Research Report: Parsing with the Logic FC

Owen M. Bell
Loughborough University
Loughborough, UK

Sam M. Thompson
Loughborough University
Loughborough, UK

Dominik D. Freydenberger
Loughborough University
Loughborough, UK

***Abstract*—FC is a logic on strings that has been primarily studied in database theory for the purpose of information extraction. In this report, we argue that it can be used for more. In particular, we explain how FC and its various extensions can be used as a unifying framework for combining parsers that aligns with the principles of Language-Theoretic Security (LangSec). We first survey the recent literature on FC and its extensions, and explain the different criteria we have for efficiency. We then describe how FC and its extensions can be seen as a replacement for regex, and contextualise FC with Language-Theoretic Security. Finally, we explain how, due to the natural compositionality of the model, we can pull the extensions of FC together into a framework for combining parsers.**

***Index Terms*—Finite model theory, first-order logic, parsing.**

I. Introduction

The logic FC was introduced by Freydenberger and Peterfreund [1] as a first-order logic over finite strings. One of the original motivations for this logic is *information extraction*: The problem of extracting relevant information from unstructured textual data. In this research report, we argue that FC and its extensions do not only constitute a logic for information extraction, but also a unifying framework for text querying, specifications, parsing, and more.

Language-Theoretic Security (or LangSec) proposes a solution for input-handling vulnerabilities; those that occur when the parser (which filters valid input to the rest of the program) does not sufficiently invalidate malicious input. This solution is to specify valid inputs as a formal language and use a recognizer for this language as the input-handling routine (see e. g. [2]). We propose that, in addition to its original motivation as a tool for information extraction, FC and its extensions represent a framework for declarative input-handling, where we can define the valid inputs as a formal language as before, but use logic rather than a grammar as a recognizer.

Furthermore, FC is declarative (i.e., formulas express the desired outcome rather than the computation), but both FC and its extensions can use techniques from database theory and finite model theory to become "efficiently declarative". That is, formulas come with a "lens" of how they can be evaluated efficiently. Such lenses give us an evaluation algorithm and use common techniques from database theory such as *quantifier rank* (the nesting depth of the quantifiers), *width* (the maximum number of free variables in any subformula), *treewidth* (intuitively, how similar a concatenation term's graph is to a tree-like structure), and *acyclicity* (if the query has a so-called *join tree*). For more details on these criteria see Section III.

This report surveys a body of recent publications and ongoing work that have addressed FC and its extensions from a theoretical perspective. In the remainder of this section, we give an informal description of FC and its extensions, and give the formal definitions in Section II. In Section III we provide a short tour of how FC can be restricted and extended, and how this give us the "lenses" to evaluate formulas efficiently. We then address how FC aligns with the LangSec principles (see e. g. [3]). Finally, in Section VI, we show how we can put all of this together to obtain a unifying framework for combining parsers.

A Quick Overview of FC: The fundamental building blocks of FC are concatenation terms of the form $x \doteq \alpha$ where x is a variable and α is a string of terminal symbols and variables. These concatenation terms are *string equations* (also called *word equations*) which have been widely examined; for example, see [4]–[7].

Variables in FC range over the *factors* (contiguous substrings) of some finite input text and therefore we can use FC to search for specific factors. For example, $\exists x \colon x \doteq \mathsf{banana}$ asks whether there exists some factor "banana". Using first-order logic connectives, we can build formulas in a modular fashion. If we wanted to ask whether there are *no* factors of the form banana, we simply negate the previous formula: $\neg\exists x \colon x \doteq \mathsf{banana}$.

With the use of variables, we can generalize the search to look for more "abstract" patterns. Consider

$$\exists x, y, z \colon x \doteq \mathsf{a}yz\mathsf{a}y.$$

This formula asks whether there are two (non-overlapping) occurrences of a factor that starts with an a symbol[1].

FC-formulas define languages in a straightforward manner: The set of all strings for which the formulas is true. We reserve a special symbol $\mathfrak{s}$ called the *universe variable* which represents the whole (input) string. For example, if we assume the terminal alphabet $\Sigma = \{\mathsf{a}, \mathsf{b}\}$, the formula

$$\exists x \colon \mathfrak{s} \doteq \mathsf{a}x\mathsf{a}x \wedge \forall y, z \colon \neg(x \doteq y\mathsf{a}z)$$

defines the language $\{\mathsf{ab}^n\mathsf{ab}^n \mid n \geqslant 0\}$. For some intuition, the previous formula states that our whole string is $\mathsf{a}x\mathsf{a}x$ for some terminal string x, *and* for all factors y and z we do not have $x = y\mathsf{a}z$ (in other words, x does not contain an a symbol).

[1]To distinguish between terminal symbols and variables, we use a sans serif typestyle for terminal symbols.

Having *free variables* (occurrences of variables that are not bound by any quantifier) allows us to query a text and "return" a relation. For example, "Return a unary relation of all factors that do not contain an a symbol" can be expressed in FC as $\varphi(x) := \forall y, z \colon \neg(x \doteq y\mathsf{a}z)$.

As we discuss in Section III, FC has decidable *model checking*. In the context of parsing, model checking represents recognition: the part of the parser that accepts or rejects the input. Moreover, relations extracted from the text by an FC-formula can represent the syntactic structure of the input, and thus, could be used as an alternative to a parse tree.

FC as a Logical Framework: FC may not be the ideal logic in every scenario. For example, FC may not have the expressive power to define what you wish to define (see [8]). Alternatively, a particular use case may not require all the features of FC, and a much more tractable subclass would be sufficient. To combat this common negotiation between expressive power and tractability, we shall look three approaches; *constraints*, *fragments*, and *recursion*.

Constraints are a concise way to increase the expressive power of FC. For example, FC cannot define all the regular languages.[2] Therefore, (when necessary) we may introduce *regular constraints*. That is, a new atomic formula of the form $x \mathrel{\dot\in} \gamma$ where x is a variable and γ is a regular expression. This states that x must be replaced with a string from the language of γ (note that x must also be a factor of our input string). If one wanted to test whether a string does not adhere to a (simplified) email address specification, they could write

$$\neg\exists x, y, z \colon (\mathfrak{s} \doteq x \mathbin{@} y \mathbin{.} z) \wedge (x \mathrel{\dot\in} \gamma) \wedge (y \mathrel{\dot\in} \gamma) \wedge (z \mathrel{\dot\in} \gamma),$$

where @ and . are terminal symbols[3] and γ is a regular expression that only accepts strings that use alphanumeric characters (the astute reader may have noticed that regular constraints are not strictly necessary for this example). So far, only regular constraints have been considered in the literature for FC (see [1], [8], [9]), however, any arbitrary constraints can be added to FC.

Adding constraints does not change the complexity of model checking, assuming the constraint is efficient to evaluate (we will discuss some examples of efficient constraints in Section III). In fact, the addition of certain constraints leaves room for some "engineering-style" optimizations (although this may not improve the worst-case complexity).

For example, consider the FC-formula $\forall x \colon (\varphi \wedge x \mathrel{\dot\in} C)$ where φ is some formula and $x \mathrel{\dot\in} C$ is some constraint. Now, if $x \mathrel{\dot\in} C$ only holds for very few factors, then we could evaluate φ only for those x such that $x \mathrel{\dot\in} C$ holds. Thus, constraints can be seen as a way to "filter out" certain factors that do not need to be considered. This leads us to using the notation $\forall x \mathrel{\dot\in} C \colon \varphi$ as an alternative to $\forall x \colon (\varphi \wedge x \mathrel{\dot\in} C)$. Analogously, we adopt the notation $\exists x \mathrel{\dot\in} C \colon \varphi$.

[2] A paper which proves this result is to appear at LICS 2025.

[3] In this example, we use . as a terminal symbol, not as a wildcard character (as it is commonly used in regex engines). Wildcards and other features that appear in implementations open up many further theoretical questions. Here, we use regular expressions in the strict theoretical sense.

As with other logics, one can consider *fragments* of FC – usually defined as a syntactic restriction. Consider a *conjunctive query* fragment of FC: That is, FC-formulas built from atomic formulas ($x \doteq \alpha$), conjunction ($\wedge$), and existential quantification ($\exists$). Although model checking for the conjunctive query fragment of FC is NP-complete, we can draw upon existing conjunctive query literature, such as *acyclicity*, to get polynomial-time model checking [9].

Another fragment (considered in [1]) is the *existential-positive fragment of* FC. Existential-positive FC (EP-FC for short) extends FC conjunctive queries with disjunction ($\vee$); thus avoids negation and the universal quantifier which makes things expensive [1]. Again, while model checking for EP-FC is NP-complete, Freydenberger and Peterfreund [1] considered EP-FC *with bounded width* which allows for polynomial-time model checking. Of course, one can combine tractable fragments of FC with constraints.

Recursion can be added to the model if further expressive power is required. As examined in [10], FC-Datalog is a variant of the query language Datalog for strings that extends the conjunctive query fragment of FC with recursion in the same way that Datalog extends first-order logic (see [11] for more details). An FC-Datalog *program* is given by a set of *rules* of the form

$$R(x_1, \ldots, x_k) \leftarrow \varphi_1,\ \varphi_2,\ \ldots,\ \varphi_n,$$

where each φ_i is an FC atomic formula (i. e., $x \doteq \alpha$) or a relation. For ease of notation we adopt the syntax used in Datalog. Here $x_1, \ldots, x_k$ are free variables and all other variables from $\varphi_1,\ \varphi_2,\ \ldots,\ \varphi_n$ are bound by implicit existential quantifiers. We also use "," symbols to represent conjunction (as opposed to non-recursive FC where $\wedge$ is used). Without the shorthand notation, the above rule would be

$$R(x_1, \ldots, x_k) \leftarrow \exists y_1, \ldots y_m \colon \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n,$$

Initially, we set all the relations to the empty set, and then iteratively apply a rule until all the relations stabilize. We use the special output relation Ans. Thus, an FC-Datalog program can define a relation or, if Ans has arity 0, a language.

Example 1 *The* FC-Datalog *program:*

$$\begin{aligned} \mathrm{Ans}() &\leftarrow \mathfrak{s} \doteq yy,\ \ R(y);\\ R(x) &\leftarrow x \doteq y\mathsf{a},\ \ R(y);\\ R(x) &\leftarrow x \doteq y\mathsf{b},\ \ R(y);\\ R(x) &\leftarrow x \doteq \varepsilon. \end{aligned}$$

defines the language $\{uu \mid u \in \{\mathsf{a}, \mathsf{b}\}^*\}$. *See Example 10 for an example execution of this program.*

II. Preliminaries

In this section, we give some notational conventions, and formal definition of FC based on the definitions given in [1]. Those readers who are less interested in the formal details are invited to skip over this section.

A. Basic Notation

Let Σ be a fixed and finite alphabet of *terminal symbols*, with at least two elements. By Σ^*, we denote all finite strings generated from elements of Σ. Note that Σ^* contains the *empty string* ε, that is, the string of length zero. A string t is a *factor* of a string w, denoted $t \sqsubseteq w$, if there exists the (potentially empty) strings u and v such that $w = utv$. For some $w \in \Sigma^*$, we use $|w|$ to denote its length. A relation is a set of tuples each of which has the same number of components. The number of components in each tuple of a relation is called the *arity* of the relation.

B. Defining FC

Let X be an infinite alphabet of variable where Σ and X do not share any elements. A *substitution* is a morphism[4] $\sigma : (\Sigma \cup X)^* \to \Sigma^*$ where $\sigma(\mathsf{a}) = \mathsf{a}$, for all $\mathsf{a} \in \Sigma$. In other words, a substitution assigns each variable to a string from Σ^*.

We distinguish a variable $\mathfrak{s} \in X$ called the *universe variable* to represent some input string. We say that a substitution σ is $\mathfrak{s}$-*safe* if $\sigma(x) \sqsubseteq \sigma(\mathfrak{s})$ for all $x \in X$. In other words, σ is $\mathfrak{s}$-safe if it maps every variable to a factor of $\sigma(\mathfrak{s})$.

Definition 2 (Syntax of FC) *The set* FC *is defined recursively as follows.*

- $(x \doteq \alpha) \in \mathsf{FC}$ *for all* $x \in X$ *and* $\alpha \in (\Sigma \cup X)^*$.
- *If* $\varphi, \psi \in \mathsf{FC}$*, then*
  - $(\varphi \wedge \psi) \in \mathsf{FC}$,
  - $(\varphi \vee \psi) \in \mathsf{FC}$,
  - $\neg\varphi \in \mathsf{FC}$*, and*
  - $\exists x\colon \varphi \in \mathsf{FC}$ *and* $\forall x\colon \varphi \in \mathsf{FC}$ *for all* $x \in X \setminus \{\mathfrak{s}\}$.

If the meaning is clear, we may add or omit parentheses.

Before defining the semantics of FC, we require the following definition: For a substitution σ, a string $u \in \Sigma^*$, and a variable $x \in X$, we write $\sigma_{x \to u}$ for the new substitution:

$$\sigma_{x \to u}(z) := \begin{cases} u, & \text{if } z = x, \\ \sigma(z), & \text{otherwise.} \end{cases}$$

That is, $\sigma_{x \to u}$ maps x to u, and for all other variables, $\sigma_{x \to u}$ is the same as σ.

Next, let us now consider the semantics of FC. Informally, for a $\mathfrak{s}$-safe substitution σ and a formula $\varphi \in \mathsf{FC}$, we write $\sigma \models \varphi$ if σ *satisfies* φ.

Definition 3 (Semantics of FC) *Let* $\sigma\colon (\Sigma \cup X)^* \to \Sigma^*$ *be a* $\mathfrak{s}$*-safe substitution. We define* $\models$ *recursively along the syntactic definition of* FC *as follows:*

- $\sigma \models (x \doteq \alpha)$ *if* $\sigma(x) = \sigma(\alpha)$,
- $\sigma \models (\varphi \wedge \psi)$ *if* $\sigma \models \varphi$ *and* $\sigma \models \psi$,
- $\sigma \models (\varphi \vee \psi)$ *if* $\sigma \models \varphi$ *or* $\sigma \models \psi$,
- $\sigma \models \neg\varphi$ *if* $\sigma \models \varphi$ *does not hold,*
- $\sigma \models \exists x\colon \varphi$ *if* $\sigma_{x \to u} \models \varphi$ *for some* $u \sqsubseteq \sigma(\mathfrak{s})$,
- $\sigma \models \forall x\colon \varphi$ *if* $\sigma_{x \to u} \models \varphi$ *for all* $u \sqsubseteq \sigma(\mathfrak{s})$.

[4]A morphism is a function $h\colon A^* \to B^*$ where $h(xy) = h(x) \cdot h(y)$ for all $x, y \in A^*$.

Informally, we can combine concatenation terms ($x \doteq \alpha$) with conjunction ("*and*"), disjunction ("*or*"), negation ("*not*") and quantified variables ("*for all*" and "*there exists*"). Furthermore, each variable is mapped to a factor of the input string ensuring that we only work with "finite models".

Example 4 *Consider the* FC*-formula*

$$\varphi := \exists x\colon (\mathfrak{s} \doteq \mathsf{a}x) \wedge (\mathfrak{s} \doteq x\mathsf{a}).$$

Let σ *be a substitution, notice that since* σ *does not "change" terminal symbols, we only need to look at how* σ *maps variables to terminal strings. Suppose* $\sigma(\mathfrak{s}) = \mathsf{aaa}$ *and we wish to know whether* $\sigma \models \varphi$*. From the definition of the semantics of* FC*, we know that* $\sigma \models \varphi$ *if there exists some* $u \sqsubseteq \mathsf{aaa}$ *such that* $\sigma_{x \to u} \models (\mathfrak{s} \doteq \mathsf{a}x) \wedge (\mathfrak{s} \doteq x\mathsf{a})$*. Consider* $u = \mathsf{aa}$*:*

- $\sigma_{x \to \mathsf{aa}}(\mathfrak{s}) = \sigma_{x \to \mathsf{aa}}(\mathsf{a}x) = \mathsf{aaa}$*, and*
- $\sigma_{x \to \mathsf{aa}}(\mathfrak{s}) = \sigma_{x \to \mathsf{aa}}(x\mathsf{a}) = \mathsf{aaa}$.

Therefore, $\sigma \models \varphi$ *does indeed hold.*

In order to be fully rigorous, the formal definitions of FC are somewhat technical. However, assuming one has some familiarity with first-order logic, FC-formulas could be considered a natural and concise way to reason about strings.

C. Other Logics

FC uses the most natural operation on strings: concatenation. This allows us to treat strings as strings, as opposed to in other logics on strings such as monadic second order logic (MSO) over a linear order, which treat strings as intervals of positions. The authors believe that this makes FC-formulas more natural, and thus easier to write and to interpret.

Example 5 *In MSO, strings are encoded as a sequence of positions which are given terminal symbols by* symbol predicates*. For example, the string* ababa *is encoded by two predicates* $R_{\mathsf{a}} = \{1, 3, 5\}$ *and* $R_{\mathsf{b}} = \{2, 4\}$*. That is,*

1	*2*	*3*	*4*	*5*
a	b	a	b	a.

Then, MSO is a particular logic over the symbol predicates and a total order relation $<$*. For example, if we wanted to write the specification "*the string contains the factor ab*" in MSO, we would write something like:*

$$\exists x_1, x_2\colon \big(R_{\mathsf{a}}(x_1) \wedge R_{\mathsf{b}}(x_2) \wedge \neg\exists x\colon (x_1 < x \wedge x < x_2)\big).$$

Intuitively, this expresses "there is a position x_1 *that has a letter* a *and a position* x_2 *that has a letter* b*, and there is no position* x *between them".*

In contrast to this, FC *allows us to write* $\exists x\colon x \doteq \mathsf{ab}$*, which states "there is a factor* x *that is* ab*".*

Furthermore, we can compare factors of unbounded length in FC; this not possible in MSO (see [1] for a comparison between FC and MSO). Moreover, we can define the whole logic with lightweight definitions.

III. A TOUR THROUGH THE LANDSCAPE OF FC

Logics similar to FC have been considered from a theoretical point of view [13], and (more relevant to this article), have been considered from a security point of view [4]. The latter taking the form of *constraint satisfaction* – that is, whether a combination (often given in a logical form) of string constraints is satisfiable. For example, if we encode an *attack pattern* as a logic formula φ, and encode some input handling code as a logic formula ψ, then $\varphi \wedge \psi$ is satisfiable whenever the input handling code is susceptible to the attack pattern (we refer to [4] for more details).

One of the main issues with this approach is the fact that satisfiability is often computationally hard [5] or even undecidable [14].

On the other hand, FC has a meaningful distinction between *model checking* (does a formula hold for a specific input string) and *satisfiability* (does a formula hold for any string). Furthermore, due to the fact that variables range over the factors of some finite input string, we have a finite domain which makes model checking decidable [1].

Model Checking for FC*:* Formally, the model checking problem for FC is defined as follows:

- *Input:* An FC-formula φ and a $\mathfrak{s}$-safe substitution σ.
- *Question:* Does $\sigma \models \varphi$?

As a convention, we assume the input substitution for the model checking problem is only given for those variables that appear in the input formula. This is to ensure our input is finite, and only contains the relevant details. In terms of parsing, model checking can be seen as a formally defined *accept/reject* mechanism for recognition.

Algorithmically, we can approach model checking using different "lenses", depending on the structure of the formulas, and these can have both *top-down* and *bottom-up* approaches. Let us first consider a naive top-down approach: This can be seen as a recursive algorithm, where if our formula is $\exists x : \varphi$ (or $\forall x : \varphi$), then we loop over all factors check whether replacing x by one (or all resp.) factor(s) results in φ being satisfied. We can recurse down in a straightforward way until we reach the atomic formulas (at which point we have values for each of the variables).

A bottom-up approach would be to start with the atomic formulas, and "build" a relation of satisfying substitutions for each subformula. For example, if we have a relation of satisfying substitutions for φ and for ψ, then taking the union of them builds a relation of satisfying substitution for $\varphi \vee \psi$.

Considering these approaches, it is clear that model checking for FC is decidable, however, it is intractable:

Theorem 6 (Theorem 4.1 of [1]) *Model Checking for* FC *is* PSPACE*-complete.*

A related (but distinct) problem to model checking is *enumeration*:

- *Input:* An FC-formula φ and a string $w \in \Sigma^*$.
- *Output:* All $\mathfrak{s}$-safe substitutions σ such that $\sigma(\mathfrak{s}) = w$ and $\sigma \models \varphi$.

Again, as a convention, we assume the output substitutions σ are only defined for those variables that appear in φ. Enumeration outputs a relation over factors of the input text and so can be seen as a way to "extract" information. In terms of parsing, we break the whole string into more manageable components (as is common with parsers).

Intuitively, model checking is the decision problem variant of enumeration. In other words, since model checking is computationally hard, we cannot hope for efficient enumeration (in general). Therefore, we need to look at fragments of the full logic. The field of Database Theory is rich with fragments of first-order logic which have tractable model checking and enumeration (see [15], [16] for starters). Many such fragments come with a canonical (enumeration/model checking) algorithm. Thus, not only is FC (and its variants) a declarative logic, but it comes with a host of algorithms for specific fragments.

One of way to make model checking for first-order logic tractable is to limit the number of free variables (that is, an occurrence of a variable that is not quantified). We say that $\varphi \in$ FC has *bounded width* if the maximum number of free variables in any subformula is bounded by some constant.

Theorem 7 (Theorem 4.2 of [1]) *If $\varphi \in$ FC has bounded width, then model checking can be done in polynomial time. In fact, if $n = |\sigma(\mathfrak{s})|$, and m is the size*[5] *of φ, then deciding $\sigma \models \varphi$ can be solved in $O(kmn^{2k})$ where k is the width of φ.*

This is the same for general first-order logic (see, e.g., [15]). For FC, we can "materialize" a table for each atomic formula ($x \doteq \alpha$) – that is, enumerate all satisfying substitutions – and then apply a bottom-up algorithm for first-order logic as usual. Due to the fact that the width is bounded, materializing the atomic formulas is not too expensive. Bounding a parameter known as *treewidth* gives us bounded width (see [1]).

Another approach to getting tractable model checking is to bound the *quantifier rank* (the nesting depth of quantifiers) by a constant. Then, using a naive top-down approach, we limit the number of "nested loops" that are required. This again results in a fragment of FC with polynomial-time model checking.

While these approaches are enough to get polynomial-time model checking, they still may not be considered "efficient". One way to get more efficient algorithms is to limit the number of factors that the formula needs to consider. For example, by using constraints.

FC with Constraints: By treating strings as strings, we can easily add arbitrary constraints. That is, instead of only have terms of the form $x \doteq \alpha$ as atomic formulas, we allow $x \mathrel{\dot\in} R$ where R is any language representation (e.g., a context-free grammar, regular expression, etc.) This allows us to express relations and languages that are not expressible with FC without constraints (see [8]). Furthermore, as long as the constraints can be evaluated efficiently, the addition of such constraints does not affect any complexity results.

[5]Using any reasonable encoding.

Most commonly, we look at *regular constraints*, which are additional atoms of the form $x \mathrel{\dot\in} \gamma$ for a variable x and a regular expression γ. For a $\mathfrak{s}$-safe substitution σ, we have $\sigma \models x \mathrel{\dot\in} \gamma$ if $\sigma(x) \in \mathcal{L}(\gamma)$. That is, $x \mathrel{\dot\in} \gamma$ constrains x to be replaced by a string that belongs to the language of γ. We use FC[REG] to denote FC extended with regular constraints. FC[REG] is strictly more expressive than FC, since FC cannot express all the regular languages. Moreover, FC[REG] captures the expressive power of generalized core spanners, a class of the popular information extraction framework of document spanners (see [1], [17]).

Analogously, we may consider constraints with higher arities. For example, we could add a new atomic formula $\mathsf{len}(x, y)$ to FC where, for a $\mathfrak{s}$-safe substitution σ, we have that $\sigma \models \mathsf{len}(x, y)$ whenever the length of $\sigma(x)$ is equal to the length of $\sigma(y)$.

We can also consider *constrained quantifiers*, constraints of the form $\exists x \mathrel{\dot\in} R\colon \varphi$ (or $\forall x \mathrel{\dot\in} R\colon \varphi$) for a variable x and a unary relation symbol R. We have $\sigma \models \exists x \mathrel{\dot\in} R\colon \varphi$ if $\sigma_{x \mapsto u} \models \varphi$ for some $u \sqsubseteq \sigma(\mathfrak{s})$ where $u \in R$, and $\sigma \models \forall x \mathrel{\dot\in} R\colon \varphi$ if $\sigma_{x \mapsto u} \models \varphi$ for all $u \sqsubseteq \sigma(\mathfrak{s})$ where $u \in R$.

Example 8 *For this example, we consider a simple Delimiter-Separated Value (DSV) file specification. Each* value *in our file is an alphanumeric string, a* record *is a string of values separated by a special # symbol. A DSV file is a string of records separated by †. For example:*

$$\text{“}\mathsf{FC}\#\mathsf{Logic}\dagger\,\mathsf{DFA}\#\mathsf{Automata}\dagger\text{”}$$

is a DSV file in our specification. If we assume # is a comma, and † is an “end of line” character, then our DSV file is a CSV file.

Let A be the alphabet of alphanumeric characters, and let $\Sigma := A \cup \{\dagger, \#\}$. We can extract a unary relation of records with the following FC[REG]*-formula:*

$$\varphi_{\mathsf{rec}}(x) := \exists p, s\colon \big((\mathfrak{s} \doteq pxs) \wedge (p \mathrel{\dot\in} \varepsilon \cup \Sigma^*\dagger) \wedge (x \mathrel{\dot\in} (A^*\#)^*A^*\dagger)\big).$$

For intuition, we split our whole string $\mathfrak{s}$ into some prefix p, some content x, and some suffix s. We say that either p is the empty string (which implies x is the first record), or ends with †. Then, we say that x is made up of a string of alphanumeric letters, separated by #, and ends with †.

Now, suppose we wished to check if there is a repeating record. This can be done with the following FC[REG]*-formula $\exists y, x, z\colon \big((y \doteq xzx) \wedge \varphi_{\mathsf{rec}}(x)\big)$. That is, there does exist a record that appears in two places (as a prefix and suffix of y).*

Alternatively, assume we have some constraint Rec *such that $\sigma \models x \mathrel{\dot\in} \mathsf{Rec}$ if $\sigma(x)$ is a record (i.e., it replaces $\varphi_{\mathsf{rec}}(x)$). Then, we can write the previous formula with a constrained quantifier: $\exists x \mathrel{\dot\in} \mathsf{Rec}\colon \big(\exists y, z\colon y \doteq xzx\big)$. Thus, instead of naively enumerating all factors for y, x, and z; and then determining whether $(y \doteq xzx) \wedge \varphi_{\mathsf{rec}}(x)$ holds, we instead enumerate all records and check whether $y \doteq xzx$ holds (for some y and z).*

Thus, using constrained quantifiers, leads to potential optimizations. Instead of considering all the factors (quadratically many in the length of $\sigma(\mathfrak{s})$), we need only need to consider those factors that belong to R.

The technique illustrated in Example 8 of splitting the input string into components, and testing constraints on those components is applicable in a variety of different situations. For example, an HTTP request is split into the request line, optional headers, an empty line, and then the body. Therefore, one could adapt Example 8 to deal with the HTTP protocol syntax.

FC-CQ: Conjunctive queries are a central topic in database theory (see [18]). They can be thought of as a fragment of first-order logic that use only conjunction and existential quantification. We denote FC-CQ as the conjunctive query fragment of FC. Even for this strict restriction of FC, model checking is computationally hard:

Theorem 9 (Theorem 4 of [19]) *Model checking for* FC-CQ *is* NP*-complete, and remains* NP*-hard even if the input string is of length one.*

However, in the database setting, a fragment known as *acyclic conjunctive queries* allows for polynomial-time model checking. A conjunctive query is acyclic if there exists a so-called *join tree* for that conjunctive query. A join tree is a tree-based data structure where each node of the tree is some atomic formula that appears in the conjunctive query. Furthermore, without going into too many details, each variable must only appear in one connected subtree. Once one has a join tree for a conjunctive query, they can apply *Yannakakis' algorithm* for efficient model checking (see [16]).

Unfortunately, we cannot immediately apply Yannakakis' algorithm for FC-CQ. This is because each atomic formula $x \doteq \alpha$ may have an exponential number of tuples. Therefore, Freydenberger and Thompson [9] treat each concatenation term $x \doteq \alpha$ as shorthand for a conjunction of *binary concatenation terms* $x \doteq yz$ where x, y, and z are variables. For example, $x \doteq y_1y_2y_1y_1$ could be considered shorthand for $(x \doteq zz) \wedge (z \doteq y_1y_2)$.

Freydenberger and Thompson [9] gave an algorithm which is given an FC-CQ, and (in polynomial time) rewrites it into an equivalent acyclic FC-CQ only using binary concatenation terms, or determines that this cannot be done for the given input and the particular type of rewriting[6]. Once we have an acyclic FC-CQ using binary concatenation terms, we can materialize the relations for each atomic formula in polynomial time, and then using Yannakakis' algorithm [16] as usual.

FC-Datalog: Extending FC-CQ with recursion gives us FC-Datalog, a variant of the relational query language Datalog for querying strings.

[6]The authors believe that the more general problem of *given an* FC-CQ*, does there exist an equivalent acyclic* FC-CQ *using only binary concatenation terms* is likely computationally hard or even undecidable.

Example 10 *Let P be the* FC-Datalog *program defined in Example 1. Here we give an example top-down execution of P on the string* abab.

1) *As it is the only rule that contains the output symbol, we first apply the rule* $\text{Ans}() \leftarrow \mathfrak{s} \doteq yy,\ R(y)$. *Since* $\sigma(\mathfrak{s}) = \mathsf{abab}$, *we have* $\sigma(y) = \mathsf{ab}$ *which is passed into the relation R.*
2) *Next, we apply* $R(x) \leftarrow x \doteq y\mathsf{b},\ R(y)$. *As we have* $\sigma(x) = \mathsf{ab}$, *then* $\sigma(y) = \mathsf{a}$. *We then recurse on this value of* $\sigma(y)$.
3) *Now, we apply* $R(x) \leftarrow x \doteq y\mathsf{a},\ R(y)$. *We then recurse on* $\sigma(y) = \varepsilon$.
4) *We then can apply* $R(x) \leftarrow x \doteq \varepsilon$. *As this holds we accept.*

Adding recursion allows us to exactly express the complexity class P. We say a logic captures a complexity class $\mathbb{C}$ if the class of languages it expresses is exactly $\mathbb{C}$. Note that if a logic captures a complexity class $\mathbb{C}$, its data complexity is $\mathbb{C}$ (that is, the model checking problem where we assume the logic formula is fixed, and some string is the input).

Theorem 11 (Theorem 4.11 of [1]) FC-Datalog *captures* P.

As P is not considered efficient for data complexity, [10] identifies more efficient fragments of FC-Datalog. In an FC-Datalog rule, we call the part to the left of the $\leftarrow$ the rule's *head* and the part to the right of the $\leftarrow$ the rule's *body*. Informally, *Linear* FC-Datalog restricts the bodies of rules such that they may only contain one atom with a relation symbol that is mutually recursive with the head relation symbol. *One Letter Lookahead* FC-Datalog restricts the string equations to only check one letter at a time, and this allows us to efficiently check for determinism. We refer to [10] for specific details on these FC-Datalog fragments.

Theorem 12 (Theorems 3.4 and 3.17 of [10]) *Linear* FC-Datalog *captures* NLOGSPACE. *Deterministic One Letter Lookahead* FC-Datalog *captures* LOGSPACE.

In [10], deterministic One Letter Lookahead is abbreviated to DOLLA FC-Datalog. In fact, [10] introduces a whole range of fragments that capture LOGSPACE, and in particular one called DOLLA+ FC-Datalog. The example program given in Example 1 is a DOLLA+ FC-Datalog program.

Furthermore, as LOGSPACE is closed under complement, we can also add *stratified negation* without affecting the complexity. In [10], it was also shown that we can view such FC-Datalog programs as generalised two-way multiheaded finite automata, a more flexible model that permits performing nonregular string computations in the transitions on top of the usual automata functionality.

When parsing in practice, the parse tree itself is also often useful. As in the relational setting, we could alternatively define the semantics of FC-Datalog in the *proof theoretic way* using so-called *proof trees*. As shown in Section 12.4 of [18], Datalog proof trees have a strong connection to derivation trees in context-free languages. As we are working on strings, we can therefore obtain parse trees for FC-Datalog programs.

Static Analysis: Unfortunately, many static analysis problems such as satisfiability and equivalence are difficult for FC, and even for restrictive fragments like FC-CQ (see [1], [19] for details). Therefore, an important direction for future research is investigating fragments where these problems become tractable. On the other hand, we will discuss in Section VI how we can see FC and its extensions as a framework for combining parsers, and use this to obtain *weak equivalence*.

IV. FC as a Replacement for Regex

Bell, Day, and Freydenberger [10] show how fragments of FC-Datalog can simulate classes of regex—that is, regular expressions with a backreference operator that matches a repetition of a previously matched string. In particular, they show how so-called deterministic regex can be expressed in DOLLA+ FC-Datalog. Deterministic regex can express nonregular languages, and have a membership problem that is almost as efficient as deterministic regular expressions (see [20]).

As found by [21], regex are commonly seen as difficult to write and interpret. Furthermore, the difficulty of using regex can cause developers to run into two main issues: portability, where reused regex or regex composed from existing expressions have unexpected behavior, and security issues, such as Regular expression Denial of Service (ReDoS), as the worst-case time complexity in most regex engines can be exponential. Both of these problems are often overlooked (see [21]). The authors argue that FC and FC-Datalog could be used as an alternative to regex as due to their declarative nature, FC-formulas are simpler and thus easier to use. When reusing FC-formulas, we do so explicitly. The compositionality of FC means we have independent components that can be reused, verified and tested. We can then modify FC-formulas more simply. For example, adding a further condition φ_2 to an existing formula φ_1 can be done with $\varphi_3 \coloneqq \varphi_1 \wedge \varphi_2$. This is not so simple in regex. Furthermore, in terms of security risks, by being able to test the components that we "plug" together, we can thus test the time complexity incrementally, which is much more difficult with regex.

Example 13 *For a binary alphabet* $\{\mathsf{a}, \mathsf{b}\}$, *we can express all strings that do not have* abab *as a factor with the regex* $\mathsf{a}^*(\mathsf{bb}^*\mathsf{aaa}^*)^*\mathsf{b}^*(\mathsf{a} \mid \varepsilon)$. *As the alphabet grows, so does the regex. For arbitrary alphabets, we can express this property with the* FC*-formula* $\neg\exists x\colon x \doteq \mathsf{abab}$.

V. Language-Theoretic Security and FC

A common attack is to exploit the inadequate input validation of parsers, which should filter out malicious input and transform the valid input to an appropriate representation for the rest of the program. Unfortunately, the set of valid inputs is often not explicitly specified, and so malicious input may be validated by the parser, rendering it insecure (see e.g. [3]). LangSec (Language-Theoretic Security) is a paradigm for defending against these attacks that proposes defining the set

of valid inputs as a formal language and building a recognizer for this input language to act as the parser. As in [3], the core principles of LangSec can be expressed as the following:

1) *Full recognition before processing*: The valid input should be defined as a formal language, and the program should decide the input before performing any non-parsing computation.
2) *Principle of least expressiveness*: The grammar used for defining the valid inputs should be only as high in the Chomsky hierarchy as necessary and not any higher.
3) *Principle of parser equivalence*: Any two different parsers that decide the same language should be functionally equivalent.

The authors believe that a declarative model (such as FC and its extensions) has clear advantages for verification, compared to a sequential or computational machine such as a grammar or automaton. By formally defining a language based on the desired outcome, we can simplify the writing process and mitigate against unintended consequences that arise when defining a specific implementation. Additionally, by presenting the unsatisfied subformulas to users, FC could also provide less complex error messaging than for grammar-based models. Also, a declarative model allows for more concise formulas than a computational model, which often can be easier to write and interpret. More precisely, the size blow up from an FC-formula to an equivalent regular expression is not bounded by any recursive function (Theorem 4.8 of [1]), even for the conjunctive query fragment (see Theorem 8 of [19]).

Example 14 *Example 13 shows an* FC*-formula for expressing that a string does not contain the factor* bab*. Like for regex in Example 13, it is more complex to express this using a grammar. For a binary alphabet* $\{\mathsf{a}, \mathsf{b}\}$ *we can use:*

$$\begin{aligned} S &\to \mathsf{a}S; & A &\to \mathsf{a}B; \\ S &\to \mathsf{b}A; & A &\to \mathsf{b}A; \\ S &\to \varepsilon; & B &\to \mathsf{a}S; \\ A &\to \varepsilon; & B &\to \varepsilon. \end{aligned}$$

Again, as the alphabet grows, so does the grammar.

FC and its extensions align directly with the full recognition before processing principle; in FC and its extensions we define formal languages and we have optimizations for model checking, the problem that is exactly deciding the validity of the input.

Furthermore, we are not limited to "pure FC", as we have a whole framework of natural fragments and extensions surrounding it (as we discussed in Section III). We can therefore also apply the principle of least expressiveness in choosing which fragment or extension to use.

The principle of parser equivalence for FC and its extensions is more difficult. From [22], the equivalence problem for non-deterministic context-free grammars is undecidable, however, [23] showed that this problem is decidable for deterministic context-free languages. It is therefore argued by [2], [3], [24] that ideally parsers should not be more expressive than deterministic context free, the class of languages accepted by the class of deterministic pushdown automata, as then can be checked if two parsers are functionally equivalent. Unfortunately, the equivalence problem is undecidable for FC (see [1]), even for the conjunctive query fragment [19].

Therefore, an important area for future work is to consider fragments that have decidable equivalence. However, there are still cases where more complex data formats are required. Anantharaman [3] states that in such cases, the solution to this problem is visually comparing grammars. The authors argue that visually comparing declarative models such as FC and its extensions is significantly simpler than computational models such as grammars or automata. Furthermore, as we shall discuss in Section VI, we can use FC and its extensions as a unifying framework for combining parsers. As such, we have a weak equivalence, where we can 'plug-in' relations defined elsewhere (such as by other formulas or from parsing another model), and thus need only decide equivalence of these relations.

VI. Conclusions and future work

Using FC and its extensions, we have a declarative model for parsing, where formulas come with different "lenses" for how they can be evaluated efficiently, depending on the particular fragment being considered. We can then combine these various fragments of FC as well as other parsers into one unifying framework. We can see any formula as defining a relation that we can use as a subformula in other formulas. Furthermore, relation symbols can be placeholders for subroutines, such as verifying a precomputed relation. This allows us to define a formula, from a particular fragment if we want performance guarantees, which can combine relations from a number of different sources. Such a relation could be defined by;

- another FC-formula or subformula,
- an FC-Datalog program (again from a restricted class if we want performance guarantees),
- a parser from another model (such as a context-free grammar),
- an existing relational database,
- a by-hand implementation (for a relation such as equal length that is efficient to verify).

As such, we can see the original formulas as unifying multiple parsers into a single text verifier. The ability to have modular "black box" components that can be swapped, combined and verified is much more natural in FC than for computational models such as automata or grammars. Moreover, from this we gain a weak equivalence, we need only to decide the equivalence of our subrelations, aligning FC more with the LangSec principle of parser equivalence discussed in Section V. The modularity and compositionality of FC and its extensions also allow us to have a further hierarchy of expressivity, which is relevant to the principle of least expressiveness also discussed in Section V.

Along with preprocessing optimizations such as RMQ (range minimum query) data structures and LCP (longest common prefix) arrays, using naive brute force has led to a

promising initial prototype implementation for FC. We are thus currently working on three main directions:

- Investigating for further fragments of FC and its extensions where static analysis problems such as equivalence and satisfiability become tractable.
- Optimizing evaluation algorithms using techniques from database theory. Our approaches here range simply rewriting a formula using standard logical equivalences to creating query plans that express the order in which to execute operations.
- Building a fully-fledged implementation for FC and its extensions, including an implementation of the optimizer discussed in the previous point.

So far, FC has only been studied from a theoretical point of view. Moving forward, we aim to investigate the three research directions from both a theoretical and practical perspective. In addition to general optimizations, there are likely to be application-specific optimizations, offering the ability to tailor the general FC framework for particular use cases.

Acknowledgements

The authors would like to thank the reviewers for their suggestions and feedback. This work was funded by the EPSRC grant EP/T033762/1. For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

Data Availability

No data was analysed, captured or generated for this work.

References

[1] D. D. Freydenberger and L. Peterfreund, "The Theory of Concatenation over Finite Models," in *Proc. ICALP 2021*, 2021, pp. 130:1–130:17. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ICALP.2021.130

[2] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, "Security Applications of Formal Language Theory," *IEEE Syst. J.*, pp. 489–500, 2013. [Online]. Available: https://ieeexplore.ieee.org/document/6553401

[3] P. Anantharaman, "Protecting systems from exploits using language-theoretic security," Ph.D. dissertation, Dartmouth College, USA, 2022. [Online]. Available: https://digitalcommons.dartmouth.edu/dissertations/80

[4] A. W. Lin and P. Barceló, "String solving with word equations and transducers: towards a logic for analysing mutation XSS," in *Proc. POPL 2016*, 2016, pp. 123–136. [Online]. Available: https://dl.acm.org/doi/10.1145/2914770.2837641

[5] W. Plandowski, "Satisfiability of word equations with constants is in PSPACE," *J. ACM*, pp. 483–496, 2004. [Online]. Available: https://dl.acm.org/doi/10.1145/990308.990312

[6] J. Karhumäki, F. Mignosi, and W. Plandowski, "The expressibility of languages and relations by word equations," *J. ACM*, pp. 483–505, 2000. [Online]. Available: https://dl.acm.org/doi/10.1145/337244.337255

[7] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective Search-Space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints," in *Proc. CAV 2015*, 2015, pp. 235–254. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-21690-4_14

[8] S. M. Thompson and D. D. Freydenberger, "Generalized core spanner inexpressibility via Ehrenfeucht-Fraïssé games for FC," *Proc. ACM Manag. Data*, pp. 1–18, 2024. [Online]. Available: https://doi.org/10.1145/3651143

[9] D. D. Freydenberger and S. M. Thompson, "Splitting Spanner Atoms: A Tool for Acyclic Core Spanners," in *Proc. ICDT 2022*, 2022, pp. 6:1–6:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.ICDT.2022.10

[10] O. M. Bell, J. D. Day, and D. D. Freydenberger, "FC-Datalog as a Framework for Efficient String Querying," in *Proc. ICDT 2025*, 2025, pp. 29:1–29:18. [Online]. Available: https://doi.org/10.4230/LIPIcs.ICDT.2025.29

[11] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Trans. Knowl. Data Eng.*, pp. 146–166, 1989. [Online]. Available: https://ieeexplore.ieee.org/document/43410

[12] H. Straubing, *Finite automata, formal logic, and circuit complexity*. Springer Science & Business Media, 2012.

[13] W. V. Quine, "Concatenation as a Basis for Arithmetic," *J. Symb. Log.*, pp. 105–114, 1946. [Online]. Available: https://www.jstor.org/stable/2268308?seq=1

[14] V. G. Durnev, "Undecidability of the positive $\forall\exists^3$-theory of a free semigroup," *Sib. Math. J.*, pp. 917–929, 1995. [Online]. Available: https://doi.org/10.1007/BF02112533

[15] I. Adler and M. Weyer, "Tree-Width for First Order Formulae," in *Proc. CSL 2009*, 2009, pp. 71–85. [Online]. Available: https://doi.org/10.1007/978-3-642-04027-6_8

[16] M. Yannakakis, "Algorithms for acyclic database schemes," in *Proc. VLDB 1981*, 1981, pp. 82–94. [Online]. Available: https://dl.acm.org/doi/10.5555/1286831.1286840

[17] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren, "Document Spanners: A Formal Approach to Information Extraction," *J. ACM*, 2015. [Online]. Available: https://doi.org/10.1145/2699442

[18] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995. [Online]. Available: http://webdam.inria.fr/Alice/

[19] S. M. Thompson and D. D. Freydenberger, "Languages Generated by Conjunctive Query Fragments of FC[REG]," *Theory Comput. Syst.*, pp. 1–43, 2024. [Online]. Available: https://doi.org/10.1007/s00224-024-10198-4

[20] D. D. Freydenberger and M. L. Schmid, "Deterministic regular expressions with back-references," *J. Comput. Syst. Sci.*, vol. 105, pp. 1–39, 2019. [Online]. Available: https://doi.org/10.1016/j.jcss.2019.04.001

[21] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions," in *Proc. ASE 2019*, 2019, pp. 415–426. [Online]. Available: https://ieeexplore.ieee.org/document/8952499

[22] J. E. Hopcroft, "On the equivalence and containment problems for context-free languages," *Math. Syst. Theory*, pp. 119–124, 1969. [Online]. Available: https://doi.org/10.1007/BF01746517

[23] G. Sénizergues, "The Equivalence Problem for Deterministic Pushdown Automata is Decidable," in *Proc. ICALP 1997*, 1997. [Online]. Available: https://api.semanticscholar.org/CorpusID:33023429

[24] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them," in *SecDev 2016*, 2016, pp. 45–52. [Online]. Available: https://ieeexplore.ieee.org/document/7839788