Static Inference of Regular Grammars for Ad Hoc Parsers

MICHAEL SCHRÖDER, TU Wien, Austria
JÜRGEN CITO, TU Wien, Austria

Parsing—the process of structuring a linear representation according to a given grammar—is a fundamental activity in software engineering. While formal language theory has provided theoretical foundations for parsing, the most common kind of parsers used in practice are written *ad hoc*. They use common string operations without explicitly defining an input grammar. These ad hoc parsers are often intertwined with application logic and can result in subtle semantic bugs. Grammars, which are complete formal descriptions of input languages, can enhance program comprehension, facilitate testing and debugging, and provide formal guarantees for parsing code. But writing grammars—e.g., in the form of regular expressions—can be tedious and error-prone. Inspired by the success of type inference in programming languages, we propose a general approach for static inference of regular input string grammars from unannotated ad hoc parser source code. We use refinement type inference to synthesize logical and string constraints that represent regular parsing operations, which we then interpret with an abstract semantics into regular expressions. Our contributions include a core calculus λ_Σ for representing ad hoc parsers, a formulation of (regular) grammar inference as refinement inference, an abstract interpretation framework for solving string refinement variables, and a set of abstract domains for efficiently representing the constraints encountered during regular ad hoc parsing. We implement our approach in the Panini system and evaluate its efficacy on a benchmark of 204 Python ad hoc parsers. Compared with state-of-the-art approaches, Panini produces better grammars (100 % precision, 93 % average recall) in less time (0.82 ± 2.85 s) without prior knowledge of the input space.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → **Type theory**; **Abstraction**; **Regular languages**.

Additional Key Words and Phrases: ad hoc parsers, grammars, refinement types, abstract interpretation

ACM Reference Format:
Michael Schröder and Jürgen Cito. 2025. Static Inference of Regular Grammars for Ad Hoc Parsers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 276 (October 2025), 31 pages. https://doi.org/10.1145/3763054

1 Introduction

Parsing is one of the fundamental activities in software engineering. It is an activity so common that pretty much every program performs some kind of parsing at one point or another. Yet in every-day software engineering, only a small minority of programs, mainly compilers and some protocol implementations, make use of formal grammars to document their input languages or use formalized parsing techniques such as combinator frameworks [Leijen and Meijer 2001] or parser generators [Johnson and Sethi 1990; Parr and Quong 1995; Warth and Piumarta 2007]. The vast majority of parsing code in software today is *ad hoc*.

Ad hoc parsers are pieces of code that combine common string operations like indexing `s[i]`, substring search `s.find(t)`, or whitespace trimming `s.strip()` to effectively perform parsing. A programmer manipulating strings in an ad hoc fashion would probably not even think about the fact that they are actually writing a parser. These string-manipulating programs can be found in

Authors' Contact Information: Michael Schröder, TU Wien, Vienna, Austria, michael.schroeder@tuwien.ac.at; Jürgen Cito, TU Wien, Vienna, Austria, juergen.cito@tuwien.ac.at.

ACM 2475-1421/2025/10-ART276
https://doi.org/10.1145/3763054

functions handling command-line arguments, reading configuration files, or as part of any number of minor programming tasks involving strings, often deeply entangled with application logic—a phenomenon known as *shotgun parsing* [Momot et al. 2016]. They have also been shown to produce subtle and difficult to identify semantic bugs [Eghbali and Pradel 2020; Kapugama et al. 2022].

A *grammar* is a complete formal description of all values an input string may assume. It can elucidate the corresponding parsing code, revealing otherwise hidden features and potentially subtle bugs or security issues. By focusing on *data* rather than code, grammars provide a high-level perspective, allowing programmers to grasp an input language directly, without being distracted by the mechanics of the parsing process and the intricacies of imperative string manipulation. Augmenting regular documentation with formal grammars can increase program comprehension by providing alternative representations for a programming task [Fitter and Green 1979; Gilmore and Green 1984]. Because a grammar is also a *generating device*, it is possible to construct any sentence of its language in a finite number of steps—manually or in an automated fashion. Being able to reliably generate concrete examples of possible inputs is invaluable during testing and debugging. *Grammar-based fuzzing* is an automated testing method that uses input grammars to generate syntactically valid fuzz inputs that can penetrate into deep program states, passing through syntactic checks (i.e., ad hoc parsers) to uncover semantic bugs [Hodován et al. 2018; Holler et al. 2012; Zeller et al. 2024].

But despite providing all these benefits, hardly anyone ever bothers to write down a grammar, even for more complex ad hoc parsers. Grammars share the same fate as most other forms of specification: they are tedious to write, hard to get right, and seem hardly worth the trouble—especially for such small pieces of code like ad hoc parsers.

The only type of grammar that people actually routinely write down are *regular expressions* [Thompson 1968], probably the biggest and most widely known success story of applied formal language theory. However, in practical use they are often embedded within bigger pieces of ad hoc parsing code and thus usually only describe part of the actual input grammar of an ad hoc parser.

But there is another form of specification that we can draw inspiration from: *types*. Formal grammars are similar to types, in that an ad hoc parser without a grammar is very much like a function without a type signature—it might still work, but you will not have any guarantees about it before actually running the program. Types have one significant advantage over grammars, however: most type systems offer a form of *type inference*, allowing programmers to omit type annotations because they can be automatically recovered from the surrounding context. If we could infer grammars like we can infer types, we would reap all the rewards of having a complete specification of our program's input language, without burdening the programmer with the tedious task of writing a grammar by hand—we would have "grammars for free" [Schröder and Cito 2022].

Our Contribution. In this paper, we present a general approach for static inference of regular string grammars from unannotated ad hoc parser source code. We pose the problem of (regular) grammar inference as a sub-goal of refinement type inference. During type inference, we synthesize logical constraints that declaratively represent the parsing operations performed on the input string. We then interpret this complex first-order formula using an abstract semantics, in order to find a minimal sub-constraint to use as an input string refinement, rendered as a regular expression.

In brief, the contributions of our work are:

- A core calculus λ_Σ for representing ad hoc parsers.
- A formulation of (regular) grammar inference as refinement inference.
- An abstract interpretation framework for solving string refinement variables.
- A set of abstract domains related to regular ad hoc parsing.
- An implementation of our approach in the Panini system.

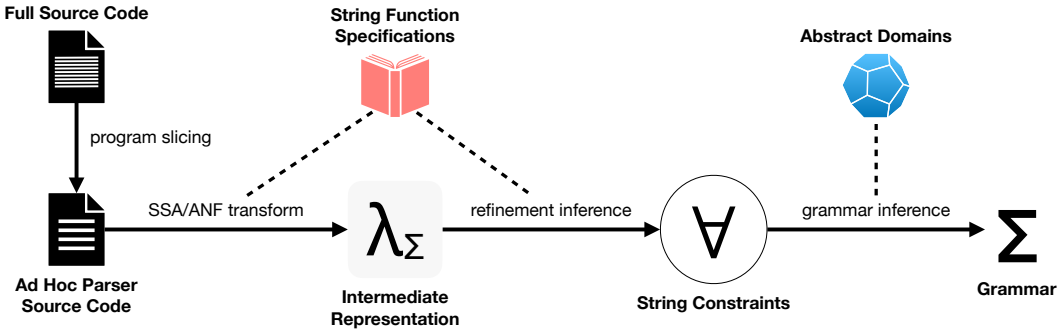

Fig. 1. The complete Panini system.

2 Overview

Figure 1 presents a schematic overview of Panini,[1] our end-to-end grammar inference system. At the center of our approach is λ_Σ, a language-agnostic intermediate representation for ad hoc parsers. It is powerful enough to represent all relevant parsing operations and simple enough to enable straight-forward refinement type inference. The refinement type system of λ_Σ allows us to synthesize constraints over a parser's input string—i.e., it allows us to infer a parser's grammar.

The Panini system can be separated into a front end and a back end. In the front end, ad hoc parsers written in a general-purpose programming language, e.g., Python, are translated into λ_Σ programs. In the back end, those λ_Σ programs are statically analyzed and their input string constraints extracted. *This paper is about the back end.* In short order, we will describe the λ_Σ calculus, its refinement type system, our grammar inference algorithm, and the underlying abstract domains. To situate this work, we first want to briefly sketch the front-end process, and then step through a number of short examples that illustrate our approach.

2.1 The Front End: From Source to λ_Σ

After locating an ad hoc parser slice, it is first translated into static single assignment (SSA) form [Braun et al. 2013] and then, via an SSA-to-ANF transformation [Chakravarty et al. 2004], into a Panini program. This requires a library of string function specifications that map the source language's string operations to equivalent λ_Σ functions. Note that it is not necessary to have actual λ_Σ *implementations* of these operations. We only need axiomatic specifications—in the form of type signatures—to capture those properties of the original functions that are necessary to synthesize string grammars. Table 1 gives (simplified) examples of such axioms for certain Python functions.

The front-end transformations (parser slicing, source-to-λ_Σ translation) and accompanying axiomatic string function specifications need to be defined and implemented (and proven correct) only once per source programming language. For the remainder of this paper, we will assume a Python-to-λ_Σ transformation and a library of Python string function specifications.

2.2 The Back End: From λ_Σ to Grammar

λ_Σ is a simple λ-calculus with a refinement type system in the style of *Liquid Types* [Rondon et al. 2008; Vazou et al. 2014]. Refinement types allow us to extend base types with logical constraints. This is useful to precisely describe subsets of values, as well as track complex relationships between values, all on the type level. For example, the type of natural numbers can be defined as a subset of the integers, $\mathbb{N} = \{\nu : \mathbb{Z} \mid \nu \geq 0\}$, and we can give a precise definition of the length function on

[1] Named in honor of the Sanskrit grammarian Pāṇini [Bhate and Kak 1991–1992], as well as the delicious Italian sandwiches.

Table 1. Python operations as axiomatic λ_Σ specifications.

Python	λ_Σ specification	
`assert b`	assert : $\{b : \mathbb{B} \mid b\} \to \mathbb{1}$	assertion
`a == b`	eq : $(a : \mathbb{Z}) \to (b : \mathbb{Z}) \to \{c : \mathbb{B} \mid c \Leftrightarrow a = b\}$	integer equality
`len(s)`	length : $(s : \mathbb{S}) \to \{n : \mathbb{N} \mid n = \lvert s\rvert\}$	string length
`s[i]`	charAt : $(s : \mathbb{S}) \to \{i : \mathbb{N} \mid i < \lvert s\rvert\} \to \{c : \mathbb{Ch} \mid c = s[i]\}$	character-at-index
`s == t`	eqChar : $(s : \mathbb{Ch}) \to (t : \mathbb{Ch}) \to \{b : \mathbb{B} \mid b \Leftrightarrow s = t\}$	character equality

strings using a dependent function type and the string length operator $\lvert\square\rvert$ of the refinement logic, $(s : \mathbb{S}) \to \{n : \mathbb{N} \mid n = \lvert s\rvert\}$. Checking a refinement type reduces to proving a so-called *verification condition* (VC), a first-order constraint in the refinement logic generated by the type system. The validity of the VC entails the correctness of the program's given and inferred types [Nelson 1980]. For example, in order to check whether $\{x : \mathbb{Z} \mid x > 42\}$ (the type of all numbers greater than forty-two) is a subtype of $\mathbb{N}$, the VC constraint $\forall x.\ x > 42 \Rightarrow x \geq 0$ has to be verified. For verification to remain practical, the refinement logic is typically chosen to allow for decidable *satisfiability modulo theories* (SMT), which means VCs can be discharged using an off-the-shelf constraint solver such as Z3 [De Moura and Bjørner 2008].

Refinement Inference. To infer a refinement type for any given term, one must find a predicate that describes (at most) all possible values the term could have during any (successful) run of the program. To facilitate this, refinement type systems typically first infer the basic shapes of all types in the program, using standard type inference à la Hindley-Damas-Milner [Damas and Milner 1982; Hindley 1969], with placeholder variables standing in for as-yet-unknown refinement predicates. These placeholder variables—variously called "κ variables" [Cosman and Jhala 2017], "Horn variables" [Jhala and Vazou 2020], or "liquid type variables" [Rondon et al. 2008]—are also present in the VC at this point and prevent it from being discharged (since they are unknown). The type system then tries to find the strongest satisfying assignments for all refinement variables in the VC constraints, in order to both validate the VC and complete the inferred type.

Example 2.1. The simple λ_Σ program **if** true **then** 1 **else** 2 can be inferred to have an incomplete type of shape $\{v : \mathbb{Z} \mid \kappa(v)\}$, where κ is an unknown refinement variable, together with the VC

$$(\text{true} \Rightarrow \forall v.\ v = 1 \Rightarrow \kappa(v)) \wedge (\text{false} \Rightarrow \forall v.\ v = 2 \Rightarrow \kappa(v)).$$

In order to complete the type, we now have to find an assignment for κ. With such a simple constraint, the refinement solver can easily infer the correct assignment $\kappa(v) \mapsto v = 1$, which validates the VC and produces the final type $\{v : \mathbb{Z} \mid v = 1\}$.

Grammar Inference. If a κ variable stands for an input string refinement, we call this a *grammar variable*, because its solution must be some finite description of all strings that are accepted by the program, i.e., a grammar. To be practical, we would like this grammar to be as complete as possible.

Example 2.2. To illustrate, consider the Python expression `assert s[0] == "a"`. Assuming the function specifications from Table 1, we can transform this expression to an equivalent λ_Σ program (below left) with an inferred top-level refinement type of $\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}$ and a VC (below right) that closely matches the program.

$$\begin{array}{l} \boldsymbol{\lambda} s : \mathbb{S}. \\ \quad \textbf{let } x = \text{charAt } s\ 0 \textbf{ in} \\ \quad \textbf{let } p = \text{eqChar } x\ \text{`a'} \textbf{ in} \\ \quad \text{assert } p \end{array} \qquad \begin{array}{l} \forall s.\ \kappa(s) \Rightarrow \\ \quad 0 < \lvert s\rvert \wedge \forall x.\ x = s[0] \Rightarrow \\ \quad\quad \forall p.\ (p \Leftrightarrow x = \text{`a'}) \Rightarrow \\ \quad\quad\quad p \end{array}$$

In order to complete both the VC and the top-level type, we have to find an appropriate assignment for the grammar variable κ. The assignment must take the form of a single-argument function constraining the string s. It is clear that choosing $\kappa(s) \mapsto \text{true}$ is not a valid solution because allowing *any* string for s does not satisfy the VC. On the other hand, choosing $\kappa(s) \mapsto \text{false}$ trivially validates the VC, but it implies that the function could never actually be called, as no string satisfies the predicate false. One possible assignment could be $\kappa(s) \mapsto s = \text{“a”}$, which only allows exactly the string “a” as a value for s. While this validates the VC and produces a correct type in the sense that it ensures the program will never go wrong, it is much too strict: we are disallowing an infinite number of other strings that would just as well fulfill these criteria (e.g., “aa”, “ab”, and so on). The correct assignment is $\kappa(s) \mapsto s[0] = \text{‘a’}$, which ensures that the first character of the string is ‘a’ but leaves the rest of the string unconstrained. This is equivalent to the regular language $\mathtt{a}\Sigma^*$, where Σ is any letter from the input alphabet. Note that the solution is a minimized version of the top-level consequent in the VC.

The key insight that allows us to find suitable assignments for grammar variables in a general manner is that the top-level VC for a parser will always be of the form $\forall s.\ \kappa(s) \Rightarrow \varphi$, where s is the input string and φ is a constraint that precisely captures all parsing operations the program performs on s. By simply taking $\kappa(s) \mapsto \varphi$, the VC becomes a trivially valid tautology and the string refinement captures exactly those inputs that the parser accepts. But clearly this solution is practically useless: we want a succinct predicate in a grammar-like form, but the top-level VC consequent φ is a complex term in first-order logic that is basically identical to the program code itself; it does not lead to any further insight about the parser's actual input language.

However, we can use φ as a starting point and reduce it into a simpler predicate by performing an *abstract interpretation* of φ. Our approach utilizes an abstract semantics of first-order formulas over string constraints in order to soundly eliminate quantifiers and approximate the parser's input language. The precision of this approximation depends on the language complexity of the parser, the ability of the refinement inference system to synthesize program invariants, and the expressiveness of the underlying abstract value domains.

To give an idea of how grammar inference works in practice, we present a brief example.

Example 2.3. The program shown in Fig. 2a is a simple parser written in Python that checks if the first character of the input string is `"a"` and, if so, asserts that the length of the string must be one and thus the string must be exactly `"a"`; otherwise, if the first character is not `"a"`, then the program asserts that the second character must be `"b"`, with no further restrictions on the input string. Note that the Python string indexing operations `s[0]` and `s[1]` are themselves types of assertions, since they will cause the program to crash if the index is out of bounds. This behavior is captured in the λ_Σ equivalent of the source program via its function axioms (Table 1).

Figure 2b shows the parser's top-level typing derivation (§ 3). The refinement inference judgement, applied to the parser function and its axioms, results in an incomplete refinement type containing an unsolved κ variable, and a verification condition of the form $\forall s.\ \kappa(s) \Rightarrow \varphi$, which tells us that this κ variable is indeed a grammar variable. We can see that the VC's consequent φ captures all of the parsers explicit and implicit constraints over its program variables.

Finally, Fig. 2c shows a (possible) step-by-step reduction of φ into a simple quantifier-free predicate using abstract interpretation (see § 4). First, the innermost quantifiers $\forall v_1, \forall n, \forall v_2$, and $\forall y$ are eliminated and their quantified variables replaced by semantically equivalent expressions. Then we eliminate $\forall p_1$, followed by $\forall x$. At this point, there are no more quantified variables and we can fully abstract each occurrence of the only free variable s. Finally, we normalize the quantifier-free predicate into a single abstract string relation $s \in (\mathtt{a} + (\Sigma \backslash \mathtt{a})\mathtt{b}\Sigma^*)$.

```python
def parser(s):
  if s[0] == "a":
    assert len(s) == 1
  else:
    assert s[1] == "b"


>>> parser("")    # IndexError
>>> parser("a")   # ✓
>>> parser("b")   # IndexError
>>> parser("ab")  # AssertionError
>>> parser("bb")  # ✓
```

$$
\begin{array}{l}
\text{parser} = \boldsymbol{\lambda} s : \mathbb{S}. \\
\quad \textbf{let } x = \text{charAt } s\ 0 \textbf{ in} \\
\quad \textbf{let } p_1 = \text{eqChar } x\ \text{`a'} \textbf{ in} \\
\quad \textbf{if } p_1 \textbf{ then} \\
\quad\quad \textbf{let } n = \text{length } s \textbf{ in} \\
\quad\quad \textbf{let } p_2 = \text{eq } n\ 1 \textbf{ in} \\
\quad\quad \text{assert } p_2 \\
\quad \textbf{else} \\
\quad\quad \textbf{let } y = \text{charAt } s\ 1 \textbf{ in} \\
\quad\quad \textbf{let } p_3 = \text{eqChar } y\ \text{`b'} \textbf{ in} \\
\quad\quad \text{assert } p_3
\end{array}
$$

(a) A Python program (left) and its λ_Σ equivalent (right).

$$
\overbrace{\Gamma}^{\text{axioms}} \vdash \text{parser} \nearrow \overbrace{\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}}^{\text{incomplete refinement}} \dashv \overbrace{\forall s.\ \kappa(s) \Rightarrow \varphi}^{\text{verification condition}}
$$

$$
\begin{aligned}
\varphi \doteq\ & (\forall v_1.\ v_1 = 0 \Rightarrow v_1 \geq 0 \wedge v_1 < |s|) \wedge \\
& (\forall x.\ x = s[0] \Rightarrow (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow \\
& \quad (p_1 = \text{true} \Rightarrow (\forall n.\ n \geq 0 \wedge n = |s| \Rightarrow n = 1)) \wedge \\
& \quad (p_1 = \text{false} \Rightarrow (\forall v_2.\ v_2 = 1 \Rightarrow v_2 \geq 0 \wedge v_2 < |s|) \wedge \\
& \quad\quad (\forall y.\ y = s[1] \Rightarrow y = \text{`b'}))))
\end{aligned}
$$

(b) An incomplete refinement typing of the λ_Σ program above.

$$
\begin{aligned}
\varphi \overset{1}{\rightsquigarrow}\ & \boxed{|s| > 0} \wedge (\forall x.\ x = s[0] \Rightarrow (\forall p_1.\ p_1 = \text{true} \Leftrightarrow x = \text{`a'} \Rightarrow \\
& \quad (p_1 = \text{true} \Rightarrow \boxed{|s| = 1}) \wedge (p_1 = \text{false} \Rightarrow \boxed{|s| > 1} \wedge \boxed{s[1] = \text{`b'}}))) \\
\overset{2}{\rightsquigarrow}\ & |s| > 0 \wedge (\forall x.\ x = s[0] \Rightarrow (\boxed{x = \text{`a'}} \wedge |s| = 1) \vee (\boxed{x \neq \text{`a'}} \wedge |s| > 1 \wedge s[1] = \text{`b'})) \\
\overset{3}{\rightsquigarrow}\ & |s| > 0 \wedge ((\boxed{s[0] = \text{`a'}} \wedge |s| = 1) \vee (\boxed{s[0] \neq \text{`a'}} \wedge |s| > 1 \wedge s[1] = \text{`b'})) \\
\overset{4}{\rightsquigarrow}\ & \boxed{s \in \Sigma^+} \wedge ((\boxed{s \in \text{a}\Sigma^*} \wedge \boxed{s \in \Sigma}) \vee (\boxed{s \in (\Sigma \backslash \text{a})\Sigma^*} \wedge \boxed{s \in \Sigma^2\Sigma^*} \wedge \boxed{s \in \Sigma\text{b}\Sigma^*})) \\
\overset{5}{\rightsquigarrow}\ & \boxed{s \in (\text{a} + (\Sigma \backslash \text{a})\text{b}\Sigma^*)}
\end{aligned}
$$

(c) Possible steps in the abstract interpretation of φ: (1) eliminate $\forall v_1, \forall n, \forall v_2, \forall y$; (2) eliminate $\forall p_1$; (3) eliminate $\forall x$; (4) abstract s; (5) normalize. Highlights indicate changes relative to previous step.

Fig. 2. An example of grammar inference.

3 Refinement Inference

We now give a formalization of λ_Σ, our core abstraction for representing ad hoc parsers. The language and its type system were heavily inspired the Sprite tutorial language [Jhala and Vazou 2020], Liquid Haskell [Vazou et al. 2014], and the Fusion algorithm [Cosman and Jhala 2017]. Our main contribution in this area is an extended refinement variable solving procedure that synthesizes precise grammars for input string constraints (§§ 3.3 and 4).

3.1 Syntax

λ_Σ is a small λ-calculus that exists solely for type synthesis and its programs are neither meant to be executed nor written by hand. Its syntax (Fig. 3) is thus minimal and has few affordances.

Values are either primitive constant literals or variables. **Terms** are comprised of values and the usual constructs: function applications, $\boldsymbol{\lambda}$-abstractions, **let**-bindings and recursive **rec**-bindings, and branches. Applications are in A-normal form (ANF) [Bowman 2022; Flanagan et al. 1993], which means that functions can only be applied to atomic values and so all intermediate results need to be **let**-bound. ANF is a natural result of the SSA translation performed on the original source code (see § 2.1), but we also generally enforce this in the syntax to simplify the typing rules (see § 3.2, rule Syn/App) and to enable better type inference [cf. Rondon et al. 2008, § 4.4]. There are no type annotations, except on $\boldsymbol{\lambda}$-binders and recursive **rec**-binders, whose (base) types we assume are inferred in a pre-processing phase or provided by the programmer.

The primitive **Base Types** are $\mathbb{1}$ (unit), $\mathbb{B}$ (Boolean), $\mathbb{Z}$ (integer), $\mathbb{Ch}$ (character), and $\mathbb{S}$ (string). **Types** are formed by decorating base types with refinements, or by constructing (dependent) function types whose output types can refer to input types. **Refinements** are either fully known predicates or contain holes $\star$ which are replaced with κ-applications during type synthesis (§ 3.2).

Predicates are terms in a Boolean logic and contain the usual Boolean constants and logical connectives, plus (in)equality relations and arithmetic comparisons between predicate expressions, as well as membership queries for regular expression matching. Predicates also contain applications of unknown κ variables and existential quantifiers, which arise during the Fusion phase of κ solving (see § 3.3). Both κ applications and existentials are not part of the user-visible surface syntax, which is limited to SMT-solvable predicates from quantifier-free linear arithmetic with uninterpreted functions (QF_UFLIA) extended with a theory of operations over strings [Barret et al. 2016]. **Expressions** are built from lifted values and functions, in particular linear integer arithmetic and operations over strings, such as length $|\square|$, character-at-index $\square[\square]$, or substring $\square[\square..\square]$. We assume the usual notational conveniences for functions and write $a + b$ for $+(a, b)$ or $|s|$ for $\mathrm{str.len}(s)$ and so on. To simplify the presentation, predicate expressions are not further syntactically stratified, but are assumed to always occur well-typed (which is assured by the implementation).

VC generation (§ 3.2) results in **Constraints** that are Horn clauses [Bjørner, Gurfinkel, et al. 2015] in negation normal form (NNF), basically tree-like conjunctions of refinement predicates, where each root-to-leaf path is a constrained Horn clause (CHC). This representation of VCs is due to Cosman and Jhala [2017], who cleverly employ the constraints' nested scoping structure to make κ solving tractable. Note that constraints arise during type inference and provide the means to infer typing holes but are not themselves part of any inferred types, which can only contain quantifier-free predicates.

3.2 Type System

The main purpose of the type system of λ_Σ is to generate constraints, in particular input string constraints for parser functions. Thus, the type system is focused on inference/synthesis, rather than type checking. Terms need only be minimally annotated, at $\boldsymbol{\lambda}$-abstractions and recursive

Values	v	::=	unit	unit
		|	true | false	Booleans
		|	$\ldots, -1, 0, 1, \ldots$	integers
		|	‘a’, ...	characters
		|	“abc”, ...	strings
		|	$x, y, z, \ldots$	variables
Terms	e	::=	v	value
		|	$e\ v$	application
		|	$\boldsymbol{\lambda} x : b.\ e$	abstraction
		|	**let** $x = e_1$ **in** e_2	binding
		|	**rec** $x : t = e_1\ \ e_2$	recursion
		|	**if** v **then** e_1 **else** e_2	branch
Base Types	b	::=	$\mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{Ch} \mid \mathbb{S}$	
Types	t	::=	$\{x : b \mid r\}$	refined base
		|	$x : t_1 \longrightarrow t_2$	dependent function
Refinements	r	::=	p	known predicate
		|	$\star$	refinement hole
Predicates	p	::=	true | false	Boolean constants
		|	$p_1 \wedge p_2 \mid p_1 \vee p_2$	connectives
		|	$p_1 \Rightarrow p_2 \mid p_1 \Leftrightarrow p_2$	implications
		|	$\neg p$	negation
		|	$\exists x : b.\ p$	existential quantification
		|	$\kappa(\overline{v})$	κ application
		|	$w_1 = w_2 \mid w_1 \neq w_2$	(in)equality
		|	$w_1 < w_2 \mid w_1 \leq w_2$	arithmetic comparison
		|	$w \in \mathrm{RE}$	regular language membership
Expressions	w	::=	v	value
		|	$f(\overline{w})$	function
Constraints	c	::=	p	predicate
		|	$c_1 \wedge c_2$	conjunction
		|	$\forall x : b.\ p \Rightarrow c$	universal implication

Fig. 3. Syntax of λ_Σ terms, types, and refinements.

bindings. The types of applied functions need to be known, however, and be available in the typing context (see the discussion of axioms, § 2.1 and Table 1). We combine typing rules and VC generation into one syntax-driven declarative system of inference rules, shown in Fig. 4.

$\boxed{t_1 <: t_2 \dashv c}$ ***Subtyping.*** A type t_1 is a subtype of t_2, meaning the values denoted by t_1 are subsumed by t_2, if the entailment constraint c is satisfied. In the Sub/Base case, this means that t_1

$\boxed{t_1 <: t_2 \dashv c}$ **Subtyping**

$$\frac{}{\{v_1 : b \mid p_1\} <: \{v_2 : b \mid p_2\} \dashv \forall v_1 : b.\ p_1 \Rightarrow p_2[v_2 := v_1]}\ \textsc{Sub/Base}$$

$$\frac{s_2 <: s_1 \dashv c_i \qquad t_1[x_1 := x_2] <: t_2 \dashv c_o}{x_1 : s_1 \to t_1 <: x_2 : s_2 \to t_2 \dashv c_i \wedge (x_2 :: s_2 \Rightarrow c_o)}\ \textsc{Sub/Fun}$$

$\boxed{\Gamma \vdash t \rhd \hat{t}}$ **Template Generation**

$$\frac{\kappa \text{ is a fresh variable of sort } b \times \overline{t}}{\overline{x : t} \vdash \{v : b \mid \star\} \rhd \{v : b \mid \kappa(v, \overline{x})\}}\ \textsc{Kap/Base} \qquad \frac{}{\Gamma \vdash \{v : b \mid p\} \rhd \{v : b \mid p\}}\ \textsc{Kap/Conc}$$

$$\frac{\Gamma \vdash t_1 \rhd \hat{t}_1 \qquad \Gamma[x \mapsto t_1] \vdash t_2 \rhd \hat{t}_2}{\Gamma \vdash x : t_1 \to t_2 \rhd x : \hat{t}_1 \to \hat{t}_2}\ \textsc{Kap/Fun}$$

$\boxed{\Gamma \vdash e \nearrow t \dashv c}$ **Type/Constraint Synthesis**

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \nearrow \mathsf{self}(x, t) \dashv \mathsf{true}}\ \textsc{Syn/Var} \qquad \frac{\mathsf{prim}(v) = t}{\Gamma \vdash v \nearrow t \dashv \mathsf{true}}\ \textsc{Syn/Con}$$

$$\frac{\Gamma \vdash e \nearrow y : t_1 \to t_2 \dashv c_e \qquad \Gamma \vdash v \nearrow t_v \qquad t_v <: t_1 \dashv c_v}{\Gamma \vdash e\ v \nearrow t_2[y := v] \dashv c_e \wedge c_v}\ \textsc{Syn/App}$$

$$\frac{\Gamma \vdash \lfloor \tilde{t}_1 \rfloor \rhd \hat{t}_1 \qquad \Gamma[x \mapsto \hat{t}_1] \vdash e \nearrow t_2 \dashv c_2}{\Gamma \vdash \boldsymbol{\lambda} x : \tilde{t}_1.\ e \nearrow x : \hat{t}_1 \to t_2 \dashv x :: \hat{t}_1 \Rightarrow c_2}\ \textsc{Syn/Lam}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \nearrow t_1 \dashv c_1 \\ \Gamma[x \mapsto t_1] \vdash e_2 \nearrow t_2 \dashv c_2 \qquad \Gamma \vdash \lfloor t_2 \rfloor \rhd \hat{t}_2 \qquad t_2 <: \hat{t}_2 \dashv \hat{c}_2\end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \nearrow \hat{t}_2 \dashv c_1 \wedge (x :: t_1 \Rightarrow c_2 \wedge \hat{c}_2)}\ \textsc{Syn/Let}$$

$$\frac{\begin{array}{lll}\Gamma \vdash \tilde{t}_1 \rhd \hat{t}_1 & \Gamma[x \mapsto \hat{t}_1] \vdash e_1 \nearrow t_1 \dashv c_1 & t_1 <: \hat{t}_1 \dashv \hat{c}_1 \\ & \Gamma[x \mapsto t_1] \vdash e_2 \nearrow t_2 \dashv c_2 \qquad \Gamma \vdash \lfloor t_2 \rfloor \rhd \hat{t}_2 & t_2 <: \hat{t}_2 \dashv \hat{c}_2\end{array}}{\Gamma \vdash \mathbf{rec}\ x : \tilde{t}_1 = e_1\ \ e_2 \nearrow \hat{t}_2 \dashv (x :: \hat{t}_1 \Rightarrow c_1 \wedge \hat{c}_1) \wedge (x :: t_1 \Rightarrow c_2 \wedge \hat{c}_2)}\ \textsc{Syn/Rec}$$

$$\frac{\begin{array}{lll}\Gamma \vdash x \nearrow \mathbb{B} & \Gamma \vdash e_1 \nearrow t_1 \dashv c_1 \qquad \Gamma \vdash \lfloor t_1 \rfloor \rhd \hat{t} & t_1 <: \hat{t} \dashv \hat{c}_1 \\ & \Gamma \vdash e_2 \nearrow t_2 \dashv c_2 & t_2 <: \hat{t} \dashv \hat{c}_2\end{array}}{\Gamma \vdash \mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \nearrow \hat{t} \dashv (x = \mathsf{true} \Rightarrow c_1 \wedge \hat{c}_1) \wedge (x = \mathsf{false} \Rightarrow c_2 \wedge \hat{c}_2)}\ \textsc{Syn/If}$$

Fig. 4. Typing rules for λ_Σ.

and t_2 must be of the same base type and the refinement predicate of t_1 must imply the predicate of t_2, for all possible values the types can have. In the Sub/Fun case, where the contra-variant input constraint is joined with the co-variant output constraint, we add an additional implication to the output constraint, strengthening it with the supertype's input predicate. To ensure that only base types are bound to quantifiers, this is done using a *generalized implication* operation,

$$x :: t \Rightarrow c \stackrel{\text{def}}{=} \begin{cases} \forall x : b.\ p[v := x] \Rightarrow c & \text{if } t \equiv \{v : b \mid p\}, \\ c & \text{otherwise.} \end{cases}$$

$\boxed{\Gamma \vdash t \triangleright \hat{t}}$ ***Template Generation.*** If the refinement of a type t is unknown (i.e., t contains a refinement hole $\star$), we can turn the type into a template $\hat{t}$, where the unknown refinement predicate is denoted by a placeholder variable whose resolution is deferred (see §§ 2.2, 3.3, and 4). The rule Kap/Base introduces a fresh κ variable for the hole, representing an n-ary relation between the type itself and all variables in the current environment Γ. In the function case Kap/Fun, templates are generated recursively while extending the output environment with the input binder. If t already has a concrete refinement (i.e., t does not contain a hole, although it might already contain a κ variable), then Kap/Conc simply leaves the type as-is. To enable complete type synthesis for all intermediate terms, it is sometimes necessary to turn a type into a template, even if it already has some known refinement (e.g., true). To facilitate this, the $\lfloor t \rfloor$ function returns the *shape* of a type t by erasing any refinements from t and returning the non-refined version of the type.

$\boxed{\Gamma \vdash e \nearrow t \dashv c}$ ***Type/Constraint Synthesis.*** Given a typing context Γ mapping values to types, and a term e, we can synthesize a type t whose correctness is implied by the constraint c. The rule Syn/Var retrieves the type of a variable from the current context, using *selfification* [Ou et al. 2004] to produce the most precise possible type by lifting the variable into the refinement, allowing each occurrence of the variable in different branches of the program to be precisely typed.

$$\mathsf{self}(x, t) \stackrel{\text{def}}{=} \begin{cases} \{v : b \mid p \wedge v = x\} & \text{if } t \equiv \{v : b \mid p\}, \\ t & \text{otherwise.} \end{cases}$$

Syn/Con synthesizes built-in primitive types for constant values, denoted by $\mathsf{prim}(v)$ in the obvious way, i.e., $\mathsf{prim}(0) \stackrel{\text{def}}{=} \{v : \mathbb{Z} \mid v = 0\}$ and so on. Syn/App synthesizes the result type of a function application, where the given input can be a subtype of the function's declared input type. In the result type, the declared input variable is replaced by the given input, using standard capture-avoiding substitution. Note that because our terms are in ANF, no arbitrary expressions are introduced into the type during this substitution. The synthesized VC is simply a conjunction of the function's VC and the constraint created by the subtyping judgement. Syn/Lam produces a function type whose input refinement is yet unknown, with a base type according to the annotation on the $\boldsymbol{\lambda}$-binder. Syn/Let first synthesizes the type t_1 for the bound term, then the type t_2 for the expression's body under an environment where x is bound to t_1. To ensure that x does not escape its scope, the whole **let**-expression is given the templated type $\hat{t}_2$, a supertype of t_2 with a κ variable in place of its refinement. Syn/Rec is similar to Syn/Let, with the addition that we first assume the bound term's type as a placeholder $\hat{t}_1$ based on the annotation $\tilde{t}_1$ on the binder, before synthesizing it as t_1, allowing for recursion in the bound term. Note that we do not erase any pre-existing refinement of $\tilde{t}_1$, if there is one. This enables us to manually direct constraint synthesis when necessary, for example to inject loop invariants. Syn/If synthesizes the type of the whole conditional as a (templated) supertype of both branches. In the VC, we imply the **then**-branch's constraints if the condition is true, and the **else**-branch's constraints if the condition is false. The κ variable in the templated supertype $\hat{t}$ allows this path-sensitive information to travel back upwards.

3.3 Variable Solving

Before the synthesized VCs can be sent off to an SMT solver to be proven valid, we have to replace all κ variables with concrete refinement predicates. Some of these variables represent input string refinements and this is the point where we need to find the grammars describing those strings.

Algorithm 1 gives a high-level overview of the VC solving procedure. We start with an incomplete VC constraint c that contains any number of κ variables. We use Fusion [Cosman and Jhala 2017] and predicate abstraction [Rondon et al. 2008] to deal with non-grammar κ variables. For grammar variables, we use our abstract interpretation approach, as detailed in § 4. If the complete VC, with all κ variables replaced by their solutions, is satisfiable, then the inferred type is valid and σ contains concrete assignments for all of the type's refinement variables, including grammar refinements for all string types. If no satisfying solution could be found, the program must be ill-typed.

Algorithm 1 Solve an incomplete VC and find all κ assignments

```
1: procedure Solve(c)
2:     c ← Fusion(c)                    ▷ eliminate local acyclic κ variables [Cosman and Jhala 2017]
3:     σ ← Liquid(c)                    ▷ solve residual non-grammar κ variables [Rondon et al. 2008]
4:     for all constraints in σ(c) of the form ∀s : 𝕊. κ(s) ⇒ φ do
5:         σ̂ ← [[φ]]♯                  ▷ infer grammar using abstract interpretation [§ 4]
6:         σ ← σ ∪ {κ(s) ↦ s ∈ σ̂(s)}
7:     end for
8:     if Satisfiable(σ(c)) then return Valid σ else return Invalid
9: end procedure
```

4 Grammar Inference

Given a program P with a partially inferred refinement type $\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}$ and an incomplete verification condition of the form $\forall s : \mathbb{S}.\ \kappa(s) \Rightarrow \varphi$, our goal is to find an assignment for κ that both validates the VC and meaningfully refines the type of s. Trivially, the assignment $\kappa(s) \mapsto \text{false}$ always validates the VC, but is hardly a meaningful refinement. Assuming that P is a parser, we ideally want an assignment for κ that constrains s in a way that

(1) disallows any string rejected by P (*soundness*) and
(2) allows all strings accepted by P (*completeness*).

In other words, we want a refinement for s that is a *grammar* for P, i.e., a finite description of the (possibly infinite) set of strings contained in the language $L(P)$.

Luckily, the VC's consequent φ already appears to be what we are looking for: it is a first-order formula over a free string variable s that exactly captures the semantics of the parsing operations performed by P on s. Assuming that φ itself does not contain any other unsolved κ variables (these having been eliminated by prior inference and solving steps, e.g., Fusion and predicate abstraction), as well as the correctness of all involved axiomatic string function specifications, then, under some model $\mathfrak{M}$ of the refinement logic (e.g., QF_UFLIA plus basic string operations), by construction,

$$\mathfrak{M}, [s \mapsto t] \models \varphi \quad \Longleftrightarrow \quad P \text{ successfully parses } t \quad \Longleftrightarrow \quad t \in L(P).$$

The parser P represented by φ accepts some particular string t if and only if φ is true under an assignment of s to t. The set of all strings that satisfy φ is exactly the language $L(P)$ accepted by P. Thus, φ is technically a complete grammar for P.

Practically, however, φ makes for a rather poor grammar. It is a first-order formula close in size and structure to the parsing program P itself. While $\kappa(s) \mapsto \varphi$ is an ideal assignment in the sense

that it is both sound and complete, it results in a rather impractical refinement that would puzzle a human programmer. The presence of quantifiers within φ complicates any further analysis.

In order to obtain a better grammar for $L(P)$, we will use φ as a starting point to derive a quantifier-free predicate that is much simpler but semantically equivalent. Our approach comprises

(1) an abstract interpretation of certain first-order string formulas as grammars (§ 4.1),
(2) an extended constraint syntax that mixes symbolic and abstract representations (§ 4.2),
(3) an abstract semantics of relations between predicate expressions (§ 5),
(4) a procedure for eliminating quantified variables by abstract substitution (§ 5.1),
(5) abstract value representations of all base types (§ 6).

4.1 Abstract Interpretation

Background. Abstract interpretation is a well-established framework for formalizing static program analyses [P. Cousot and R. Cousot 1977, 1979]. It involves the sound approximation of all possible states of a program, usually trading precision for efficiency. The concrete semantics of a program $\mathcal{P}$ are defined by a semantic function $[\![\square]\!] : \mathcal{P} \to \wp(C)$, which produces a power set of concrete values C. The concrete domain $\wp(C)$ can be approximated by an abstract domain $\mathcal{A}$, with an abstraction function $\alpha : \wp(C) \to \mathcal{A}$ and a concretization function $\gamma : \mathcal{A} \to \wp(C)$ mapping elements between the domains. Usually, $\mathcal{A}$ is a complete lattice $\langle \mathcal{A}, \sqsubseteq, \sqcap, \sqcup, \bot, \top \rangle$ and the two domains form a Galois connection $\langle \wp(C), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{A}, \sqsubseteq \rangle$, which intuitively means that relationships between elements of $\wp(C)$ also hold between the corresponding abstracted elements of $\mathcal{A}$. We can then define an abstract semantics $[\![\square]\!]^\sharp : \mathcal{P} \to \mathcal{A}$ to abstractly interpret programs and directly produce abstract values. The abstract interpretation is *sound* iff $\alpha([\![\mathcal{P}]\!]) \sqsubseteq [\![\mathcal{P}]\!]^\sharp$ for all programs $\mathcal{P}$, and it is also *complete* iff $\alpha([\![\mathcal{P}]\!]) = [\![\mathcal{P}]\!]^\sharp$. Completeness in this context means that the abstract semantics incurs no loss of precision relative to the underlying abstract domain, i.e., the abstract semantics can take full advantage of the whole domain. Finally, an abstract interpretation is *exact* iff $[\![\mathcal{P}]\!] = \gamma([\![\mathcal{P}]\!]^\sharp)$, meaning the abstraction loses no information and the abstract semantics exactly captures the concrete semantics of the program [P. Cousot 1997; Giacobazzi and Quintarelli 2001].

Parser Semantics. In our case, the kind of program we want to abstractly interpret is a parser P represented by a first-order formula φ. We take the concrete semantics $[\![\varphi]\!]$ to be an assignment σ of free variables to values, with the parser's input string denoted by the free variable s, such that

$$\mathfrak{M}, \sigma \models \varphi \quad \Longleftrightarrow \quad \sigma \in [\![\varphi]\!] \quad \Longleftrightarrow \quad \sigma(s) \in L(P).$$

Trying to compute $[\![\varphi]\!]$ directly will generally result in an infinite set of values. Hence our desire for an abstract semantics $[\![\varphi]\!]^\sharp$ that produces a finite approximation of this set such that

$$\mathfrak{M}, \hat{\sigma} \models \varphi \quad \Longleftarrow \quad \hat{\sigma} = [\![\varphi]\!]^\sharp \quad \Longrightarrow \quad \hat{\sigma}(s) \subseteq L(P),$$

where $\hat{\sigma}$ is an assignment of free variables to abstract values. In particular, $\hat{\sigma}(s)$ is now a grammar describing (a subset of) the strings in $L(P)$.

The completeness of the approximation $\hat{\sigma}$ depends on the underlying abstract domains. Panini uses the following domains to abstract primitive values (complete definitions are given in § 6):

unit	$\wp(\mathbb{1})$	$=$	$\hat{\mathbf{1}}$	the two-element lattice
Booleans	$\wp(\mathbb{B})$	$=$	$\hat{\mathbf{B}}$	Boolean subset lattice
integers	$\wp(\mathbb{Z})$	$\supseteq$	$\hat{\mathbf{Z}}$	open-ended interval lists
characters	$\wp(\mathbb{Ch})$	$=$	$\hat{\mathbf{C}}$	Unicode character sets
strings	$\wp(\mathbb{S})$	$\supseteq$	$\hat{\mathbf{S}}$	regular expressions over $\hat{\mathbf{C}}$

$$[\![\varphi]\!]^\sharp \doteq \left\{x \mapsto [\![\varphi]\!]\uparrow_x \;\middle|\; x \in \mathrm{vars}(\varphi)\right\}$$

$$\begin{aligned}
[\![\varphi_1 \wedge \varphi_2]\!]\uparrow_x &\doteq [\![\varphi_1]\!]\uparrow_x \sqcap [\![\varphi_2]\!]\uparrow_x \quad \text{if } \varphi_1, \varphi_2 \text{ quantifier-free} \\
[\![\varphi_1 \vee \varphi_2]\!]\uparrow_x &\doteq [\![\varphi_1]\!]\uparrow_x \sqcup [\![\varphi_2]\!]\uparrow_x \quad \text{if } \varphi_1, \varphi_2 \text{ quantifier-free} \\
[\![\neg\varphi]\!]\uparrow_x &\doteq \neg[\![\varphi]\!]\uparrow_x \quad \text{if } \varphi \text{ quantifier-free} \\
[\![\varphi]\!]\uparrow_x &\doteq [\![\mathsf{qelim}(\varphi)]\!]\uparrow_x \\
[\![\rho]\!]\uparrow_x &\doteq \begin{cases} \omega, & \text{as defined by domain;} \\ \langle \rho[x := \bullet] \rangle, & \text{otherwise.} \end{cases}
\end{aligned}$$

Fig. 5. Abstract semantics of λ_Σ constraints.

Note that our abstract string domain $\hat{\mathbf{S}}$ represents sets of strings with regular expressions. This means that we must under-approximate any $L(P)$ above regular in the Chomsky hierarchy. For super-regular languages, we can only infer a partial grammar representing a regular subset, if one exists. However, if $L(P)$ is (at most) regular, then we can infer a complete grammar for P.

Using our abstract interpretation, we can construct a finite but quantifier-free solution for the grammar refinement variable,

$$\kappa(s) \mapsto s \in \hat{\sigma}(s), \quad \text{where } \hat{\sigma} = [\![\varphi]\!]^\sharp.$$

The definition of the abstract semantics function $[\![\square]\!]^\sharp$ is given in Fig. 5. It depends on a novel variable-focused relational semantics $[\![\square]\!]\uparrow_\square$ and a quantifier elimination procedure $\mathsf{qelim}(\square)$. We describe these in the remainder of this section, after establishing some extensions to λ_Σ constraints.

4.2 Extended Constraint Syntax

We extend the formal syntax of predicates and constraints from Fig. 3 to include abstract values and relations. The extended syntax is given in Fig. 6.

Constraints φ, in addition to Boolean constants and the usual logical connectives, now include both universal and existential quantification, as well as generalized relations ρ between (potentially abstract) expressions. There are no κ applications at this point. The base types of all bound variables are known, but we elide them from the presentation to reduce clutter.

Expressions ω, in addition to concrete values (e.g., integers or string literals, as well as variables; see Fig. 3), now also include abstract values and abstract relations. **Abstract Values** $\hat{v}$ are taken from the underlying abstract domains (see § 6) and are representations of potentially infinite sets of concrete values. For example, the abstract value $[1, \infty]$ represents an infinite set of integers $\{1, 2, 3, \dots\}$ and the abstract string $\Sigma^*\mathsf{a}$ represents the set of all strings that end with the character a. If an abstract value appears in a function, it lifts the whole expression into the abstract realm; for example, $x + [1, \infty]$ represents the set of expressions $\{x + 1, x + 2, x + 3, \dots\}$.

If a relation contains a hole $\bullet$, it is an **Abstract Relation** $\langle\rho\rangle$ that represents the maximum set of values that can be put into the hole to make the relation true. For example, the abstract relation $\langle \bullet > 5 \rangle$ represents "the set of all values that are greater than five" and $\langle s[\bullet] = c \rangle$ represents the set of all indexes at which the string in variable s contains the character in variable c.

Constraints	φ	::=	true | false	Boolean constants
		|	$\varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$	connectives
		|	$\varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2$	implications
		|	$\neg\varphi$	negation
		|	$\exists x.\, \varphi$	existential quantification
		|	$\forall x.\, \varphi$	universal quantification
		|	ρ	relations
Relations	ρ	::=	$\omega_1 \bowtie \omega_2$ where $\bowtie \in \{=, \neq, <, \leq, \in, \notin, \between, \parallel\}$	
Expressions	ω	::=	v	concrete value
		|	$\hat{v}$	abstract value
		|	$\langle\rho\rangle$	abstract relation
		|	$\bullet$	hole
		|	$f(\overline{\omega})$	function

Fig. 6. Extended syntax of λ_Σ constraints.

In general, **Relations** ρ are comprised of the usual (in)equality and arithmetic comparisons, as well as set (non)inclusion ($\in, \notin$) and (non-)empty intersection ($\between, \parallel$). The latter are simply abbreviations for common set operations, with

$$A \between B \equiv A \cap B \neq \varnothing, \quad \text{meaning "}A \text{ and } B \text{ have at least one element in common,"}$$
$$A \parallel B \equiv A \cap B = \varnothing, \quad \text{meaning "}A \text{ and } B \text{ have no elements in common."}$$

Since abstract values are essentially sets, relating them works the same way. Note the distinction between $x \in [0, \infty]$ (i.e., "x is a member of the set {0,1,…}"), $x = [0, \infty]$ (i.e., "x is the set {0,1,…}"), and $x \between [0, \infty]$ (i.e., "x has at least one element in common with the set {0,1,…}").

Normalization. Both expressions and relations can be normalized by partial evaluation. If abstract values are involved, the semantics of the particular abstract domains apply. If possible, relations are fully abstracted using their relational semantics (§ 5). In the remainder, we assume that expressions and relations are always fully normalized; some examples are given below.

$$1 + x - 2 \rightsquigarrow x - 1 \qquad x + 2 \leq 5 \rightsquigarrow x \leq 3 \rightsquigarrow x \in [-\infty, 3]$$
$$\langle \bullet > 5 \rangle \rightsquigarrow [6, \infty] \qquad |s| + 1 > 2 \rightsquigarrow |s| > 1 \rightsquigarrow s \in \Sigma^+$$
$$s[i..i] = \text{“a”} \rightsquigarrow s[i] = \text{‘a’} \qquad [1,2] \between \langle s[\bullet] = \text{‘a’} \rangle \rightsquigarrow s\big[[1,2]\big] \ni \text{‘a’} \rightsquigarrow s \in \Sigma\Sigma^?\text{a}\Sigma^*$$

5 Relational Semantics

The concrete semantics $[\![\rho]\!]$ of a single relation are all those assignments of the relation's free variables that make the relation true. For example,

$$[\![x > 3]\!] \doteq \{x \mapsto 4, x \mapsto 5, \dots\},$$
$$[\![|s| > 3]\!] \doteq \{\dots, s \mapsto \text{“abaa”}, s \mapsto \text{“abab”}, \dots\}$$
$$[\![|s| > x]\!] \doteq \left\{\dots, \begin{pmatrix} s \mapsto \text{“ab”} \\ x \mapsto 0 \end{pmatrix}, \begin{pmatrix} s \mapsto \text{“ab”} \\ x \mapsto 1 \end{pmatrix}, \dots\right\}.$$

Unfortunately, constructing an abstract semantics even for such simple relations is decidedly non-trivial. It requires complex relational domains to capture just a limited set of constraints between multiple variables [P. Cousot and Halbwachs 1978; Logozzo and Fähndrich 2010; Simon et al. 2003].

Fortunately, we do not actually need a full abstract semantics that condenses relations over multiple variables into singular abstract values. For our purposes, it is sufficient to consider relational semantics on a per-variable basis. To this end, we introduce a variable-focused abstract semantics function $[\![\rho]\!]\uparrow_x$ that for a given variable x occurring free in ρ produces an abstract expression whose concrete values are exactly those that could be substituted for x to make ρ true,

$$\mathfrak{M}, [x \mapsto c] \models \rho \quad \iff \quad c \in [\![\rho]\!]\uparrow_x.$$

Abstract relations $\langle\rho\rangle$ provide a convenient "default" implementation,

$$[\![\rho]\!]\uparrow_x \doteq \langle\rho[x := \bullet]\rangle.$$

But we can often do better than this. Depending on the underlying abstract domains and domain-specific knowledge about the involved operations, more specific definitions of $[\![\square]\!]\uparrow_\square$ are possible (see § 6). For example, for the domains of regular strings and integers, we can define precise abstractions for simple string length relations,

$$[\![|s| = \mathbf{n}]\!]\uparrow_s \doteq \Sigma^{\mathbf{n}}, \qquad [\![|s| \geq \mathbf{n}]\!]\uparrow_s \doteq \Sigma^{\mathbf{n}}\Sigma^*, \qquad [\![|s| \leq \mathbf{n}]\!]\uparrow_s \doteq \Sigma^0 + \Sigma^1 + \cdots + \Sigma^{\mathbf{n}},$$

where $\mathbf{n}$ is a meta-variable denoting some concrete integer value. With these and other domain-specific semantics, we can construct variable-focused abstractions for the earlier examples. Note how in all cases the abstraction effectively eliminates the chosen variable:

$$[\![x > 3]\!]\uparrow_x \doteq [4, \infty] \qquad [\![|s| > 3]\!]\uparrow_s \doteq \Sigma^4\Sigma^* \qquad [\![|s| > x]\!]\uparrow_s \doteq \langle|\bullet| > x\rangle$$
$$[\![|s| > x]\!]\uparrow_x \doteq |s| - [1, \infty]$$

5.1 Quantifier Elimination

Figure 7 presents our procedure for eliminating quantifiers by abstract substitution. The top-level function $\mathsf{qelim}(\varphi)$ traverses a given constraint φ to eliminate all of its quantifiers. During this traversal, we apply standard logical transformations to simplify the problem, such as DeMorgan's law and distributivity of $\exists$ over $\forall$. The actual variable elimination happens in $\mathsf{qelim1}(x, R)$, where we eliminate a single (existentially quantified) variable x from one conjunction of relations R, returning a conjunctive predicate that is logically equivalent to R but no longer contains x, i.e.,

$$\mathfrak{M} \models \exists x.\, R \iff \mathfrak{M} \models \hat{R} \quad \text{where } \hat{R} = \mathsf{qelim1}(x, R) \text{ and } x \notin \hat{R}.$$

To eliminate a particular variable x from the relations in R, we first compute the x-relative relational semantics $[\![\rho]\!]\uparrow_x$ for all relations that contain x as a free variable. We then form the (partial) meet closure of these expressions, resulting in a minimal set of x-less expressions E, whose members all represent some aspect of x in R. While the expressions in E do not contain x, they might contain other free variables. We now generate all unordered pairwise combinations of expressions in E and create a new non-empty intersection relation between each pair of expressions. The result is a set of relations $\hat{R}$ that contain no reference to x yet preserve the semantics of the original conjunction of relations. Finally, we return $\hat{R}$ together with those relations in R that did not contain x.

6 Abstract Domains

We now define abstract domains for each of the base types in our system. Each domain efficiently captures (possibly infinite) sets of concrete values of the corresponding type, lifts certain operations of the type to the abstract domain, and defines some abstract relational semantics $[\![\square]\!]\uparrow_\square$ for expression involving those operations. To simplify the definitions and avoid boilerplate repetition,

$$
\begin{aligned}
\mathsf{qelim}(\exists x.\ \varphi) &\doteq \bigvee \left\{ \mathsf{qelim1}(x, R) \;\middle|\; R \in \mathrm{dnf}(\mathsf{qelim}(\varphi)) \right\} \\
\mathsf{qelim}(\forall x.\ \varphi) &\doteq \mathsf{qelim}(\neg\exists x.\ \neg\varphi) \\
\mathsf{qelim}(\varphi_1 \wedge \varphi_2) &\doteq \mathsf{qelim}(\varphi_1) \wedge \mathsf{qelim}(\varphi_2) \\
\mathsf{qelim}(\varphi_1 \vee \varphi_2) &\doteq \mathsf{qelim}(\varphi_1) \vee \mathsf{qelim}(\varphi_2) \\
\mathsf{qelim}(\neg\varphi) &\doteq \neg\mathsf{qelim}(\varphi) \\
\mathsf{qelim}(\rho) &\doteq \rho \\
\mathsf{qelim}(\mathsf{true}) &\doteq \mathsf{true} \\
\mathsf{qelim}(\mathsf{false}) &\doteq \mathsf{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{qelim1}(x, R) &\doteq \hat{R} \wedge \{\rho \in R \mid x \notin \mathrm{vars}(\rho)\} \\
&\text{where} \\
&\hat{R} = \left\{ \omega_1 \between \omega_2 \;\middle|\; (\omega_1, \omega_2) \in \binom{E}{2} \right\} \\
&E = \dot{\bigsqcap} \left\{ [\![\rho]\!]\!\uparrow_x \mid \rho \in R,\ x \in \mathrm{vars}(\rho) \right\}
\end{aligned}
$$

Fig. 7. Quantifier elimination

we generally assume that expressions have already been arranged in a uniform manner and are fully normalized (§ 4.2). We also forego subscripting domain-specific operations and elements if there is no ambiguity (e.g., we write $\top$ instead of differentiating $\top_{\hat{\mathbf{1}}}$, $\top_{\hat{\mathbf{B}}}$, and so on).

6.1 Unit

The abstract unit type $\hat{\mathbf{1}}$ simply adds a bottom element, forming the two-element lattice, with

$$\alpha = \gamma = \mathrm{id}, \qquad [\![x = \mathrm{unit}]\!]\!\uparrow_x \doteq \mathrm{unit}, \qquad [\![x \neq \mathrm{unit}]\!]\!\uparrow_x \doteq \bot.$$

6.2 Booleans

The Boolean subset lattice $\langle \wp(\mathbb{B}), \subseteq \rangle$ is a complete abstraction of the Boolean values, adding $\varnothing$ and $\{\mathrm{true}, \mathrm{false}\}$ as bottom and top elements, respectively, and forming a complete complemented lattice via subset inclusion and set complement. For consistency with the other definitions, we call this abstraction $\hat{\mathbf{B}}$ and will use the notation $\langle \hat{\mathbf{B}}, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg \rangle$ for the lattice elements and operations. The abstract semantics are defined by

$$\alpha = \gamma = \mathrm{id}, \qquad [\![x = \mathbf{b}]\!]\!\uparrow_x \doteq \{\mathbf{b}\}, \qquad [\![x \neq \mathbf{b}]\!]\!\uparrow_x \doteq \{\neg\mathbf{b}\}.$$

6.3 Integers

Any concrete set of contiguous integers $\{x \in \mathbb{Z} \mid a \leq x \leq b\}$ can be represented efficiently as an interval $[a, b]$, even allowing for a or b to be $\pm\infty$. The domain of integer intervals

$$\mathbf{IZ} \stackrel{\text{def}}{=} \left\{ [a, b] \;\middle|\; a, b \in \mathbb{Z} \cup \{-\infty, \infty\},\ a \leq b \right\}$$

forms a pseudo-semi-lattice $\langle \mathbf{IZ}, \sqsubseteq, \top, \sqcup, \sqcap \rangle$ with a bounded meet but no $\bot$ element:

$$\top = [-\infty, \infty] \qquad [a,b] \sqcup [c,d] = [\min(a,c), \max(b,d)]$$
$$[a,b] \sqsubseteq [c,d] \iff c \le a \wedge b \le d \qquad [a,b] \sqcap [c,d] = [\max(a,c), \min(b,d)]$$

We can also define some standard operations and convenient relations between intervals:[2]

$$[a,b] + [c,d] = [a+c, b+d] \qquad [a,b] \text{ precedes } [c,d] \iff b < (c-1)$$
$$[a,b] - [c,d] = [a-d, b-c] \qquad [a,b] \text{ is before } [c,d] \iff b < c$$
$$[a,b] \text{ contains } [c,d] \iff a \le c \wedge d \le b$$
$$[a,b] \text{ overlaps } [c,d] \iff a \le c \wedge c \le b \wedge b < d$$

While **IZ** can represent infinite contiguous sets, such as those defined by a single inequality relation like $\{x \in \mathbb{Z} \mid x > 5\}$, it cannot represent even finite non-contiguous sets like $\{1, 5, 7\}$ or inequalities like $\{x \in \mathbb{Z} \mid x \neq 2\}$. Thus the connection between $\mathbb{Z}$ and **IZ** is only partial:

$$\alpha : \wp(\mathbb{Z}) \hookrightarrow \mathbf{IZ} \qquad \gamma : \mathbf{IZ} \to \wp(\mathbb{Z})$$
$$\alpha\big(\{x \in \mathbb{Z} \mid a \le x \le b\}\big) = [a,b] \qquad \gamma\big([a,b]\big) = \{x \in \mathbb{Z} \mid a \le x \le b\}$$

To completely represent *non*-contiguous sets of integers, we can use ordered lists of non-overlapping intervals; e.g., $\{1, 2, 3, 7, 8, \ldots\}$ can be represented as $[1,3][7,\infty]$. As long as the number of gaps between intervals is bounded, $\wp(\mathbb{Z})$ can be efficiently abstracted by the domain

$$\hat{\mathbf{Z}} \stackrel{\text{def}}{=} \left\{ x_1 x_2 \cdots x_n \;\middle|\; x_i \in \mathbf{IZ},\ x_i \text{ is before } x_{i+1},\ i \le n \right\},$$

which forms a complete complemented lattice $\langle \hat{\mathbf{Z}}, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg \rangle$, with $\bot = \varnothing$ and $\top = [-\infty, \infty]$ and the operations defined below. We use Haskell list notation $(x : xs)$ to peel off (or add on) the first interval x in a list, with xs denoting the remaining intervals.

$$(x : xs) \sqsubseteq (y : ys) \iff (x \sqsubseteq_{\mathbf{IZ}} y \wedge xs \sqsubseteq (y : ys)) \vee (y \text{ is before } x \wedge (x : xs) \sqsubseteq ys)$$

$$(x : xs) \sqcup (y : ys) = \begin{cases} x : (xs \sqcup (y : ys)) & \text{if } x \text{ precedes } y \\ y : ((x : xs) \sqcup ys) & \text{if } y \text{ precedes } x \\ ((x \sqcup_{\mathbf{IZ}} y) \sqcup xs) \sqcup ys & \text{otherwise} \end{cases}$$

$$(x : xs) \sqcap (y : ys) = \begin{cases} xs \sqcap (y : ys) & \text{if } x \text{ is before } y \\ (x : xs) \sqcap ys & \text{if } y \text{ is before } x \\ (x \sqcap_{\mathbf{IZ}} y) : ((x : xs) \sqcap ys) & \text{if } x \text{ contains } y \text{ or } y \text{ overlaps } x \\ (x \sqcap_{\mathbf{IZ}} y) : (xs \sqcap (y : ys)) & \text{if } y \text{ contains } x \text{ or } x \text{ overlaps } y \\ (x \sqcap_{\mathbf{IZ}} y) : (xs \sqcap ys) & \text{otherwise} \end{cases}$$

$$\neg[a_1,b_1]\cdots[a_n,b_n] = \begin{cases} [b_1{+}1, a_2{-}1]\cdots[b_{n-1}{+}1, a_n{-}1] & \text{if } a_1, b_n = -\infty, \infty \\ [b_1{+}1, a_2{-}1]\cdots[b_{n-1}{+}1, a_n{-}1][b_n{+}1, \infty] & \text{if } a_1 = -\infty \\ [-\infty, a_1{-}1][b_1{+}1, a_2{-}1]\cdots[b_{n-1}{+}1, a_n{-}1] & \text{if } b_n = \infty \\ [-\infty, a_1{-}1][b_1{+}1, a_2{-}1]\cdots[b_{n-1}{+}1, a_n{-}1][b_n{+}1, \infty] & \text{otherwise} \end{cases}$$

[2]These interval relations are reminiscent of the temporal interval algebra of Allen [1983]. Our definitions of "precedes" and "overlaps" are identical to Allen's, whereas "is before" corresponds to Allen's "precedes or meets," and our "contains" is equivalent to Allen's "contains or equals or is started by or is finished by."

The standard arithmetic operations are lifted into $\hat{\mathbf{Z}}$ via pointwise mapping of the equivalent operations on **IZ**. We assume a standard semantics for abstract integer expressions, allowing us to reduce and rewrite arithmetic expressions into a canonical form, e.g., $[3,5]+1 \rightsquigarrow [4,6]$.

We can construct the abstraction function $\alpha : \wp(\mathbb{Z}) \to \hat{\mathbf{Z}}$ for any finite subset of integers $X \in \wp(\mathbb{Z})$ by first sorting all elements of X in ascending order and then identifying all non-overlapping intervals of consecutive integers. This strategy does not work if X is infinite, and in any case is rather inefficient. Likewise, the concretization function $\gamma : \hat{\mathbf{Z}} \to \wp(\mathbb{Z})$, defined as

$$\gamma(x_1 x_2 \cdots x_n) = \bigcup_{i \le n} \gamma_{\mathbf{IZ}}(x_i),$$

is not very practical. Fortunately, for our use case we only need to abstract and concretize sets of integers defined via relational predicates, which is easily tractable, even with infinite bounds. The corresponding semantic functions are defined below, with $\mathbf{a}$ standing for a concrete integer.

$$[\![x = \mathbf{a}]\!]\uparrow_x \doteq [\mathbf{a},\mathbf{a}] \qquad [\![x \ge \mathbf{a}]\!]\uparrow_x \doteq [\mathbf{a},\infty] \qquad [\![x \le \mathbf{a}]\!]\uparrow_x \doteq [-\infty,\mathbf{a}]$$
$$[\![x \ne \mathbf{a}]\!]\uparrow_x \doteq [-\infty,\mathbf{a}-1][\mathbf{a}+1,\infty] \qquad [\![x > \mathbf{a}]\!]\uparrow_x \doteq [\mathbf{a}+1,\infty] \qquad [\![x < \mathbf{a}]\!]\uparrow_x \doteq [-\infty,\mathbf{a}-1]$$

6.4 Characters

Abstract characters (i.e., sets of possible characters) are a common component of string grammars—whitespace, for example, is usually defined as a set of certain invisible characters. While the size of any string alphabet is always bounded and thus the maximum number of possibilities for a single character is finite, these bounds can be quite large—the Unicode standard currently defines 154 998 characters [Unicode 16.0]. Additionally, it is often desirable to define elements of a string by what characters are *not* allowed to be there. Thus the need for an efficient abstract representation of large sets of (im)possible characters.

Formally, we can define the domain of abstract characters $\hat{\mathbf{C}}$ via the alphabet subset lattice $\langle \wp(\Sigma), \subseteq \rangle$, with the usual operations and elements $\langle \hat{\mathbf{C}}, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg \rangle$ (cf. abstract Booleans $\hat{\mathbf{B}}$). We use the familiar notation Σ interchangeably with $\top$, indicating the set of all characters of the alphabet. We use set difference to indicate exclusion, e.g., $\Sigma\backslash\{\mathsf{a},\mathsf{b}\}$ means the set of all characters excluding 'a' and 'b'. For singleton sets like $\{\mathsf{a}\}$ we will usually drop the braces and just write a. Note that $\bot = \varnothing$ is *not* equivalent to the empty string; rather, it is the empty set of characters, representing a space that is impossible to fill or a character that is impossible to produce. The abstract semantics of $\hat{\mathbf{C}}$ are defined by

$$\alpha = \gamma = \mathrm{id}, \qquad [\![x = \mathbf{c}]\!]\uparrow_x \doteq \{\mathbf{c}\}, \qquad [\![x \ne \mathbf{c}]\!]\uparrow_x \doteq \Sigma\backslash\{\mathbf{c}\}.$$

6.5 Strings

To abstractly represent infinite sets of strings, we define a domain $\hat{\mathbf{S}}$ of regular expressions over abstract characters $\hat{\mathbf{C}}$, consisting of the empty language $\varnothing$, the empty string ε, abstract character literals $\hat{c} \in \hat{\mathbf{C}}$ such that $\hat{c} = \{c_1, c_2, \ldots, c_n\} \equiv c_1 + c_2 + \cdots + c_n$, concatenation $\square \cdot \square$ (usually elided), alternation $\square + \square$, the Kleene star $\square^*$, and optionals $\square^? \equiv \varepsilon + \square$. We also allow the abbreviations $\square^+ = \square\square*$ and $\square^n = \square\square\cdots\square$ where $\square$ is repeated n times, with $n < 1$ resulting in $\varnothing$ and $\square^0 = \varepsilon$.

The domain $\hat{\mathbf{S}}$ forms a complete complemented lattice $\langle \hat{\mathbf{S}}, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg \rangle$ with $\bot = \varnothing$, $\top = \Sigma^*$, and the standard operations on regular sets [Hopcroft and Ullman 1979, chap. 3]. The language $L(\hat{s})$ describes all strings generated/recognized by a regular expression $\hat{s} \in \hat{\mathbf{S}}$, thus $\gamma(\hat{s}) = L(\hat{s})$. We can of course only abstract sets of strings that form a regular language, i.e., $\alpha(S) = \hat{s} \iff S = L(\hat{s})$. While α is not generally realizable [Gold 1967], we can define an abstract semantics over string relations, excerpted in Fig. 8, that allows us to abstract all strings expressed via relations between common string operations.

$$\square^{\hat{n}} = \bigsqcup_{[a,b]\in\hat{n}} \square^{[a,b]} \qquad \square^{[a,b]} = \begin{cases} \square^a\square^* & \text{if } b = \infty \\ \square^a + \square^{a+1} + \cdots + \square^b & \text{otherwise} \end{cases}$$

$$\begin{aligned} [\![|s| \between \hat{n}]\!]\uparrow_s &\doteq \Sigma^{\hat{n}} & [\![|s| \between \varnothing]\!]\uparrow_s &\doteq \varnothing \\ [\![s[\hat{\imath}] \between \hat{c}]\!]\uparrow_s &\doteq \Sigma^{\hat{\imath}}\hat{c}\Sigma^* & [\![s[\hat{\imath}] \between \varnothing]\!]\uparrow_s &\doteq \Sigma^{\hat{\imath}} \\ [\![s[|s| - \hat{\imath}] \between \hat{c}]\!]\uparrow_s &\doteq \Sigma^*\hat{c}\Sigma^{\hat{\imath}-1} & [\![s[|s| - \hat{\imath}] \between \varnothing]\!]\uparrow_s &\doteq \varnothing \\ [\![s[\hat{\imath}..\hat{\jmath}] \between \hat{t}]\!]\uparrow_s &\doteq \Sigma^{\hat{\imath}}(\Sigma^{\hat{\jmath}-\hat{\imath}+1} \sqcap \hat{t})\Sigma^* & [\![s[\hat{\imath}..\hat{\jmath}] \between \varnothing]\!]\uparrow_s &\doteq \Sigma^{\hat{\imath}} \end{aligned}$$

$$[\![\mathsf{str.indexof}(s, \hat{t}, \hat{k}) \between \hat{\imath}]\!]\uparrow_s \doteq \begin{cases} \Sigma^{\hat{k}}\Sigma^{\hat{\imath}-\hat{k}} & \text{if } \hat{t} = \varnothing \\ \Sigma^{\hat{k}}\neg(\Sigma^*\hat{t}\Sigma^*) + \Sigma^{\hat{k}}((\neg(\Sigma^*\hat{t}\Sigma^*)) \sqcap \Sigma^{\hat{\imath}-\hat{k}})\hat{t}\Sigma^* & \text{if } \min\hat{\imath} < 0 \\ \Sigma^{\hat{k}}((\neg(\Sigma^*\hat{t}\Sigma^*)) \sqcap \Sigma^{\hat{\imath}-\hat{k}})\hat{t}\Sigma^* & \text{otherwise} \end{cases}$$

Fig. 8. Abstract relational semantics for strings. Abstract values are denoted $\hat{\square}$. We assume all concrete values have been lifted to singleton sets, e.g., $s[i] = c \rightsquigarrow s[\hat{\imath}] \between \hat{c}$ where $\hat{\imath} = [i, i]$ and $\hat{c} = \{c\}$.

7 Implementation and Evaluation

We have implemented our approach in the Panini prototype system, available at

https://github.com/mcschroeder/panini

It includes a basic Python frontend with a small default set of λ_Σ axioms mimicking the semantics of common Python string functions. Our implementation is written in Haskell and uses the Z3 theorem solver [De Moura and Bjørner 2008] and is modular with respect to the abstract domains used during grammar inference. Panini can be used as a library, a standalone batch-mode command-line application, or interactively via a read-eval-print loop.

7.1 Efficient Regular Expressions

An important aspect for the practical viability of our approach is an efficient implementation of the underlying abstract domains, in particular abstract strings $\hat{\mathbf{S}}$ and abstract characters $\hat{\mathbf{C}}$. The machine representation of $\hat{\mathbf{C}}$ is based on complemented PATRICIA tries [Kmett 2012; Morrison 1968; Okasaki and Gill 1998], which enable compact representation of complemented sets while allowing all common set operations. For $\hat{\mathbf{S}}$, we implemented an efficient regular expression type whose literals are abstract characters from $\hat{\mathbf{C}}$ and whose operations, like regular intersection and complement, are performed purely algebraically, without going through finite automata. Our approach is based on Brzozowski [1964] derivatives [see also V. Antimirov 1996] and uses Arden's lemma and Gaussian elimination to solve systems of regular equations that model these operations [Acay 2018; Arden 1961]. To effectively compute precise derivatives over large alphabets, we use a local mintermization technique [Keil and Thiemann 2014]. To keep regular expressions concise, for readability as well as performance, we aggressively simplify intermediate terms using a set of rewrite rules and heuristics [Kahrs and Runciman 2022]. We also employ a variety of efficient algebraic decision procedures for regular language membership [Nipkow and Traytel 2014], inclusion (which admits a polynomial test in many practical situations) [Hovland 2012], and equivalence (via refutation) [Almeida et al. 2008; V. M. Antimirov and Mosses 1995].

7.2 Experiments

To provide insights into the efficacy and applicability of our approach, we used the Panini prototype to infer regular grammars for a number of ad hoc parser implementations.

Dataset. We created a benchmark dataset comprising 204 regular ad hoc parsers written in Python and then translated to C. The dataset is diverse across two dimensions: complexity of accepted grammar and structure of parser code. We generated the dataset by writing straightforward parsers for increasingly complex combinations of regular language operations and then added variations of each parser, e.g., using loops to iterate over the characters in a string, using high-level Python string functions such as `index`, parsing the input from back-to-front, and so on. Figures 2 and 9 to 12 show subjects from the benchmark dataset.

To facilitate a structured analysis, we classified the parsers within the dataset into three overarching categories based on their structural features: *Straight-Line Programs* are ad hoc parsers characterized by linear execution flow without branching constructs such as conditionals or loops (e.g., Fig. 9); *Programs with Conditionals* are ad hoc parsers that incorporate conditional statements to make decisions based on input characteristics (e.g., Fig. 2); and *Programs with Loops* are ad hoc parsers containing iterative constructs alongside conditional statements for string manipulation tasks (e.g., Fig. 10).

```
def getAddrSpec(email):
  b1 = email.index('<',0)+1
  b2 = email.index('>',b1)
  return email[b1:b2]
```

```
getAddrSpec = λemail : 𝕊.
  let v0 = index email "<" 0 in
  let v1 = ge v0 0 in
  let _ = assert v1 in
  let b1 = add v0 1 in
  let b2 = index email ">" b1 in
  let v3 = ge b2 0 in
  let _ = assert v3 in
  slice email b1 b2
```

Fig. 9. A straight-line ad hoc parser.

```
def lsb_check(s):
  i = 0
  while i < len(s)-1:
    assert s[i] == '0'
    i += 1
  assert s[i] == '1'
```

```
lsb_check = λs : 𝕊.
  rec L2 : {i : ℤ | ⋆} → 𝟙 = λi : ℤ.
    let v0 = length s in
    let v1 = sub v0 1 in
    let v2 = lt i v1 in
    if v2 then
      let v3 = slice1 s i in
      let v4 = match v3 "0" in
      let _ = assert v4 in
      let i = add i 1 in L2 i
    else
      let v5 = slice1 s i in
      let v6 = match v5 "1" in
      assert v6
  in L2 0
```

Fig. 10. An ad hoc parser with loops.

Methodology. For each ad hoc parser in the dataset, we used Panini to transpile the original Python source to λ_Σ and automatically infer a grammar from the parser's λ_Σ representation. We compared each inferred grammar G_i against a manually derived ground truth grammar $\hat{G}_i$. We differentiate between exact matches, where $L(G_i) = L(\hat{G}_i)$; successful under-approximations $L(G_i) \subset L(\hat{G}_i)$, which correctly identify a subset of allowed strings; trivial under-approximations to the empty language $L(G_i) = \varnothing$; and cases where Panini reports an error due to limitations of the prototype implementation or the underlying abstract domains. We additionally computed the *precision* and *recall* of each inferred grammar. Precision measures the percentage of inputs accepted by G_i that are also accepted by $\hat{G}_i$, whereas recall measures the pecentage of inputs accepted by $\hat{G}_i$ that are also accepted by G_i. We generate up to 1000 random sample inputs from the respective grammars to compute each measure. Low precision indicates over-approximation, i.e., the inferred grammar allows more inputs than would be safely accepted by the

Table 2. Results of evaluating Panini on a varied dataset of ad hoc parsers.

Type	Parser Example		#	Inferred Language =	⊂	∅	Error	SR	P	R	Time (s)
Straight	getAddrSpec	`[^<>]*<[^>]*>.*`	89	84	0	0	5	.94	1.00	1.00	0.16 ±0.27
+ Cond.	Figure 3	`a|[^a]b.*`	51	50	0	0	1	.98	1.00	1.00	0.09 ±0.05
+ Loops	lsb_check	`0*1`	64	40	7	7	10	.84	1.00	.76	2.32 ±4.77
			204	174	7	7	16	.92	1.00	.93	0.82 ±2.85

SR = success rate, P = precision, R = recall

parser program, while low recall indicates under-approximation, i.e., the inferred grammar is unnecessarily stricter than the actual program.

We aggregated precision and recall across each of the three parser categories and additionally report the *success rate*, the percentage of benchmark subjects for which Panini is able to infer a grammar. Finally, we also report average wall-clock execution time. All benchmarks were run on a MacBook Pro with an Apple M4 processor and 24 GB RAM, using Z3 4.8.10 for SMT solving.

Results. Table 2 presents our experimental results. Out of the 204 parser programs in our dataset, Panini is able to successfully infer a grammar for 188, a success rate of 92 %. As expected, the precision of inferred grammars is 100 % across all types of parsers: *Panini never over-approximates.* The average overall recall is 93 %. Panini achieves exact inference for all straight-line and purely conditional programs and for 40 programs containing loops; it safely under-approximates the grammar of 7 loop programs; and it fails to find a meaningful refinement beyond the empty language for another 7 loop programs.

The performance of grammar inference is mostly in the sub-second range, with some loop programs as outliers. Most of the inference time is spent during classical refinement inference, where SMT solving is still the biggest bottleneck.

7.3 Comparison with Other Approaches

Table 3 presents a comparison of Panini and other grammar inference systems. We compare operating requirements (whether the parser's source code needs to be available; whether the parser needs to be executed in some way, perhaps instrumented or modified; and whether the approach requires pre-existing input samples), the type of output (a regular expression, a context-free grammar, an automaton), and the results of running the approach on the Panini benchmark dataset (see § 7.2). The systems and algorithms selected for comparison represent the state-of-the-art in grammar inference:

Exbar [Lang 1999] is the fastest known algorithm for finding minimal DFAs from labeled samples only. This type of grammar inference—known as *passive automata learning*—is notable in that it does not require the existence of a parser program at all, neither to run nor inspect. Other prominent algorithms in this category include RPNI [Oncina and García 1992] and the approximative ED-BEAM [Lang 1999]. Unfortunately, the requirement of a well-labeled set of representative input samples is unrealistic in many settings, including ours.

TTT [Isberner 2015; Isberner et al. 2014] is a leading algorithm in *active automata learning*, specifically within the *minimally adequate teacher* (MAT) framework. Established by Angluin [1987] with the introduction of the seminal L^* algorithm, MAT formulates grammar learning as an interactive process, in which a teacher—an oracle or black-box parser program—can answer two types of question: whether a certain input is a *member* of the target language, and whether a hypothesized language is *equivalent* to the target language, providing a counterexample if it is not.

Table 3. Comparison of state-of-the-art grammar inference approaches.

	Requirements				Panini benchmark			
Approach	Source	Execution	Samples	Output	SR	P	R	Time (s)
Panini	Python or λ_Σ	-	-	Regex	.92	1.00	.93	0.82 ±2.85
STALAGMITE	C	symbolic	-	CFG	.96	.87	.73	80.01 ±178.84
Mimid	Python or C	traced	positive	CFG	.67	.97	.81	27.39 ±42.78
TreeVada	-	oracle	positive	CFG	.98	.99	.66	72.63 ±100.84
TTT	-	oracle	positive	DFA	.99	.13	1.00	2.74 ±15.17
Exbar	-	-	pos. + neg.	DFA	-	-	-	-

In practice, the requirement of equivalence queries is quite onerous, which is why they are usually approximated by conformance testing [Aichernig et al. 2024] using known positive input samples.

When running TTT on the Panini benchmark, we generated up to 10 000 positive input samples for each parser, simulating a best-case scenario. The results (13 % average precision, 100 % recall) indicate that TTT tends to significantly over-approximate the true input language for these kinds of ad hoc parsers, even with a very large number of input samples.

TreeVada [Arefin et al. 2024] is the state-of-the-art in black-box inference of context-free grammars. Other approaches in this vein are Arvada [Kulkarni et al. 2021] and the pioneering Glade [Bastani et al. 2017]. These systems do not require equivalence queries, which makes them much more practical than traditional MAT algorithms, but they do all require a well-covering set of positive input samples in order to produce accurate grammars [Bendrissou et al. 2022].

When given up to 20 positive input samples per parser in the Panini benchmark, TreeVada achieves near-perfect 99 % precision (on average) but under-approximates the true grammars with only 66 % average recall. We found that doubling the amount of input samples improves recall by less than 10 % while more than doubling the runtime.

Mimid [Gopinath et al. 2020a] generalizes positive sample inputs into a context-free grammar by analyzing execution traces of an instrumented version of the parser, which needs to be available in source form. *Mimid* significantly improves on the previous white-box approach AUTOGRAM [Höschele and Zeller 2016], but it still requires a good set of pre-existing input samples to produce an accurate grammar. A general advantage of white-box approaches is that the inferred grammars tend to be very readable, because they can incorporate identifiers from the source code.

On the Panini benchmark, with up to 1000 positive sample inputs for each parser, *Mimid* produces grammars of reasonably high precision (97 % on average) but tends to under-approximate (81 % average recall). It also has a low success rate of only 67 %, meaning *Mimid* failed to infer any grammar for a third of the parsers in the dataset.

STALAGMITE [Bettscheider and Zeller 2024] is a recent white-box approach that obviates the need for input samples by transforming the source program into a version amenable for symbolic execution. In addition to inserting tracing calls, this includes limiting recursion depth and the number of loop iterations, which enables a symbolic test generator like KLEE [Cadar, Dunbar, et al. 2008] to automatically generate input samples that cover all execution paths. After running the modified parser on a large enough number of samples, the collected symbolic input traces are woven together to produce a context-free grammar.

STALAGMITE exhibits mediocre performance on the Panini benchmark, producing grammars that are both imprecise (87 % average precision) and under-approximated (73 % average recall). While it does not require any pre-existing input samples, the symbolic execution of the parser programs is quite resource intensive.

Note that **Panini** is the only approach that requires neither positive input samples nor any interaction with the parser. It is therefore uniquely suited to deal with ad hoc parsers, for which input samples are generally not available and which occur as code fragments that cannot always be expected to run as-is. In its current form, Panini is limited to inference of at-most regular languages, but we conjecture that these make up the highest share of ad hoc parsers in the wild [Schröder, Goritschnig, et al. 2023]. We argue that all other existing approaches have requirements that make them impractical to use in this setting, including prohibitive resource usage.

7.4 Real-World Case Studies

To investigate Panini's suitability for real-world grammar inference, we look at two regular-language ad hoc parsers from the *Mimid* benchmark suite [Gopinath et al. 2020b].

cgidecode.py. This is a Python program to decode CGI-encoded strings, an encoding where spaces are replaced by plus signs and other invalid characters are replaced by a hexadecimal encoding prefixed with a percent sign. If an improperly encoded string is encountered, the program raises an error. The regular expression `([^%]|%[0-9A-Fa-f][0-9A-Fa-f])*` encompasses the input language of this ad hoc parser.[3] As Table 4 shows, all approaches do fairly well on precision. TreeVada exhibits the lowest recall, likely due to prioritizing larger-scale inference of context-free properties, to the detriment of regular language accuracy. Panini, STALAGMITE, and TTT all achieve perfect precision and recall. TTT does so with the help of 10 000 known positive input samples, while STALAGMITE requires a one-off symbolic execution harness specifically tailored to cgidecode.py. Only Panini can infer an exact grammar without having access to any pre-existing knowledge about the parser program or its possible inputs.

Table 4. cgidecode.py benchmark

	P	R	Time (s)
Panini	1.00	1.00	1.49
STALAGMITE	1.00	1.00	58.52
Mimid	.90	.96	139.07
TreeVada	.96	.60	288.40
TTT	1.00	1.00	3.60

urlparse.py. This is the URL parser part of the Python *urllib* library. Even though this parser is quite large and complex (>1000 LoC), it is written very defensively. The only time it actually rejects an input is when the network location part of an otherwise valid URL contains an insufficiently bracketed IPv6 address. The parser's input language is equivalent to the regular expression `(([a-zA-Z0-9.+-]+:)?//([^/?#]*(\[[^/?#]*[」]]|[]][^/?#]*\[).*|[^][]*([/?#].*)?))|(.*([^:]」|[^a-zA-Z0-9.+-].*)//.*)`, which looks somewhat baroque but is surprisingly permissive.[4] This proves to be a difficult case for grammar inference: As shown in Table 5, TreeVada and TTT over-approximate the correct grammar to the extent that they achieve no precision; *Mimid* achieves perfect precision at the cost of severely under-approximating the true grammar; and neither STALAGMITE nor Panini are able to infer any grammar at all. STALAGMITE requires C source code, which does not exist for this program, while Panini's Python frontend is not yet capable of automatically translating this syntactically complex parser into λ_Σ.

Table 5. urlparse.py benchmark

	P	R	Time (s)
Panini	-	-	-
STALAGMITE	-	-	-
Mimid	1.00	0.19	157.00
TreeVada	.00	0.96	625.35
TTT	.00	1.00	1.75

[3]The golden grammar included by the *Mimid* artifact is insufficient: it does not include all possible two-digit hexadecimals and is limited to a subset of the ASCII alphabet.

[4]This is again significantly different from the golden grammar included by the *Mimid* artifact, which is much more restrictive, yet at the same time does not prevent the parser's actual error case.

7.5 Current Limitations and Future Work

Our Panini prototype is a proof-of-concept that shows the viability of our approach; it is not yet a practical end-user system. Based on the results of our evaluation, including the two case studies, we see the following main areas of improvement as part of future work:

Parser Extraction. Panini is built around λ_Σ, a calculus for representing ad hoc parsers. While the present work focuses on synthesizing grammars from λ_Σ programs, a significant part of our envisioned grammar inference process is the extraction of λ_Σ from general-purpose programming languages (cf. § 2.1). The Panini prototype does include some machinery to automatically extract and transpile ad hoc parser slices from Python to λ_Σ, but this is still very limited. We are currently experimenting with different parser slicing techniques and continue to improve our Python front end. We are also hoping to add prototypical support for other languages in the near future.

Language Features. The λ_Σ language is by design minimal, in order to provide a common intermediate representation for a wide range of ad hoc parser implementations and to simplify many aspects of refinement and grammar inference. However, its lack of language features makes it difficult to easily capture many real-world parser programs, such as those involving generic data types. We are currently working on extending the λ_Σ language to add support for polymorphism and user-defined data types.

Invariant Inference. In our approach, the solving of non-grammar κ variables is delegated to the classical refinement inference machinery (§ 3.3). In particular, this includes the generation of loop/recursion invariants, which our prototype infers using a textbook implementation of purely conjunctive predicate abstraction, a technique that is inherently limited in the shape of invariants that can be produced and is highly dependent on a good set of candidate predicates; it is also a runtime bottleneck. We plan to improve our qualifier extraction heuristics and integrate external state-of-the-art invariant generators.

Abstract Domains. Abstract interpretation is bounded by the limits of the underlying abstract domains (§ 6). For example, our integer domain $\hat{\mathbf{Z}}$ cannot efficiently represent congruence classes, e.g., infinite sets of even or odd numbers. This can lead to under-approximations, as in Fig. 11, where Panini can only infer the subset `(ab)?` of the true grammar `(ab)*`. By design, our system is modular in the choice of underlying abstract domains, and we are working on extending them.

```
def f250(s):
  i = 0
  while i < len(s):
    assert s[i] == "a"
    assert s[i+1] == "b"
    i += 2
```

Fig. 11. A parser for `(ab)*`

Relational Semantics. If Panini lacks the semantics to rewrite, normalize, or abstract a particular relation (see § 5), it will eventually become stuck. We can increase the capabilities of the system by extending the set of semantic rules—i.e., adding more equations to Fig. 8. But take the parser in Fig. 12, which leads to Panini trying to compute the abstraction $[\![s[0] = s[1]]\!]\uparrow_s$. In isolation, this constraint is not even expressible in a regular string domain—even though the final grammar is regular. To handle such tricky cases, a more complex non-local rewriting strategy might be needed.

```
def f521(s: str):
  assert s[0] == "a"
  assert s[1] == s[0]
```

Fig. 12. A parser for `aa.*`

Context-Free Grammars. If Panini encounters a context-free parser, it will likely yield $\emptyset$ or get stuck, since the underlying string domain is built on regular expressions and cannot represent context-free constructs. Non-trivial recursion exhibited by a parser might impede invariant inference, however we do not see any limitations inherent to our technique that would in principle prevent us from eventually inferring (deterministic) context-free grammars.

8 Related Work

String Constraint Solving. Precise formal reasoning over strings can be accomplished using *string constraint solving* (SCS), a declarative paradigm of modeling relations between string variables and solving attendant combinatorial problems [Amadini 2023]. It is usually assumed that *collecting* string constraints requires some kind of (dynamic) symbolic execution [Kausler and Sherman 2014], with practical applications ranging from generating test inputs [Bjørner, Tillmann, et al. 2009; Cadar, Ganesh, et al. 2008; Li et al. 2011] to detecting vulnerabilities (e.g., cross-site scripting, SQL injection) [Bultan et al. 2018; Holík et al. 2018; Loring et al. 2017; Saxena et al. 2010]. The latter type of work is concerned with the inverse of our problem: modeling the possible strings a function can return or express, instead of the strings a function can accept.

In our approach, we use purely static means (viz. refinement type inference) to collect input string constraints (see § 3), which we then simplify/solve in ways not dissimilar but nonetheless different from traditional SCS techniques (see § 4). Our implementation makes some use of the (incomplete) string theories embedded in the Z3 constraint solver [De Moura and Bjørner 2008], whose current version unfolds regular membership relations using symbolic derivatives [Stanford et al. 2021]. Recent advances in SCS for SMT promise stronger procedures that support checking more complex functions over strings and regular expressions [Berzish et al. 2021; Blahoudek et al. 2023; Y.-F. Chen, Havlena, Hečko, et al. 2025; Y.-F. Chen, Havlena, Lengál, et al. 2020; T. Chen et al. 2019; Day, Ehlers, et al. 2019; Day, Kulczynski, et al. 2020; Hague et al. 2025; Kan et al. 2022; Kulczynski et al. 2022; Lotz et al. 2023; Mora et al. 2021; Trinh et al. 2020].

Abstract Domains. Abstract string domains approximate strings to track information precisely enough to analyze particular behaviors while only preserving relevant information. Most of the existing work in string domains differs in what kind of behavior is of interest and how the approximation is achieved efficiently. Costantini et al. [2015] introduce a suite of different abstract semantics for concatenation, character inclusion, and substring extraction (particularly pre- and suffixes). In their work, they explicitly discuss the trade-off between precision and efficiency. Amadini, Gange, et al. [2020] review the dashed string abstraction, an approach that considers strings as blocks of characters and the constraints on these blocks, which exhibits good performance on constraints involving length, equality, concatenation, and regular membership. M-String [Cortesi and Olliaro 2018] considers a parametric abstract domain for strings in the C programming language, extending the array segmentation approach of P. Cousot, R. Cousot, and Logozzo [2011]. Arceri et al. [2022] focus on relational string domains that try to capture the relation between string variables and expressions for which we cannot compute static values, such as user input. Tsaris [Negrini, Arceri, Cortesi, et al. 2024; Negrini, Arceri, Ferrara, et al. 2021] is an automata-based domain that approximates string values through finite automata over an alphabet of whole strings instead of single characters, obtaining more precise results than other approaches.

There have been a multitude of abstract domains that aim at strings in specific target languages, e.g., JavaScript [Amadini, Jordan, et al. 2017; Jensen et al. 2009; Kashyap et al. 2014; Park et al. 2016].

Precondition Inference. Input grammars are essentially preconditions for parsers, thus grammar inference can be viewed as a special case of *precondition inference*. There exist a wide variety of approaches for computing various kinds of preconditions: weakest precondition inference for unstructured programs [Barnett and Leino 2005], inference of necessary preconditions using abstract interpretation [P. Cousot, R. Cousot, Fähndrich, et al. 2013], abductive inference for inductive invariants [Dillig et al. 2013], counterexample-guided inference [Seghir and Kroening 2013], data-driven inference [Padhi et al. 2016], etc. We are not aware of any approaches that focus specifically on string operations, or that would enable the reconstruction of an input grammar in a way suitable for the envisioned applications.

Acknowledgments

The authors would like to thank Leon Bettscheider for providing support in evaluating the latest version of STALAGMITE. This research was funded by the Austrian Science Fund (FWF) under grant 10.55776/PIN3275223 (https://doi.org/10.55776/PIN3275223).

Data-Availability Statement

The latest version of the PANINI system is available at https://github.com/mcschroeder/panini. This repository also includes our full evaluation dataset. We additionally provide a self-contained artifact at https://doi.org/10.5281/zenodo.15732005 to easily reproduce our claims.

References

Coşku Acay. May 22, 2018. "A Regular Expression Library for Haskell." Unpublished manuscript and source code. (May 22, 2018). https://github.com/cacay/regexp.

Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. Mar. 2024. "Benchmarking Combinations of Learning and Testing Algorithms for Automata Learning." *Formal Aspects of Computing*, 36, 1, Article 3, (Mar. 2024). doi:10.1145/3605360.

James F. Allen. Nov. 1983. "Maintaining Knowledge about Temporal Intervals." *Communications of the ACM*, 26, 11, (Nov. 1983), 832–843. doi:10.1145/182.358434.

Marco Almeida, Nelma Moreiara, and Rogério Reis. 2008. *Testing the equivalence of regular expressions*. Tech. rep. DCC-2007-07. Version 1.1. Universidade do Porto. https://www.dcc.fc.up.pt/Pubs/TR07/dcc-2007-07.pdf.

Roberto Amadini. Jan. 2023. "A Survey on String Constraint Solving." *ACM Computing Surveys*, 55, 1, Article 16, (Jan. 2023). doi:10.1145/3484198.

Roberto Amadini, Graeme Gange, and Peter J. Stuckey. Dec. 2020. "Dashed strings for string constraint solving." *Artificial Intelligence*, 289, Article 103368, (Dec. 2020). doi:10.1016/j.artint.2020.103368.

Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald SØndergaard, Peter J. Stuckey, and Chenyi Zhang. 2017. "Combining String Abstract Domains for JavaScript Analysis: An Evaluation." In: *Tools and Algorithms for the Construction and Analysis of Systems. 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Part I*. Springer, 41–57. doi:10.1007/978-3-662-54577-5_3.

Dana Angluin. Nov. 1987. "Learning Regular Sets from Queries and Counterexamples." *Information and Computation*, 75, 2, (Nov. 1987), 87–106. doi:10.1016/0890-5401(87)90052-6.

Valentin Antimirov. Mar. 1996. "Partial derivatives of regular expressions and finite automaton constructions." *Theoretical Computer Science*, 155, 2, (Mar. 1996), 291–319. doi:10.1016/0304-3975(95)00182-4.

Valentin M. Antimirov and Peter D. Mosses. July 10, 1995. "Rewriting extended regular expressions." *Theoretical Computer Science*, 143, 1, (July 10, 1995), 51–72. doi:10.1016/0304-3975(95)80024-4.

Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Pietro Ferrara. 2022. "Relational String Abstract Domains." In: *Verification, Model Checking, and Abstract Interpretation. 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings*. Springer, 20–42. doi:10.1007/978-3-030-94583-1_2.

Dean N. Arden. Sept. 1961. "Delayed-Logic and Finite-State Machines." In: *Switching Circuit Theory and Logical Design*. Proceedings of the Second Annual Symposium and Papers from the First Annual Symposium. SWCT 1961, Detroit, MI, USA, October 17–20, 1961 / SWCT 1960, Chicago, IL, USA, October 9–14, 1960. American Institute of Electrical Engineers, (Sept. 1961), 133–151. doi:10.1109/FOCS.1961.13.

Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, and Christoph Csallner. 2024. "Fast Deterministic Black-box Context-free Grammar Inference." In: *ICSE'24. Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* Article 117 (Lisbon, Portugal, Apr. 14–20, 2024). ACM. doi:10.1145/3597503.3639214.

Mike Barnett and K. Rustan M. Leino. 2005. "Weakest-Precondition of Unstructured Programs." In: *PASTE'05. Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Sftware Tools and Engineering* (Lisbon, Portugal, Sept. 5–6, 2005). ACM, 82–87. doi:10.1145/1108792.1108813.

Clark Barret, Pascal Fontaine, and Cesare Tinelli. 2016. *The Satisfiability Modulo Theories Library (SMT-LIB)*. (2016). https://www.smt-lib.org.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. "Synthesizing Program Input Grammars." In: *PLDI'17. Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain, June 18–23, 2017). ACM, 95–110. doi:10.1145/3062341.3062349.

Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. "'Synthesizing Input Grammars': A Replication Study." In: *PLDI '22. Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA, June 13, 2021–June 17, 2020). ACM, 260–268. doi:10.1145/3519939.3523716.

Murphy Berzish, Mitja Kulczynski, Federico Mora, Florin Manea, Joel D. Day, Dirk Nowotka, and Vijay Ganesh. 2021. "An SMT Solver for Regular Expressions and Linear Arithmetic over String Length." In: *Computer Aided Verification. 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Springer, 289–312. doi:10.1007/978-3-030-81688-9_14.

Leon Bettscheider and Andreas Zeller. 2024. "Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing." In: *FSE Companion '24. Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil, July 15–19, 2024). ACM, 522–526. doi:10.1145/3663529.3663790.

Saroja Bhate and Subhash Kak. 1991–1992. "Pāṇini's Grammar and Computer Science." *Annals of the Bhandarkar Oriental Research Institute*, 72/73, 1–4, 79–94, Amṛtamahotsava Volume. https://www.jstor.org/stable/41694883.

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. "Horn Clause Solvers for Program Verification." In: *Fields of Logic and Computation II. Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte. Springer, 24–51. doi:10.1007/978-3-319-23534-9_2.

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. "Path Feasibility Analysis for String-Manipulating Programs." In: *Tools and Algorithms for the Construction and Analysis of Software. 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferenes on Theory and Practice of Software, ETAPS 2009, York, UK, March 2009, Proceedings*. Springer, 307–321. doi:10.1007/978-3-642-00768-2_27.

František Blahoudek, David Chen Yu-Fangand Chocholatý, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Juraj Síč. 2023. "Word Equations in Synergy with Regular Constraints." In: *Formal Methods. 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*. Springer, 403–423. doi:10.1007/978-3-031-27481-7_23.

William J. Bowman. June 30, 2022. "The A Means A," (June 30, 2022). https://www.williamjbowman.com/blog/2022/06/30/the-a-means-a/.

Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. "Simple and Efficient Construction of Static Single Assignment Form." In: *Compiler Construction. 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013, Proceedings*. Springer, 102–122. doi:10.1007/978-3-642-37051-9_6.

Janusz A. Brzozowski. Oct. 1964. "Derivatives of Regular Expressions." *Journal of the ACM*, 11, 4, (Oct. 1964), 481–494. doi:10.1145/321239.321249.

Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2018. *String Analysis for Software Verification and Security*. Springer. doi:10.1007/978-3-319-68670-7.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)* (San Diego, CA, USA, Dec. 8–10, 2008). USENIX Association, 209–224. http://usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Dec. 2008. "EXE: Automatically generating inputs of death." *ACM Transactions on Information and System Security*, 12, 2, Article 10, (Dec. 2008). doi:10.1145/1455518.1455522.

Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. Apr. 2004. "A Functional Perspective on SSA Optimisation Algorithms." *Electronic Notes in Theoretical Computer Science*, 82, 2, (Apr. 2004), 347–361. doi:10.1016/S1571-0661(05)82596-4.

Yu-Fang Chen, Vojtěch Havlena, Michal Hečko, Lukáš Holík, and Ondřej Lengál. June 2025. "A Uniform Framework for Handling Position Constraints in String Solving." *Proceedings of the ACM on Programming Languages*, 9, PLDI, Article 169, (June 2025). doi:10.1145/3729273.

Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. 2020. "A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving." In: *Programming Languages and Systems. 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings*. Springer, 343–363. doi:10.1007/978-3-030-64437-6_18.

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. Jan. 2019. "Decision Procedures for Path Feasibility of String-Manipulating Programs with Complex Operations." *Proceedings of the ACM on Programming Languages*, 3, POPL, Article 49, (Jan. 2019). doi:10.1145/3290362.

Agostino Cortesi and Martina Olliaro. 2018. "M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs." In: *TASE 2018. 2018 12th International Symposium on Theoretical Aspects of Software Engineering*. Proceedings (Guangzhou, China, Aug. 29–31, 2018). IEEE Computer Society, 1–8. doi:10.1109/TASE.2018.00009.

Benjamin Cosman and Ranjit Jhala. Aug. 2017. "Local Refinement Typing." *Proceedings of the ACM on Programming Languages*, 1, ICFP, Article 26, (Aug. 2017). doi:10.1145/3110270.

Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Feb. 2015. "A Suite of Abstract Domains for Static Analysis of String Values." *Software: Practice and Experience*, 45, 2, (Feb. 2015), 245–287. doi:10.1002/spe.2218.

Patrick Cousot. 1997. "Types as Abstract Interpretations." In: *POPL'97. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, Jan. 15–17, 1997). ACM, 316–331. doi:10.1145/263699.263744.

Patrick Cousot and Radhia Cousot. 1977. "Abstract Interpretation. A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *POPL'77. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, CA, USA, Jan. 17–19, 1977). ACM, 238–252. doi:10.1145/512950.512973.

Patrick Cousot and Radhia Cousot. 1979. "Systematic Design of Program Analysis Frameworks." In: *POPL'79. Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, TX, USA, Jan. 29–31, 1979). ACM, 269–282. doi:10.1145/567752.567778.

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. "Automatic Inference of Necessary Preconditions." In: *Verification, Model Checking, and Abstract Interpretation. 14th International Conference, VMCAI 2013, Rome, Italy, January 20–22, 2013, Proceedings*. Springer, 128–148. doi:10.1007/978-3-642-35873-9_10.

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. "A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis." In: *POPL'11. Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austrin, TX, USA, Jan. 26–28, 2011). ACM, 105–118. doi:10.1145/1926385.1926399.

Patrick Cousot and Nicolas Halbwachs. 1978. "Automatic Discovery of Linear Restraints Among Variables of a Program." In: *POPL'78. Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, AZ, USA, Jan. 23–25, 1978). ACM. doi:10.1145/512760.512770.

Luis Damas and Robin Milner. 1982. "Principal Type-Schemes for Functional Programs." In: *POPL'82. Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium in Principles of Programming Languages* (Albuquerque, NM, USA, Jan. 25–27, 1982). ACM, 207–212. doi:10.1145/582153.582176.

Joel D. Day, Thorsten Ehlers, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. 2019. "On Solving Word Equations Using SAT." In: *Reachability Problems. 13th International Conference, RP 2019, Brussels, Belgium, September 11–13, 2019, Proceedings*. Springer, 93–106. doi:10.1007/978-3-030-30806-3_8.

Joel D. Day, Mitja Kulczynski, Florin Manea, Dirk Nowotka, and Danny Bøgsted Poulsen. 2020. "Rule-based Word Equation Solving." In: *FormaliSE 2020. 2020 IEEE/ACM 8th International Conference on Formal Methods in Software Engineering*. Proceedings (Virtual, July 13, 2020). ACM, 87–97. doi:10.1145/3372020.3391556.

Leonardo De Moura and Nikolaj Bjørner. 2008. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems. 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29–April 6, 2008, Proceedings*. Springer, 337–340. doi:10.1007/978-3-540-78800-3_24.

Işil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. "Inductive Invariant Generation via Abductive Inference." In: *OOPSLA'13. The Proceedings of the 2013 International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, IN, USA, Oct. 29–31, 2013). ACM, 443–456. doi:10.1145/2509136.2509511.

Aryaz Eghbali and Michael Pradel. 2020. "No Strings Attached: An Empirical Study of String-related Software Bugs." In: *2020 35th IEEE/ACM International Conference on Automated Software Engineering*. Proceedings. ASE 2020 (Virtual, Sept. 22–25, 2020). ACM, 956–967. doi:10.1145/3324884.3416576.

Mike J. Fitter and Thomas R. G. Green. Mar. 1979. "When do diagrams make good computer languages?" *International Journal of Man-Machine Studies*, 11, 2, (Mar. 1979), 235–261. doi:10.1016/S0020-7373(79)80019-X.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. "The Essence of Compiling with Continuations." In: *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Proceedings. PLDI 1993 (Albuquerque, NM, USA, June 21–25, 1993). ACM, 237–247. doi:10.1145/155090.155113.

Roberto Giacobazzi and Elisa Quintarelli. 2001. "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking." In: *Static Analysis. 8th International Symposium, SAS 2001, Paris, France, July 16–18, 2001, Proceedings*. Springer, 356–373. doi:10.1007/3-540-47764-0_20.

David J. Gilmore and Thomas R. G. Green. July 1984. "Comprehension and recall of miniature programs." *International Journal of Man-Machine Studies*, 21, 1, (July 1984), 31–48.

E. Mark Gold. May 1967. "Language identification in the limit." *Information and Control*, 10, 5, (May 1967), 447–474. doi:10.1016/S0019-9958(67)91165-5.

Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020a. "Mining Input Grammars from Dynamic Control Flow." In: *ESEC/FSE '20. Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual, Nov. 8–13, 2020). ACM, 172–183. doi:10.1145/3368089.3409679.

Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020b. *Replication package for Mining Input Grammars from Dynamic Control Flow*. (2020). doi:10.5281/zenodo.3876969.

Matthew Hague, Denghang Hu, Artur Jeż, Anthony W. Lin, Oliver Markgraf, Philipp Rümmer, and Zhilin Wu. Aug. 15, 2025. "OSTRICH2: Solver for Complex String Constraints," (Aug. 15, 2025). arXiv preprint. doi:10.48550/arXiv.2506.14363.

J. Roger Hindley. Dec. 1969. "The Principal Type-Scheme of an Object in Combinatory Logic." *Transactions of the American Mathematical Society*, 146, (Dec. 1969), 29–60. doi:10.1090/S0002-9947-1969-0253905-6.

Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. "Grammarinator: A Grammar-Based Open Source Fuzzer." In: *A-TEST '18. Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA, Nov. 5, 2018). ACM, 45–48. doi:10.1145/3278186.3278193.

Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. Jan. 2018. "String Constraints with Concatenation and Transducers Solved Efficiently." *Proceedings of the ACM on Programming Languages*, 2, POPL, Article 4, (Jan. 2018). doi:10.1145/3158092.

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. "Fuzzing with Code Fragments." In: *Proceedings of the 21st USENIX Security Symposium* (Bellevue, WA, USA, Aug. 8–19, 2012). USENIX Association, 445–458. https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final73.pdf.

John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley. ISBN: 0-201-02988-X.

Matthias Höschele and Andreas Zeller. 2016. "Mining Input Grammars from Dynamic Taints." In: *ASE'16. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Sept. 3–7, 2016). ACM, 720–725. doi:10.1145/2970276.2970321.

Dav Hovland. Nov. 2012. "The inclusion problem for regular expressions." *Journal of Computer and System Sciences*, 78, 6, (Nov. 2012), 1795–1813. doi:10.1016/j.jcss.2011.12.003.

Malte Isberner. 2015. "Foundations of Active Automate Learning: An Algorithmic Perspective." Ph.D. Dissertation. TU Dortmund. doi:10.17877/DE290R-16359.

Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning." In: *Runtime Verification. 5th International Conference, RV 2014, Toronto, ON, Canada, September 22–25, 2014, Proceedings*. Springer, 307–322. doi:10.1007/978-3-319-11164-3_26.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. "Type Analysis for JavaScript." In: *Static Analysis. 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009, Proceedings*. Springer, 238–255. doi:10.1007/978-3-642-03237-0_17.

Ranjit Jhala and Niki Vazou. Oct. 15, 2020. "Refinement Types. A Tutorial," (Oct. 15, 2020). arXiv preprint. doi:10.48550/arXiv.2010.07763.

Ed. by Andrew Hume and Doug McIlroy. "Yacc: A Parser Generator." *UNIX Research System*. Vol. 2: *UNIX Research System Papers*. (10th ed.). AT&T Bell Labs, 347–374. ISBN: 0-03-047529-5.

Stefan Kahrs and Colin Runciman. Mar.–Apr. 2022. "Simplifying regular expressions further." *Journal of Symbolic Computation*, 109, (Mar.–Apr. 2022), 124–143. doi:10.1016/j.jsc.2021.08.003.

Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. 2022. "CertiStr: A Certified String Solver." In: *CPP '22. Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA, Jan. 17–18, 2022). ACM, 210–224. doi:10.1145/3497775.3503691.

Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel Böhme. 2022. "Human-in-the-Loop Oracle Learning for Semantic Bugs in String Processing Programs." In: *ISSTA '22. Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, July 18–22, 2022). ACM, 215–226. doi:10.1145/3533767.3534406.

Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. "JSAI: A Static Analysis Platform for JavaScript." In: *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. Proceedings (Hong Kong, China, Nov. 16–21, 2014), 121–132. doi:10.1145/2635868.2635904.

Scott Kausler and Elena Sherman. 2014. "Evaluation of String Constraint Solvers in the Context of Symbolic Execution." In: *ASE'14. Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Västerås, Sweden, Sept. 15–19, 2014). ACM, 259–270. doi:10.1145/2642937.2643003.

Matthias Keil and Peter Thiemann. 2014. "Symbolic Solving of Extended Regular Expression Inequalities." In: *34th International Conference on Foundation of Software Technology and Theoretical Computer Science*. FSTTCS 2014 (New Delhi, India, Dec. 15–17, 2014). Schloss Dagstuhl—Leibniz-Zentrum für Informatik, 175–186. doi:10.4230/LIPIcs.FSTTCS.2014.175.

Edward Kmett. 2012. *charset: Fast unicode character sets based on complemented PATRICIA tries*. source code. (2012). https://hackage.haskell.org/package/charset.

Mitja Kulczynski, Kevin Lotz, Dirk Nowotka, and Danny Bøgsted Poulsen. 2022. "Solving String Theories Involving Regular Membership Predicates Using SAT." In: *Model Checking Software. 28th International Symposium, SPIN 2022, Virtual Event, May 21, 2022, Proceedings*. Springer, 134–151. doi:10.1007/978-3-031-15077-7_8.

Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. "Learning Highly Recursive Input Grammars." In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering*. Proceedings. ASE 2021 (Virtual, Nov. 15–19, 2021). IEEE Computer Society, 456–467. doi:10.1109/ASE51524.2021.9678879.

Kevin J Lang. Dec. 12, 1999. *Faster Algorithms for Finding Minimal Consistent DFAs*. Tech. rep. NEC Research Institute, (Dec. 12, 1999). https://citeseerx.ist.psu.edu/document?doi=f74c5462cec67439490bf73f652ecd7d5f3f2679.

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Tech. rep. UU-CS-2001-35. Utrecht University. https://ics-archive.science.uu.nl/research/techreps/repo/CS-2001/2001-35.pdf.

Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. "KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs." In: *Computer Aided Verification. 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 2011, Proceedings*. Springer, 609–615. doi:10.1007/978-3-642-22110-1_49.

Francesco Logozzo and Manuel Fähndrich. Sept. 2010. "Pentagons: A weakly relational abstract domain for the efficient validation of array accesses." *Science of Computer Programming*, 75, 9, (Sept. 2010), 796–807. doi:10.1016/j.scico.2009.04.004.

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. "ExpoSE: Practical Symbolic Execution of Standalone JavaScript." In: *SPIN'17. Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software* (Santa Barbara, CA, USA, July 13–14, 2017). ACM, 196–199. doi:10.1145/3092282.3092295.

Kevin Lotz, Amit Goel, Bruno Dutertre, Benjamin Kiesl-Reiter, Soonho Kong, Rupak Majumdar, and Dirk Nowotka. 2023. "Solving String Constraints Using SAT." In: *Computer Aided Verification. 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*. Springer, 187–208. doi:10.1007/978-3-031-37703-7_9.

Falcon Darkstar Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. 2016. "The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them." In: *2016 IEEE Cybersecurity Development*. Proceedings. SecDev 2016 (Boston, MA, USA, Nov. 3–4, 2016). IEEE Computer Society, 45–52. doi:10.1109/SecDev.2016.019.

Federico Mora, Murphy Berzish, Mitja Kulczynski, Dirk Nowotka, and Vijay Ganesh. 2021. "Z3str4: A Multi-armed String Solver." In: *Formal Methods. 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Springer, 389–406. doi:10.1007/978-3-030-90870-6_21.

Donald R. Morrison. Oct. 1968. "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric." *Journal of the ACM*, 15, 4, (Oct. 1968), 514–534.

Luca Negrini, Vincenzo Arceri, Agostino Cortesi, and Pietro Ferrara. Aug. 2024. "Tarsis: An effective automata-based abstract domain for string analysis." *Journal of Software: Evolution and Process*, 36, 8, (Aug. 2024). doi:10.1002/smr.2647.

Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2021. "Twinning Automata and Regular Expressions for String Static Analysis." In: *Verification, Model Checking, and Abstract Interpretation. 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17–19, 2021, Proceedings*. Springer, 267–290. doi:10.1007/978-3-030-67067-2_13.

Greg Nelson. 1980. "Techniques for Program Verification." Ph.D. Dissertation. Stanford University. https://people.eecs.berkeley.edu/~necula/Papers/nelson-thesis.pdf.

Tobias Nipkow and Dmitriy Traytel. 2014. "Unified Decision Procedures for Regular Expression Equivalence." In: *Interactive Theorem Proving. 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014, Proceedings*. Springer, 450–466. doi:10.1007/978-3-319-08970-6_29.

Chris Okasaki and Andy Gill. 1998. "Fast Mergeable Integer Maps." In: *The 1998 ACM SIGPLAN Workshop on ML*. informal proceedings (Baltimore, MD, USA, Sept. 26, 1998), 77–86. https://citeseerx.ist.psu.edu/document?doi=bf0657d381196a93bc17220f1999dc40ee35e64a.

"Inferring Regular Languages in Polynomial Updated Time." *Pattern Recognition and Image Analysis. Selected Papers from the IVth Spanish Symposium*. Series in Machine Perception and Artifical Intelligence 1. World Scientific, 49–61. doi:10.1142/9789812797902_0004.

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. "Dynamic Typing with Dependent Types." In: *Exploring New Frontiers of Theoretical Informatics*. IFIP 18th World Computer Congress; TC1 3rd International Conference on Theoretical Computer Science (TCS2004) (Toulouse, France, Aug. 22–27, 2004). Ed. by Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell. Springer, 437–450. doi:10.1007/1-4020-8141-3_34.

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. "Data-Driven Precondition Inference with Learned Features." In: *PLDI'16. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA, June 13–17, 2016). ACM, 42–56. doi:10.1145/2908080.2908099.

Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. "Precise and Scalable Static Analysis of jQuery using a Regular Expression Domain." In: *DLS'16. Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands, Nov. 1, 2016). ACM, 25–36. doi:10.1145/2989225.2989228.

Terence J. Parr and Russell W. Quong. July 1995. "ANTLR: A Predicated-$LL(k)$ Parser Generator." *Software: Practice and Experience*, 25, 7, (July 1995), 789–810. doi:10.1002/spe.4380250705.

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. "Liquid Types." In: *PLDI'08. Proceedings of the 2008 SIGPLAN Conference on Programming Language Design & Implementation* (Tucson, AZ, USA, June 7–13, 2008). ACM, 159–169. doi:10.1145/1375581.1375602.

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. "A Symbolic Execution Framework for JavaScript." In: *2010 IEEE Symposium on Security and Privacy*. Proceedings (Berkeley/Oakland, CA, USA, May 16–19, 2010). IEEE Computer Society, 513–528. doi:10.1109/SP.2010.38.

Michael Schröder and Jürgen Cito. 2022. "Grammars for Free: Toward Grammar Inference for Ad Hoc Parsers." In: *2022 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. Proceedings. ICSE-NIER 2022 (Pittsburgh, PA, USA, May 22–27, 2022). IEEE Computer Society, 41–45. doi:10.1145/3510455.3512787.

Michael Schröder, Marc Goritschnig, and Jürgen Cito. Apr. 19, 2023. "An Exploratory Study of Ad Hoc Parsers in Python," (Apr. 19, 2023). arXiV preprint. Accepted as a registered report for MSR 2023 with Continuity Acceptance (CA). doi:10.48550/arXiv.2304.09733.

Mohamed Nassim Seghir and Daniel Kroening. 2013. "Counterexample-Guided Precondition Inference." In: *Programming Languages and Systems. 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 2013, Proceedings*. Springer, 451–471. doi:10.1007/978-3-642-37036-6_25.

Axel Simon, Andy King, and Jacob M. Howe. 2003. "Two Variables per Linear Inequality as an Abstract Domain." In: *Logic Based Program Synthesis and Transformation. 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 2002, Revised Selected Papers*. Ed. by Michael Leuschel. Springer, 71–89. doi:10.1007/3-540-45013-0_7.

Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. "Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints." In: *PLDI '21. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, June 20–25, 2021). ACM, 620–635. doi:10.1145/3453483.3454066.

The Unicode Standard, Version 16.0.0. The Unicode Consortium, (2024). https://www.unicode.org/versions/Unicode16.0.0/.

Ken Thompson. June 1968. "Regular Expression Search Algorithm." Programming Techniques. *Communications of the ACM*, 11, 6, (June 1968), 419–422.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2020. "Inter-theory Dependency Analysis for SMT String Solvers." *Proceedings of the ACM on Programming Languages*, 4, OOPSLA, Article 192. doi:10.1145/3428260.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. "Refinement Types for Haskell." In: *ICFP'14. Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden, Sept. 1–6, 2015). ACM, 269–282. doi:10.1145/2628136.2628161.

Alessandro Warth and Ian Piumarta. 2007. "OMeta: An Object-Oriented Language for Pattern Matching." In: *DLS'07: Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, QC, Canada, Oct. 22, 2007). ACM, 11–19. doi:10.1145/1297081.1297086.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/.

Received 2024-10-13; accepted 2025-08-12