fixed point, FO(LFP), had to be restricted. The operations of Preferential Union and Maximum Iterate are from Jackson and Stokes [25]. While these algebraic operations are more restrictive than those of Regular Expressions, they define the main constructs of Imperative programming.

Substitution Monoid and Connection to Kleene Algebra First, we explain a connection to a Kleene algebra on a meta-level, and then discuss the differences in the languages. The algebra forms a monoid with respect to term substitution. The proof of this statement is lengthy and is outside of the scope of this paper. However, as for every monoid, there is a well-known and natural connection to a Kleene algebra.

Let M be a monoid with identity element $\mathbf{1}$ and let A be the set of all subsets of M. For two such subsets S and T, let $S + T := S \cup T$, and let $ST := \{st : s \in S \text{ and } t \in T\}$. We define S^* as the submonoid of M generated by S, which can be described as $S^* := \{\mathbf{1}\} \cup S \cup SS \cup SSS \cup ...$ Then A forms a Kleene algebra with 0 being the empty set and 1 being $\{\mathbf{1}\}$.

Comparison to Kleene Algebra at the Level of Algebraic Languages The connection to a Kleene algebra we have just explained exists at the meta-level, when we consider term substitution as the monoid operation. But, what is the connection to Kleene algebra at the level of the algebra itself, i.e., its algebraic operations? One difference is that, unlike the operations of Union ($\cup$) and Iteration (*) of Regular Expressions, the operations of Preferential Union ($\sqcup$) and Maximum Iterate ($^\uparrow$) are *function-preserving*.

Temporal and Dynamic Logics The algebra (2) has an alternative (and equivalent) syntax in the form of a Dynamic logic. Our Dynamic logic is fundamentally different from Propositional Dynamic Logic (PDL) [36, 14] in that, because of the Choice function semantics, it has a linear time (as opposite to

[17] Unary negation is related to the negation of modal logics. It is the modal negation in the modal Dynamic Logic we discuss later. In general, modal logics are known to be robustly decidable [47, 18] due to a combination of properties.

branching time) semantics. This is similar to to Linear Temporal Logic (LTL) and Linear Dynamic logic on finite traces LDL_f [11].[18] But, in addition, branching from nodes is used for complex tests. So, there is some similarity to CTL^*. To the best of our knowledge, the data complexity of temporal and dynamic logics, with respect to an input database, has never been studied.

Algebra vs Logic Our algebra-Dynamic-logic connection is reminiscent that of Kleene algebras with tests (KAT) [28]; but, unlike KAT that allows for simple tests only, our logic allows for arbitrarily complex nested tests, in general.

We believe that ours is the first *algebraic* formalization of a *linear-time* logic, i.e., a logic interpreted over traces of computation (as opposite to branching time logics with branching at every state). A crucial step of this formalization is the use of Choice functions that map strings to strings. We are not aware of any use of such functions in temporal or Dynamic logics.

[18] LDL_f has the same syntax as PDL, but is interpreted over (finite) traces.