Executable First-Order Queries in the Logic of Information Flows

Heba Aamer
Universiteit Hasselt, Belgium

Bart Bogaerts
Vrije Universiteit Brussel, Belgium

Dimitri Surinx
Universiteit Hasselt, Belgium

Eugenia Ternovska
Simon Fraser University, Canada

Jan Van den Bussche
Universiteit Hasselt, Belgium

Abstract

The logic of information flows (LIF) has recently been proposed as a general framework in the field of knowledge representation. In this framework, tasks of a procedural nature can still be modeled in a declarative, logic-based fashion. In this paper, we focus on the task of query processing under limited access patterns, a well-studied problem in the database literature. We show that LIF is well-suited for modeling this task. Toward this goal, we introduce a variant of LIF called "forward" LIF, in a first-order setting. We define FLIF^{io}, a syntactical fragment of forward LIF, and show that it corresponds exactly to the "executable" fragment of first-order logic defined by Nash and Ludäscher. The definition of FLIF^{io} involves a classification of the free variables of an expression into "input" and "output" variables. Our result hinges on inertia and determinacy laws for forward LIF expressions, which are interesting in their own right. These laws are formulated in terms of the input and output variables.

2012 ACM Subject Classification Information systems → Query languages; Computing methodologies → Knowledge representation and reasoning

Keywords and phrases Logic of Information Flows, Limited access pattern, Executable first-order logic

Digital Object Identifier 10.4230/LIPIcs.ICDT.2020.4

Funding Jan Van den Bussche is partially supported by the National Natural Science Foundation of China (61972455). This research was partially supported by FWO project G0D9616N and by the Flanders AI Research Program.

Acknowledgements We thank the ICDT reviewers for pointing out the connection to related work [10, 12].

1 Introduction

An information source is said to have a limited access pattern if it can only be accessed by providing values for a specified subset of the attributes; the source will then respond with tuples giving values for the remaining attributes. A typical example is a restricted telephone directory $D(\text{name}; \text{tel})$ that will show the phone numbers for a given name, but not the other way around.

The querying of information sources with limited access patterns has been quite intensively investigated. The research is motivated by diverse considerations, such as query processing using indices, or information integration on the Web. We refer to the review given by Benedikt et al. [5, Chapter 3.12]. We also cite the work by Calì and collaborators [7, 8, 9].

In this paper, we offer a fresh perspective on querying with limited access patterns, based on the Logic of Information Flows (LIF). This framework has been recently introduced in the field of knowledge representation [18, 19]. The general aim of LIF is to model how information propagates in complex systems. LIF allows machine-independent characterizations of computation; in particular, it allows tasks of a procedural nature to be modeled in a declarative fashion.

In the full setting, LIF is a rich family of logics with higher-order features. The present paper is self-contained, however, and we will work in a lightweight, first-order fragment, which we call *forward* LIF (FLIF). Specifically, we will define a well-behaved, syntactic fragment of FLIF, called *io-disjoint* FLIF. Our main result then is to establish an equivalence between io-disjoint FLIF and *executable first-order logic* (executable FO).

Executable FO [15] is a syntactic fragment of FO in which formulas can be evaluated over information sources in such a way that the limited access patterns are respected. Furthermore, the syntactical restrictions are not very severe and become looser the more free variables are declared as input.

The standard way of formalizing query processing with limited access patterns is by a form of relational algebra programs, called plans [5]. In such plans, database relations can only be accessed by joining them on their input attributes with a relation that is either given as input or has already been computed. Apart from that, plans can use the usual relational algebra operations. Plans can be expressed by executable FO formulas. The strong result [6] is known that every (boolean) FO formula with the semantic property of being *access-determined* can be evaluated by a plan. We will not need this result further on, but it provides a strong justification for working with executable FO formulas.

Our language, FLIF, provides a new, *navigational* perspective on query processing with limited access patterns. In our approach, we formalize the database as a *graph* of variable bindings. Directed edges are labeled with the names of source relations (we are simplifying a bit here). A directed edge $\nu_1 \xrightarrow{R} \nu_2$ indicates that, if we access R with input values given by ν_1, then the output values in ν_2 are a possible result. In a manner very similar to navigational or XPath-like graph query languages [16, 14, 4, 11, 17, 3], FLIF expressions represent paths in the graph.

The io-disjoint fragment of FLIF is defined in terms of *input* and *output* variables that are inferred for expressions. We establish *inertia* and *input-determinacy* properties for FLIF expressions which are instrumental in proving our equivalence between io-disjoint expressions and executable FO, but are also interesting in their own right. Apart from the intuitive navigational nature, another advantage of io-disjoint FLIF is that it is very obvious how expressions in this language can be evaluated by plans. As we will show, the structure of the evaluation plan closely follows the shape of the expression, and all joins can be taken to be natural joins; no attribute renamings are needed.

This paper is further organized as follows. Section 2 recalls the basic setting of executable FO on databases with limited access patterns. Section 3 introduces the language FLIF. Section 4 gives translations between executable FO and io-disjoint FLIF, showing that the evaluation problems for the two languages can be naturally reduced to each other. Section 5 discusses evaluation plans. We conclude in Section 6.

2 Executable FO

Relational database schemas are commonly formalized as finite relational vocabularies, i.e., finite collections of relation names, each name with an associated arity (a natural number). To model limited access patterns, we additionally specify an *input arity* for each name. For example, if R has arity five and input arity two, this means that we can only access R by giving input values, say a_1 and a_2, for the first two arguments; R will then respond with all tuples $(x_1, x_2, x_3, x_4, x_5)$ in R where $x_1 = a_1$ and $x_2 = a_2$.

Thus, formally, we define a *database schema* as a triple $\mathcal{S} = (\mathit{Names}, \mathrm{ar}, \mathrm{iar})$, where *Names* is a set of relation names; ar assigns a natural number $\mathrm{ar}(R)$ to each name R in *Names*, called the arity of R; and iar similarly assigns an input arity to each R, such that $\mathrm{iar}(R) \leq \mathrm{ar}(R)$.

▶ Remark 1. In the literature, a more general notion of schema is often used, allowing, for each relation name, several possible sets of input arguments; each such set is called an access method. In this paper, we stick to the simplest setting where there is only one access method per relation, consisting of the first k arguments, where k is set by the input arity. All subtleties and difficulties already show up in this setting. Nevertheless, our definitions and results can be easily generalized to the setting with multiple access methods per relation. ◀

The notion of database instance remains the standard one. Formally, we fix a countably infinite universe **dom** of atomic data elements, also called *constants*. Now an *instance* D of a schema $\mathcal{S}$ assigns to each relation name R an $\mathrm{ar}(R)$-ary relation $D(R)$ on **dom**. We say that D is *finite* if every relation $D(R)$ is finite. The *active domain* of D, denoted by $\mathrm{adom}(D)$, is the set of all constants appearing in the relations of D.

The syntax and semantics of first-order logic (FO, relational calculus) over $\mathcal{S}$ is well known [2]. In formulas, we allow constants only in equalities of the form $x = c$, where x is a variable and c is a constant. Also, in writing relation atoms, we find it clearer to separate input arguments from output arguments by a semicolon. Thus, we write relation atoms in the form $R(\bar{x}; \bar{y})$, where $\bar{x}$ and $\bar{y}$ are tuples of variables such that the length of $\bar{x}$ is $\mathrm{iar}(R)$ and the length of $\bar{y}$ is $\mathrm{ar}(R) - \mathrm{iar}(R)$. The set of free variables of a formula φ is denoted by $\mathrm{FV}(\varphi)$.

We use the "natural" semantics [2] and let variables in formulas range over the whole of **dom**. Formally, a *valuation* on a set X of variables is a mapping $\nu : X \to \mathbf{dom}$. Given an instance D of $\mathcal{S}$, an FO formula φ over $\mathcal{S}$, and a valuation ν defined on $\mathrm{FV}(\varphi)$, the definition of when φ is satisfied by D and ν, denoted by $D, \nu \models \varphi$, is standard.

A well-known problem with the natural semantics for general FO formulas is that φ may be satisfied by infinitely many valuations on $\mathrm{FV}(\varphi)$, even if D is finite. However, as motivated in the Introduction, we will focus on *executable* formulas, formally defined in this section. These formulas can safely be used under the natural semantics.

The notion of when a formula is executable is defined relative to a set of variables $\mathcal{V}$, which specifies the variables for which input values are already given. We first give a few examples.

▶ **Example 2.**

Let φ be the formula $R(x; y)$. As mentioned above, this notation makes clear that the input arity of R is one. If we provide an input value for x, then the database will give us all y such that $R(x, y)$ holds. Indeed, φ will turn out to be $\{x\}$-executable. Giving a value for the first argument of R is mandatory, so φ is neither $\emptyset$-executable nor $\{y\}$-executable. However, it is certainly allowed to provide input values for both x and y; in

that case we are merely testing if $R(x, y)$ holds for the given pair (x, y). Thus, φ is also $\{x, y\}$-executable. In general, a $\mathcal{V}$-executable formula will also be $\mathcal{V}'$-executable for any $\mathcal{V}' \supseteq \mathcal{V}$.

- Also the formula $\exists y\, R(x; y)$ is $\{x\}$-executable. In contrast, the formula $\exists x\, R(x; y)$ is not, because even if a value for x is given as input, it will be ignored due to the existential quantification. In fact, the latter formula is not $\mathcal{V}$-executable for any $\mathcal{V}$.
- The formula $R(x; y) \wedge S(y; z)$ is $\{x\}$-executable, intuitively because each y returned by the formula $R(x; y)$ can be fed into the formula $S(y; z)$, which is $\{y\}$-executable in itself.
- The formula $R(x; y) \vee S(x; z)$ is not $\{x\}$-executable, because any y returned by $R(x; y)$ would already satisfy the formula, leaving variable z unconstrained. This would lead to an infinite number of satisfying valuations. The formula is neither $\{x, z\}$-executable; if $S(x, z)$ holds for the given values for x and z, then y is left unconstrained. Of course, the formula is $\{x, y, z\}$-executable.
- For a similar reason, $\neg R(x; y)$ is only $\mathcal{V}$-executable for $\mathcal{V}$ containing x and y. ◀

Formally, for any set of variables $\mathcal{V}$, the $\mathcal{V}$-executable formulas are defined as follows.

- An equality $x = y$, for variables x and y, is $\mathcal{V}$-executable if at least one of x and y belongs to $\mathcal{V}$.
- An equality $x = c$, for a variable x and a constant c, is always $\mathcal{V}$-executable.
- A relation atom $R(\bar{x}; \bar{y})$ is $\mathcal{V}$-executable if $X \subseteq \mathcal{V}$, where X is the set of variables from $\bar{x}$.
- A negation $\neg\varphi$ is $\mathcal{V}$-executable if φ is, and moreover $\mathrm{FV}(\varphi) \subseteq \mathcal{V}$.
- A conjunction $\varphi \wedge \psi$ is $\mathcal{V}$-executable if φ is, and moreover ψ is $\mathcal{V} \cup \mathrm{FV}(\varphi)$-executable.
- A disjunction $\varphi \vee \psi$ is $\mathcal{V}$-executable if both φ and ψ are, and moreover $\mathrm{FV}(\varphi) \triangle \mathrm{FV}(\psi) \subseteq \mathcal{V}$. Here, $\triangle$ denotes symmetric difference.
- An existential quantification $\exists x\, \varphi$ is $\mathcal{V}$-executable if φ is $\mathcal{V} - \{x\}$-executable.

Note that universal quantification is not part of the syntax of executable FO.

▶ Remark 3. The naturalness of the above definition may be attested by its reinvention in the context of a different application, namely, inferring bounds on result sizes of FO queries. Indeed, the notion of "controlled" formula that was introduced for this purpose, strikingly conforms to that of executable formula [10]. In the setting of controlled formulas, the input arity k of an n-ary relation R is interpreted as an integrity constraint. An instance D satisfies the constraint if for each k-tuple $\bar{a}$ of constants, the number of $n - k$-tuples $\bar{b}$ such that $\bar{a} \cdot \bar{b} \in D(R)$ stays below a fixed upper bound. ◀

Given an FO formula φ and a finite set of variables $\mathcal{V}$ such that φ is $\mathcal{V}$-executable, we describe the following task:

Problem: The evaluation problem $Eval_{\varphi,\mathcal{V}}(D, \nu_{\mathrm{in}})$ for φ with input variables $\mathcal{V}$.
Input: A database instance D and a valuation ν_{in} on $\mathcal{V}$.
Output: The set of all valuations ν on $\mathcal{V} \cup \mathrm{FV}(\varphi)$ such that $\nu_{\mathrm{in}} \subseteq \nu$ and $D, \nu \models \varphi$.

As mentioned in the Introduction, this problem is known to be solvable by a relational algebra plan respecting the access patterns. In particular, if D is finite, the output is always finite: each valuation ν in the output can be shown to take only values in $\mathrm{adom}(D) \cup \nu_{\mathrm{in}}(\mathcal{V})$.[1]

[1] Actually, a stronger property can be shown: only values that are "accessible" from ν_{in} in D can be taken [5], and if this accessible set is finite, the output of the evaluation problem is finite. This will also follow immediately from our equivalence between executable FO and $\mathrm{FLIF}^{\mathrm{io}}$.

3 Forward LIF, inputs, and outputs

In this section, we introduce the language FLIF.[2] It will be notationally convenient here to work under the following proviso:

▶ **Proviso 4.** *When we write "valuation" without specifying on which variables it is defined, we assume it is defined on all variables. (Formally, we assume a countably infinite universe of variables.)*

Importantly, we will define the semantics of an FLIF expression in such a way that it depends only on the value of the valuations on the free variables of the expression. This situation is comparable to the classical way in which the semantics of first-order logic is often defined.

The central idea is to view a database as a graph. The nodes of the graph are all possible valuations (hence the graph is infinite.) The edges in the graph are labeled with *atomic FLIF expressions*. Over a schema $\mathcal{S}$, there are five kinds of atomic expressions τ, given by the following grammar:

$$\tau ::= R(\bar{x};\bar{y}) \mid (x = y) \mid (x = c) \mid (x := y) \mid (x := c)$$

Here, $R(\bar{x};\bar{y})$ is a relation atom over $\mathcal{S}$ as in first-order logic, x and y are variables, and c is a constant.

Given an instance D of $\mathcal{S}$, and an atomic expression τ, we define the set of τ-labeled edges in the graph representation of D as a set $[\![\tau]\!]_D$ of ordered pairs of valuations, as follows.

1. $[\![R(\bar{x};\bar{y})]\!]_D$ is the set of all pairs (ν_1, ν_2) of valuations such that the concatenation $\nu_1(\bar{x}) \cdot \nu_2(\bar{y})$ belongs to $D(R)$, and ν_1 and ν_2 agree outside the variables in $\bar{y}$.
2. $[\![(x := y)]\!]_D$ is the set of all pairs (ν_1, ν_2) of valuations such that $\nu_2 = \nu_1[x := \nu_1(y)]$. Thus, $\nu_2(x) = \nu_1(y)$ and ν_2 agrees with ν_1 on all other variables.
3. Similarly, $[\![(x := c)]\!]_D$ is the set of all pairs (ν_1, ν_2) of valuations such that $\nu_2 = \nu_1[x := c]$.
4. $[\![(x = y)]\!]_D$ is the set of all identical pairs (ν, ν) such that $\nu(x) = \nu(y)$.
5. Likewise, $[\![(x = c)]\!]_D$ is the set of all identical pairs (ν, ν) such that $\nu(x) = c$.

The syntax of all FLIF expressions α is now given by the following grammar:

$$\alpha ::= \tau \mid \alpha\,;\alpha \mid \alpha \cup \alpha \mid \alpha \cap \alpha \mid \alpha - \alpha$$

Here, τ ranges over atomic expressions as defined above. The semantics of ';' is composition, defined as follows:

$$[\![\alpha_1\,;\alpha_2]\!]_D = \{(\nu_1, \nu_3) \mid \exists \nu_2 : (\nu_1, \nu_2) \in [\![\alpha_1]\!]_D \text{ and } (\nu_2, \nu_3) \in [\![\alpha_2]\!]_D\}$$

The semantics of the set operations are standard union, intersection and set difference.

We see that FLIF expressions describe paths in the graph, in the form of source–target pairs. Composition is used to navigate through the graph, and to conjoin paths. Paths can be branched using union, merged using intersection, and excluded using set difference.

▶ Remark 5. The way the smantics of FLIF is defined is in line with first-order dynamic logic or dynamic predicate logic (DPL) [13, 12]. DPL gives a dynamic interpretation to existential quantification and interprets conjunction as composition. For example, the FLIF expression $R(x;y)\,;S(y;z)$ would be expressed in DPL as $\exists y\, R(x,y) \wedge \exists z\, S(y,z)$. On the other hand, disjunction in DPL is always interpreted as a test. Because of this, FLIF expressions such as $R(x;y) \cup S(u;v)$ seem inexpressible in DPL.

[2] Pronounced as "eff-lif".

▶ **Example 6.** Consider a simple Facebook abstraction with a single binary relation F of input arity one. When given a person as input, F returns all their friends. We assume that this relation is symmetric. Suppose, for an input person x (say, a famous person), we want to find all people who are friends with at least two friends of x. Formally, we want to navigate from a valuation ν_1 giving a value for x, to all valuations ν_2 giving values to variables y_1, y_2, and z, such that

- $\nu_1(x)$ is friends with both $\nu_2(y_1)$ and $\nu_2(y_2)$;
- $\nu_2(y_1)$ and $\nu_2(y_2)$ are both friends with $\nu_2(z)$; and
- $\nu_2(y_1) \neq \nu_2(y_2)$.

This can be done by the expression $\alpha - (\alpha\,;(y_1 = y_2))$, where α is the expression

$$(F(x;y_1)\,;F(y_1;z)) \cap (F(x;y_2)\,;F(y_2;z)).$$

▶ Remark 7. In the above example, it would be more efficient to simply write $\alpha\,;(y_1 \neq y_2)$. For simplicity, we have not added nonequality tests in FLIF as they are formally redundant in the presence of set difference, but they can easily be added in practice. ◀

In every expression we can identify the *input* and the *output* variables. Intuitively, the output variables are those that can change value along the execution path; the input variables are those whose value at the beginning of the path is needed in order to know the possible values for the output variables. These intuitions will be formalized below. We first give some examples.

▶ **Example 8.**

- In the expression α from Example 6, the only input variable is x, and the other variables are output variables.
- FLIF, in general, allows expressions where a variable is both input and output. For example, assume **dom** contains the natural numbers and consider a binary relation *Inc* of input arity one that holds pairs of natural numbers $(n, n+1)$. Then it is reasonable to use an expression $Inc(x;x)$ to increment the value x. Formally, this expression defines all pairs of valuations (ν_1, ν_2) such that $\nu_2(x) = \nu_1(x) + 1$ (and ν_2 agrees with ν_1 on all other variables).
- Consider the expression $R(x;y_1) \cap S(x;y_2)$. Then not only x, but also y_1 and y_2 are input variables. Indeed, the expression $R(x;y_1)$ will pair an input valuation ν_1 with an output valuation ν_2 that sets y_1 such that $R(\nu_1(x), \nu_2(y_1))$ holds, but ν_2 will have the same value as ν_1 on any other variable. In particular, $\nu_2(y_2) = \nu_1(y_2)$. The expression $S(x;y_2)$ has a similar behavior, but with y_1 and y_2 interchanged. Thus, the intersection expression checks two conditions on the input valuation; formally, it defines all identical pairs (ν, ν) for which $R(\nu(x), \nu(y_1))$ and $S(\nu(x), \nu(y_2))$ hold. Since the expression only tests conditions, it has no output variables.
- On the other hand, for the expression $R(x;y_1) \cup S(x;y_2)$, the output variables are y_1 and y_2. Indeed, consider an input valuation ν_1 with $\nu_1(x) = a$. The expression pairs ν_1 either with a valuation giving a new value for y_1, or with a valuation giving a new value for y_2. However, y_1 and y_2 are also input variables (together with x). Indeed, when pairing ν_1 with a valuation ν_2 that sets y_2 to some b for which $S(a,b)$ holds, we must know the value of $\nu_1(y_1)$ so as to preserve it in ν_2. A similar argument holds for y_2. ◀

Table 1 now formally defines, for any expression α, the sets $I(\alpha)$ and $O(\alpha)$ of input and output variables. We denote the union of $I(\alpha)$ and $O(\alpha)$ by $\mathrm{FV}(\alpha)$. We refer to this set as the *free variables* of α, but note that it actually equals the set of all variables occurring in the expression.

Table 1 Input and output variables of FLIF expressions. In the case of $R(\bar{x};\bar{y})$, the set X is the set of variables in $\bar{x}$, and the set Y is the set of variables in $\bar{y}$. Recall that $\triangle$ is symmetric difference.

α	$I(\alpha)$	$O(\alpha)$
$R(\bar{x};\bar{y})$	X	Y
$(x = y)$	$\{x, y\}$	$\emptyset$
$(x := y)$	$\{y\}$	$\{x\}$
$(x = c)$	$\{x\}$	$\emptyset$
$(x := c)$	$\emptyset$	$\{x\}$
$\alpha_1; \alpha_2$	$I(\alpha_1) \cup (I(\alpha_2) - O(\alpha_1))$	$O(\alpha_1) \cup O(\alpha_2)$
$\alpha_1 \cup \alpha_2$	$I(\alpha_1) \cup I(\alpha_2) \cup (O(\alpha_1) \triangle O(\alpha_2))$	$O(\alpha_1) \cup O(\alpha_2)$
$\alpha_1 \cap \alpha_2$	$I(\alpha_1) \cup I(\alpha_2) \cup (O(\alpha_1) \triangle O(\alpha_2))$	$O(\alpha_1) \cap O(\alpha_2)$
$\alpha_1 - \alpha_2$	$I(\alpha_1) \cup I(\alpha_2) \cup (O(\alpha_1) \triangle O(\alpha_2))$	$O(\alpha_1)$

We next establish three propositions that show that our definition of inputs and outputs, which is purely syntactic, reflects actual properties of the semantics. The first proposition confirms an intuitive property and can be straightforwardly verified by induction.

▶ **Proposition 9** (Law of inertia). *If $(\nu_1, \nu_2) \in [\![\alpha]\!]_D$ then ν_2 agrees with ν_1 outside $O(\alpha)$.*

The second proposition confirms, as announced earlier, that the semantics of expressions depends only on the free variables; outside $\mathrm{FV}(\alpha)$, the binary relation $[\![\alpha]\!]_D$ is cylindrical. The proof for difference expressions is not immediate, and uses the law of inertia.

▶ **Proposition 10** (Free variable property). *Let $(\nu_1, \nu_2) \in [\![\alpha]\!]_D$ and let ν_1' and ν_2' be valuations such that*

- *ν_1' agrees with ν_1 on $\mathrm{FV}(\alpha)$, and*
- *ν_2' agrees with ν_2 on $\mathrm{FV}(\alpha)$, and agrees with ν_1' outside $\mathrm{FV}(\alpha)$.*

Then also $(\nu_1', \nu_2') \in [\![\alpha]\!]_D$.

The third proposition is the most important one, and is proven using the previous two. It confirms that the values for the input variables determine the values for the output variables.

▶ **Proposition 11** (Input determinacy). *Let $(\nu_1, \nu_2) \in [\![\alpha]\!]_D$ and let ν_1' be a valuation that agrees with ν_1 on $I(\alpha)$. Then there exists a valuation ν_2' that agrees with ν_2 on $O(\alpha)$, such that $(\nu_1', \nu_2') \in [\![\alpha]\!]_D$.*

By the law of inertia, the valuation ν_2' given by the above proposition is unique.

We are now in a position to formulate the FLIF evaluation problem. Given an expression α, we consider the following task:[3]

Problem: The evaluation problem $Eval_\alpha(D, \nu_{\text{in}})$ for α.
Input: A database instance D and a valuation ν_{in} on $I(\alpha)$.
Output: The set $\{\nu_{\text{out}}|_{\mathrm{FV}(\alpha)} \mid \exists \nu_{\text{in}}' : \nu_{\text{in}} \subseteq \nu_{\text{in}}' \text{ and } (\nu_{\text{in}}', \nu_{\text{out}}) \in [\![\alpha]\!]_D\}$.

[3] For a valuation ν on a set of variables X (possibly all variables), and a subset Y of X, we use $\nu|_Y$ to denote the restriction of ν to X.

By inertia and input determinacy, the choice of ν'_{in} in the definition of the output does not matter. Moreover, if D is finite, the output is finite as well. As was the case for executable FO, the above problem can be solved by a relational algebra plan respecting the access patterns. Unfortunately, since the sets of input and output variables of general FLIF expressions need not be disjoint, the plan is a bit intricate; we have to work with relations that have two copies for every variable, to keep track of how assignments are paired up.

For this reason, in the next section, we introduce a well-behaved fragment called *io-disjoint* FLIF. Plans for expressions in this fragment can be generated in a very transparent manner, as is shown in Section 5.

4 Executable FO and io-disjoint FLIF

Consider an FLIF expression α for which the set $O(\alpha)$ is disjoint from $I(\alpha)$. Then any pair $(\nu_1, \nu_2) \in [\![\alpha]\!]_D$ satisfies that ν_1 and ν_2 are equal on $I(\alpha)$. Put differently, every $\nu_{\text{out}} \in \mathit{Eval}_\alpha(D, \nu_{\text{in}})$ is equal to ν_{in} on $I(\alpha)$; all that the evaluation does is expand the input valuation with output values for the new output variables. This makes the evaluation process for expressions α where $I(\beta) \cap O(\beta) = \emptyset$, for every subexpression β of α (including α itself), very transparent. We call such expressions *io-disjoint*.

The following proposition makes it easier to check if an expression is io-disjoint:

▶ **Proposition 12.** *The following alternative definition of io-disjointness is equivalent to the definition given above:*

- *An atomic expression $R(\bar{x}; \bar{y})$ is io-disjoint if $X \cap Y = \emptyset$, where X is the set of variables in $\bar{x}$, and Y is the set of variables in $\bar{y}$.*
- *Atomic expressions of the form $(x = y)$, $(x = c)$, $(x := y)$ or $(x := c)$ are io-disjoint.*
- *A composition $\alpha_1 \,;\, \alpha_2$ is io-disjoint if α_1 and α_2 are, and moreover $I(\alpha_1) \cap O(\alpha_2) = \emptyset$.*
- *A union $\alpha_1 \cup \alpha_2$ is io-disjoint if α_1 and α_2 are, and moreover $O(\alpha_1) = O(\alpha_2)$.*
- *An intersection $\alpha_1 \cap \alpha_2$ is io-disjoint if α_1 and α_2 are.*
- *A difference $\alpha_1 - \alpha_2$ is io-disjoint if α_1 and α_2 are, and moreover $O(\alpha_1) \subseteq O(\alpha_2)$.*

The fragment of io-disjoint expressions is denoted by FLIF^{io}. We are going to show that FLIF^{io} is expressive enough, in the sense that executable FO can be translated into FLIF^{io}. The converse translation is also possible, so, FLIF^{io} exactly matches executable FO in expressive power.

Recall the evaluation problem for executable FO, as defined at the end of Section 2, and the evaluation problem for α, as defined at the end of the previous section. We can now formulate the translation result from executable FO to FLIF^{io} as follows.

▶ **Theorem 13.** *Let φ be a $\mathcal{V}$-executable formula over schema $\mathcal{S}$. There exists an FLIF^{io} expression α over $\mathcal{S}$ with the following properties:*

1. *$I(\alpha) = \mathcal{V}$.*
2. *$O(\alpha) \supseteq \text{FV}(\varphi) - \mathcal{V}$.*
3. *For every D and ν_{in}, we have $\mathit{Eval}_{\varphi,\mathcal{V}}(D, \nu_{\text{in}}) = \pi_{\text{FV}(\varphi)\cup\mathcal{V}}(\mathit{Eval}_\alpha(D, \nu_{\text{in}}))$.*

The length of α is polynomial in the length of φ and the cardinality of $\mathcal{V}$.

The above projection operator π restricts each valuation in $\mathit{Eval}_\alpha(D, \nu_{\text{in}})$ to $\text{FV}(\varphi) \cup \mathcal{V}$. It is imposed because we allow $O(\alpha)$ to have auxiliary variables not in $\text{FV}(\varphi)$.

▶ **Example 14.** Before giving the proof, we give a few examples.

- Suppose φ is $R(x; y)$ with input variable x. Then, as expected, α can be taken to be $R(x; y)$.

- However, now consider $T(x; x, y)$, again with input variable x. Intuitively, the formula asks for outputs (u, y) where u equals x. Hence, a suitable io-disjoint translation is $T(x; u, y) \,;\, (u = x)$.
- If φ is $R(x; y) \wedge S(y; z)$, still with input variable x, we can take $R(x; y) \,;\, S(y; z)$ for α. The same expression also serves for the formula $\exists y\, \varphi$. However, if φ is $\exists y\, R(x; y)$ with $\mathcal{V} = \{x, y\}$, we must use a fresh variable and use $R(x; u) \,;\, (y = y)$ for α. The test $(y = y)$ may seem spurious but is needed to ensure that $I(\alpha) = \mathcal{V}$.
- Suppose φ is $R(x; x) \vee S(y;)$ with $\mathcal{V} = \{x, y\}$. For this $\mathcal{V}$, we translate $R(x; x)$ to $R(x; u) \,;\, (x = u) \,;\, (y = y)$. Similarly, $S(y;)$ is translated to $S(y;) \,;\, (x = x)$. Unfortunately the union of these two expressions is not io-disjoint. We can formally solve this by composing the second expression with a dummy assignment to u. So the final α can be taken to be $R(x; u) \,;\, (x = u) \,;\, (y = y) \cup S(y;) \,;\, (x = x) \,;\, (u := 42)$. Since the output valuations will be projected on $\{x, y\}$, the choice of the constant assigned to u is irrelevant.
- A similar trick can be used for negation. For example, if φ is $\neg R(x; y)$ with $\mathcal{V} = \{x, y\}$, then α can be taken to be $(u := 42) \;-\; R(x; u) \,;\, (u = y) \,;\, (u := 42)$.

Proof. We only describe the translation; its correctness, which hinges on the law of inertia and input determinacy, also involves verifying that io-disjointness holds.

If φ is a relation atom $R(\bar{x}; \bar{y})$, then α is $R(\bar{x}; \bar{z}) \,;\, \xi \,;\, \xi'$, where $\bar{z}$ is obtained from $\bar{y}$ by replacing each variable from $\mathcal{V}$ by a fresh variable. The expression ξ consists of the composition of all equalities $(y_i = z_i)$ where y_i is a variable from $\bar{y}$ that is in $\mathcal{V}$ and z_i is the corresponding fresh variable. The expression ξ' consists of the composition of all equalities $(u = u)$ with u a variable in $\mathcal{V}$ not mentioned in φ.

If φ is $x = y$, then α is

$$\begin{cases} (x = y) \,;\, \xi & \text{if } x, y \in \mathcal{V} \\ (x := y) \,;\, \xi & \text{if } x \notin \mathcal{V} \\ (y := x) \,;\, \xi & \text{if } y \notin \mathcal{V}, \end{cases}$$

where ξ is the composition of all equalities $(u = u)$ with u a variable in $\mathcal{V}$ not mentioned in φ.

If φ is $x = c$, then α is

$$\begin{cases} (x = c) \,;\, \xi & \text{if } x \in \mathcal{V} \\ (x := c) \,;\, \xi & \text{otherwise,} \end{cases}$$

with ξ as in the previous case.

If φ is $\varphi_1 \wedge \varphi_2$, then by induction we have an expression α_1 for φ_1 and $\mathcal{V}$, and an expression α_2 for φ_2 and $\mathcal{V} \cup \mathrm{FV}(\varphi_1)$. Now α can be taken to be $\alpha_1 \,;\, \alpha_2$.

If φ is $\exists x\, \varphi_1$, then without loss of generality we may assume that $x \notin V$. By induction we have an expression α_1 for φ_1 and $\mathcal{V}$. This expression also works for φ.

If φ is $\varphi_1 \vee \varphi_2$, then by induction we have an expression α_i for φ_i and $\mathcal{V}$, for $i = 1, 2$. Fix an arbitrary constant c, and let ξ_1 be the composition of all expressions $(z := c)$ for $z \in O(\alpha_2) - O(\alpha_1)$; let ξ_2 be defined symmetrically. Now α can be taken to be $\alpha_1 \,;\, \xi_1 \cup \alpha_2 \,;\, \xi_2$.

Finally, if φ is $\neg\varphi_1$, then by induction we have an expression α_1 for φ_1 and $\mathcal{V}$. Fix an arbitrary constant c, and let ξ be the composition of all expressions $(z := c)$ for $z \in O(\alpha_1)$. (If $O(\alpha_1)$ is empty, we add an additional fresh variable.) Then α can be taken to be $\xi \;-\; \alpha_1 ; \xi$. ◀

We next turn to the converse translation. Here, a sharper equivalence is possible, since executable FO has an explicit quantification operation which is lacking in FLIF.

Table 2 Translation showing how FLIF$^{\text{io}}$ embeds in executable FO. In the table, φ_i abbreviates φ_{α_i} for $i = 1, 2$.

α	φ_α
$R(\bar{x};\bar{y})$	$R(\bar{x};\bar{y})$
$(x = y)$	$x = y$
$(x := y)$	$x = y$
$x = c$	$x = c$
$x := c$	$x = c$
$\alpha_1;\alpha_2$	$(\exists x_1 \dots \exists x_k\, \varphi_1) \wedge \varphi_2$ where $\{x_1, \dots, x_k\} = O(\alpha_1) \cap O(\alpha_2)$
$\alpha_1 \cup \alpha_2$	$\varphi_1 \vee \varphi_2$
$\alpha_1 \cap \alpha_2$	$\varphi_1 \wedge \varphi_2$
$\alpha_1 - \alpha_2$	$\varphi_1 \wedge \neg\varphi_2$

▶ **Theorem 15.** *Let α be an* FLIF$^{\text{io}}$ *expression over schema $\mathcal{S}$. There exists an $I(\alpha)$-executable FO formula φ_α over $\mathcal{S}$, with* $\mathrm{FV}(\varphi_\alpha) = \mathrm{FV}(\alpha)$*, such that for every D and ν_{in}, we have $Eval_\alpha(D, \nu_{\text{in}}) = Eval_{\varphi_\alpha, I(\alpha)}(D, \nu_{\text{in}})$. The length of φ_α is linear in the length of α.*

▶ **Example 16.** To illustrate the proof, consider the expression $R(x; y, u)\,; S(x; z, u)$. Procedurally, this expression first retrieves a (y, u)-binding from R for the given x. It proceeds to retrieve a (z, u)-binding from S for the given x, effectively overwriting the previous binding for u. Thus, a correct translation into executable FO is $(\exists u\, R(x; y, u)) \wedge S(x; z, u)$.

For another example, consider the assignment $(x := y)$. This translates to $x = y$ considered as a $\{y\}$-executable formula. The equality test $(x = y)$ also translates to $x = y$, but considered as an $\{x, y\}$-executable formula.

Proof. Table 2 shows the translation, which is almost an isomorphic embedding, except for the case of composition. The correctness of the translation for composition again hinges on inertia and input determinacy. ◀

Notably, in the proof of Theorem 13, we do not need the intersection operation. Hence, by translating FLIF$^{\text{io}}$ to executable FO and then back to FLIF$^{\text{io}}$, we obtain that intersection is redundant in FLIF$^{\text{io}}$, in the following sense:

▶ **Corollary 17.** *For every* FLIF$^{\text{io}}$ *expression α there exists a* FLIF$^{\text{io}}$ *expression α' with the following properties:*

1. *α' does not use the intersection operation.*
2. *$I(\alpha') = I(\alpha)$.*
3. *$O(\alpha') \supseteq O(\alpha)$.*
4. *For every D and ν_{in}, we have $Eval_\alpha(D, \nu_{\text{in}}) = \pi_{\mathrm{FV}(\alpha)}(Eval_{\alpha'}(D, \nu_{\text{in}}))$.*

▶ Remark 18. One may wonder whether the above corollary directly follows from the equivalence between $\alpha_1 \cap \alpha_2$ and $\alpha_1 - (\alpha_1 - \alpha_2)$. While these two expressions are semantically equivalent and have the same input variables, they do not have the same output variables, so a simple inductive proof eliminating intersection while preserving the guarantees of the above corollary does not work. Moreover, the corollary continues to hold for the positive fragment of FLIF$^{\text{io}}$ (without the difference operation). Indeed, positive FLIF$^{\text{io}}$ can be translated into executable FO without negation, which can then be translated into positive FLIF$^{\text{io}}$ without intersection.

5 Relational algebra plans for io-disjoint FLIF

In this section we show how the evaluation problem for FLIF$^{\text{io}}$ expressions can be solved in a very direct manner, using a translation into a particularly simple form of relational algebra plans.

We generalize the evaluation problem so that it can take a set of valuations as input, rather than just a single valuation. Formally, for an FLIF$^{\text{io}}$ expression α over database schema $\mathcal{S}$, an instance D of $\mathcal{S}$, and a set N of valuations on $I(\alpha)$, we want to compute $Eval_\alpha(D, N) := \bigcup\{Eval_\alpha(D, \nu_{\text{in}}) \mid \nu_{\text{in}} \in N\}$.

Viewing variables as attributes, we can view a set of valuations on a finite set of variables Z, like the set N above, as a relation with relation schema Z. Consequently, it is convenient to use the named perspective of the relational algebra [2], where every expression has an output relation schema (a finite set of attributes; variables in our case). We briefly review the well-known operators of the relational algebra and their behavior on the relation schema level:

- Union and difference are allowed only on relations with the same relation schema.
- Natural join ($\bowtie$) can be applied on two relations with relation schemas Z_1 and Z_2, and produces a relation with relation schema $Z_1 \cup Z_2$.
- Projection (π) produces a relation with a relation schema that is a subset of the input relation schema.
- Selection (σ) does not change the schema.
- Renaming will not be needed. Instead, however, to accommodate the assignment expressions present in FLIF, we will need the generalized projection operator that adds a new attribute with the same value as an existing attribute, or a constant. Let N be a relation with relation schema Z, let $y \in Z$, and let x be a variable not in Z. Then

$$\pi_{Z,x:=y}(N) = \{\nu[x := \nu(y)] \mid \nu \in N\}$$
$$\pi_{Z,x:=c}(N) = \{\nu[x := c] \mid \nu \in N\}$$

Plans are based on *access methods*, which have the following syntax and semantics. Let $R(\bar{x};\bar{y})$ be an atomic FLIF$^{\text{io}}$-expression. Let X be the set of variables in $\bar{x}$ and let Y be the set of variables in $\bar{y}$ (in particular, X and Y are disjoint). Let N be a relation with a relation schema Z that contains X but is disjoint from Y. Let D be a database instance. We define the result of the *access join* of N with $R(\bar{x};\bar{y})$, evaluated on D, to be the following relation with relation schema $Z \cup Y$:

$$N \overset{\text{access}}{\bowtie} R(\bar{x};\bar{y}) := \{\nu \text{ valuation on } Z \cup Y \mid \nu|_Z \in N \text{ and } \nu(\bar{x}) \cdot \nu(\bar{y}) \in D(R)\}$$

This result relation can clearly be computed respecting the limited access pattern on R. Indeed, we iterate through the valuations in N, feed their X-values to the source R, and extend the valuations with the obtained Y-values.

Formally, over any database schema $\mathcal{S}$ and for any finite set of variables I, we define a *plan over $\mathcal{S}$ with input variables I* as an expression that can be built up as follows:

- The special relation name In, with relation schema I, is a plan.
- If $R(\bar{x};\bar{y})$ is an atomic FLIF$^{\text{io}}$ expression over $\mathcal{S}$, with sets of variables X and Y as above, and E is a plan with output relation schema Z as above, then also $E \overset{\text{access}}{\bowtie} R(\bar{x};\bar{y})$ is a plan, with output relation schema $Z \cup Y$.
- Plans are closed under union, difference, natural join, and projection.

Given a database instance D, a set N of valuations on I, and a plan E with input variables I, we can instantiate the relation name In by N and evaluate E on (D, N) in the obvious manner. We denote the result by $E(D, N)$.

We establish:

▶ **Theorem 19.** *For every* FLIF^{io} *expression* α *over database schema* $\mathcal{S}$ *there exists a plan* E_α *over* $\mathcal{S}$ *with input variables* $I(\alpha)$*, such that* $Eval_\alpha(D, N) = E_\alpha(D, N)$*, for every instance* D *of* $\mathcal{S}$ *and set* N *of valuations on* $I(\alpha)$*.*

▶ **Example 20.**

- A plan for $R(x; y)\,;S(y; z)$ is $(In \overset{\text{access}}{\bowtie} R(x; y)) \overset{\text{access}}{\bowtie} S(y; z)$.
- A plan for $R(x_1; y, u)\,;S(x_2, y; z, u)$ is

  $$\pi_{x_1,x_2,y}(In \overset{\text{access}}{\bowtie} R(x_1; y, u)) \overset{\text{access}}{\bowtie} S(x_2, y; z, u).$$

- Recall the expression $R(x; y_1) \cap S(x; y_2)$ from Example 8, which has input variables $\{x, y_1, y_2\}$ and no output variables. A plan for this expression is

  $$(\pi_{x,y_2}(In) \overset{\text{access}}{\bowtie} R(x; y_1)) \bowtie In \;\cap\; (\pi_{x,y_1}(In) \overset{\text{access}}{\bowtie} S(x; y_2)) \bowtie In.$$

  The joins with In ensure that the produced output values are equal to the given input values.

Proof. To prove the theorem we need a stronger induction hypothesis, where we allow N to have a larger relation schema $Z \supseteq I(\alpha)$, while still being disjoint with $O(\alpha)$. The claim then is that

$$E_\alpha(D, N) = \{\nu \text{ on } Z \cup O(\alpha) \mid \nu|_{\text{FV}(\alpha)} \in Eval_\alpha(D, \nu|_{I(\alpha)})\}.$$

The base cases are clear. If α is $R(\bar{x}; \bar{y})$, then E_α is $In \overset{\text{access}}{\bowtie} R(\bar{x}; \bar{y})$ for E_α. If α is $(x = y)$, then E_α is the selection $\sigma_{x=y}(In)$. If α is $(x := y)$, then E_α is the generalized projection $\pi_{y,x:=y}(In)$.

In what follows we use the following notation. Let P and Q be plans. By $Q(P)$ we mean the plan obtained from Q by substituting P for In.

Suppose α is $\alpha_1\,;\alpha_2$. Plan E_{α_1}, obtained by induction, assumes an input relation schema that contains $I(\alpha_1)$ and is disjoint from $O(\alpha_1)$. Since $I(\alpha) = I(\alpha_1) \cup (I(\alpha_2) - O(\alpha_1))$, $I(\alpha_1) \cap O(\alpha_1) = \emptyset$, and Z is disjoint from $O(\alpha) = O(\alpha_1) \cup O(\alpha_2)$, we can apply E_{α_1} with input relation schema Z. Let P_1 be the plan $\pi_{Z-O(\alpha_2)}(E_{\alpha_1})$. Then E_α is the plan $E_{\alpha_2}(P_1)$. (One can again verify that this is a legal plan.)

Next, suppose α is $\alpha_1 \cup \alpha_2$. Then $I(\alpha) = I(\alpha_1) \cup I(\alpha_2)$, which is disjoint from $O(\alpha_1) = O(\alpha_2)$ (compare Proposition 12). Hence for E_α we can simply take the plan $E_{\alpha_1} \cup E_{\alpha_2}$.

Next, suppose α is $\alpha_1 \cap \alpha_2$. Note that $I(\alpha) = I(\alpha_1) \cup I(\alpha_2) \cup (O(\alpha_1) \triangle O(\alpha_2))$. Now E_α is

$$E_{\alpha_1}(\pi_{I(\alpha)-O(\alpha_1)}(In)) \bowtie In \;\cap\; E_{\alpha_2}(\pi_{I(\alpha)-O(\alpha_2)}(In)) \bowtie In.$$

Finally, suppose α is $\alpha_1 - \alpha_2$. Then E_α is

$$E_{\alpha_1} - (E_{\alpha_2}(\pi_{I(\alpha)-O(\alpha_2)}(In)) \bowtie In.$$

In general, in the above translations, we follow the principle that the result of a subplan E_{α_i} must be joined with In whenever $O(\alpha_i)$ may intersect with $I(\alpha)$. ◀

▶ Remark 21. When we extend plans with assignment statements such that common expressions can be given a name [5], the translation given in the above proof leads to a plan E_α of size linear of the length of α. Each time we do a substitution of a subexpression for In in the proof, we first assign a name to the subexpression and only substitute the name.

6 Conclusion

Nash and Ludäscher [15] deserve credit for having come up with executable FO as a beautiful declarative query language that strikes a perfect balance between first-order logic expressiveness and the limitations imposed by the access patterns on the information sources. On the other hand, relational algebra plans are more operational and rather low-level. We think of FLIF as an intermediate language between the two levels. FLIF is still declarative, as it is still a logic, be it an algebraic one. On the other hand FLIF is also operational, in view of its dynamic semantics akin to dynamic logics [13] and navigational graph query languages. For us, the main novelty of FLIF lies in the mechanism of input and output variables, and the law of inertia.

The book by Benedikt et al. [5] stands as an authoritative reference on the topic of querying under limited access patterns. Remarkably, Benedikt et al. do not follow Nash and Ludäscher's proposal, but use their own, quite different notion of executable first-order query. This notion involves a two-step process where, first, an executable UCQ (union of conjunctive queries) retrieves a set of tuples from the sources, which is then filtered by a first-order condition that is "executable for membership". The filter condition must be expressed in a range-restricted version of first-order logic. In a result similar to our Theorem 19, Benedikt et al. then proceed to show [5, Theorem 3.4] that their executable FO queries are equivalent in expressive power to plans. We feel that our work makes a contribution, enabled by the LIF perspective, by providing a more declarative formalism, a simpler format of plans, and more streamlined translations between the languages.

On the other hand we should stress that the main strength of the work by Benedikt et al. lies elsewhere, namely, in matching semantic properties to syntactic restrictions, for a variety of settings and languages. In this respect, we recall the result [5, Theorem 3.9] already mentioned in the Introduction, to the effect that every "access-determined" boolean first-order query has a plan. This result, proven using model-theoretic interpolation, assumes access-determinacy over unrestricted structures (not necessarily finite). It is open whether a similar result holds in restriction to finite structures.

Our three results (Theorems 13, 15 and 19) exploit the good properties enjoyed by io-disjointness of FLIF expressions. However, as far as expressive power is concerned, io-disjointness may not be a real restriction. Indeed, we conjecture that that every FLIF expression is equivalent, modulo variable renaming, to a FLIF^{io} expression that can use more variables.

Another topic for further research concerns our definition of inputs and outputs of FLIF expressions (Table 1). While guaranteeing the properties of inertia and input determinacy, this definition cannot be complete in this respect, as said properties are undecidable. Yet, the definition may be "locally" optimal in some sense analogous to an optimality result obtained for the notion of controlled formula [10, Proposition 4.3].

Finally, it would be interesting to look more closely into the practical aspects of the plans generated for FLIF^{io} expressions. We have shown that these plans have linear size, do not need renaming, and the only joins are natural joins. Does this lead to more efficiency or better optimizability?

In closing, we note that querying under limited access patterns has applicability beyond traditional data or information sources. For instance in the context of distributed data, when performing tasks involving the composition of external services, functions, or modules, limited access patterns are a way for service providers to protect parts of their data, while still allowing their services to be integrated seamlessly in other applications. Limited access patterns also have applications in active databases, where we like to think of FLIF as an analogue of Active XML [1] for the relational data model.

References

1. S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *The VLDB Journal*, 17(5):1019–1040, 2008.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
3. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.
4. R. Angles, P. Barceló, and G. Rios. A practical query language for graph DBs. In L. Bravo and M. Lenzerini, editors, *Proceedings 7th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 1087 of *CEUR Workshop Proceedings*, 2013.
5. M. Benedikt, J. Leblay, B. ten Cate, and E. Tsamoura. *Generating Plans from Proofs: The Interpolation-based Approach to Query Reformulation*. Morgan & Claypool, 2016.
6. M. Benedikt, B. ten Cate, and E. Tsamoura. Generating plans from proofs, 2016.
7. A. Calì, D. Calvanese, and D. Martinenghi. Dynamic query optimization under access limitations and dependencies. *Journal of Universal Computer Science*, 15(1):33–62, 2009.
8. A. Calì, D. Martinenghi, I. Razon, and M. Ugarte. Querying the deep web: Back to the foundations. In J.L. Reutter and D. Srivastava, editors, *Proceedings 11th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 1912 of *CEUR Workshop Proceedings*, 2017.
9. A. Calì and M. Ugarte. On the complexity of query answering under access limitations: A computational formalism. In D. Olteanu and B. Poblete, editors, *Proceedings 12th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 2100 of *CEUR Workshop Proceedings*, 2018.
10. W. Fan, F. Geerts, and L. Libkin. On scale independence for querying big data. In *Proceedings 33th ACM Symposium on Principles of Database Systems*, pages 51–62, 2014.
11. G.H.L. Fletcher, M. Gyssens, D. Leinders, D. Surinx, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs. *Information Sciences*, 298:390–406, 2015.
12. J. Groenendijk and M. Stokhof. Dynamic predicate logic. *Linguistics and Philosophy*, 14:39–100, 1991.
13. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
14. L. Libkin, W. Martens, and D. Vrgoč. Quering graph databases with XPath. In *Proceedings 16th International Conference on Database Theory*. ACM, 2013.
15. A. Nash and B. Ludäscher. Processing first-order queries under limited access patterns. In *Proceedings 23th ACM Symposium on Principles of Database Systems*, pages 307–318, 2004.
16. J. Pérez, M. Arenas, and C. Gutierrez. nSPARQL: A navigational language for RDF. *Journal of Web Semantics*, 8(4):255–270, 2010.
17. D. Surinx, G.H.L. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, S. Vansummeren, and Y. Wu. Relative expressive power of navigational querying on graphs using transitive closure. *Logic Journal of the IGPL*, 23(5):759–788, 2015.
18. E. Ternovska. Recent progress on the algebra of modular systems. In J.L. Reutter and D. Srivastava, editors, *Proceedings 11th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 1912 of *CEUR Workshop Proceedings*, 2017.
19. E. Ternovska. An algebra of modular systems: static and dynamic perspectives. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems: Proceedings 12th FroCos*, volume 11715 of *Lecture Notes in Artificial Intelligence*, pages 94–111. Springer, 2019.