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 for parsing, 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 approach this problem as an intersection of refinement typing and abstract interpretation. 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 refinement variables, and a set of abstract domains for efficiently representing the kinds of numeric and string values encountered during regular ad hoc parsing. We implement our approach in the Panini system and evaluate its efficacy on a benchmark of 202 Python ad hoc parsers.

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 use combinations of common string operations like `slice`, `index`, or `trim` 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 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

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.

2018. ACM 2475-1421/2018/1-ART1
https://doi.org/

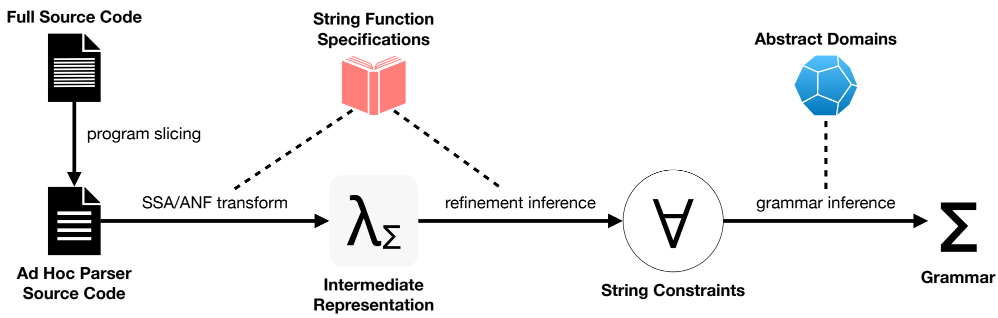

Fig. 1. The complete Panini system.

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. 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 [MacQueen et al. 2020, § 4]. 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.

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.

Table 1. Python operations as axiomatic λ_Σ specifications.

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

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- 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.

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 shows some examples of such axioms, based on the semantics of certain Python functions. We will use these axioms in the examples throughout this paper.

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 strings using a dependent function type and the string length operator $\lvert\Box\rvert$ of the refinement logic,

$$\text{length} \;:\; (s : \mathbb{S}) \to \{n : \mathbb{N} \mid n = \lvert s \rvert\}.$$

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

```
assert s[0] == "a"
```

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

Fig. 2. A simple Python expression (left) and the equivalent λ_Σ program (middle) with an incomplete verification condition (right) for its inferred type $\{s : \mathbb{S} \mid \kappa(s)\} \rightarrow \mathbb{1}$.

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 *satisfiability modulo theories* (SMT) [Barrett et al. 2021], which means VCs can be discharged using an off-the-shelf constraint solver such as Z3 [De Moura and Bjørner 2008]. Our system uses quantifier-free linear arithmetic with uninterpreted functions (QF_UFLIA) [Barrett et al. 2016] for its refinement predicates, extended with a theory of operations over strings [Berzish et al. 2017].

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 in Figure 2 (on the left). Assuming the function specifications from Table 1, we can transform this expression to an equivalent λ_Σ program (in the middle) with an inferred top-level refinement type of $\{s : \mathbb{S} \mid \kappa(s)\} \rightarrow \mathbb{1}$ and a VC (on the right) that closely matches the program. 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 $\kappa(s) \mapsto \text{true}$, i.e., allowing *any* string for s, is not a valid solution because it 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.

Example 2.3. To give an idea of how grammar inference works, we present a brief example.

The program shown in Figure 3a is a simple parser written in Python that checks if the first character of the input string is an ‘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 an ‘a’, then the program asserts that the second character must be a ‘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 3b 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, Figure 3c shows a (possible) step-by-step reduction of φ into a simple quantifier-free predicate using abstract interpretation (§ 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, equivalent to a regular expression.

```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")      ✓
```

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

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

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

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

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

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

(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. 3. 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 by the Sprite tutorial language by Jhala and Vazou [2020], and incorporate ideas from various other refinement type systems [Cosman and Jhala 2017; Dunfield and Krishnaswami 2021; Montenegro et al. 2020]. 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 in A-normal form (ANF) [Bowman 2022; Flanagan et al. 1993]. It exists solely for type synthesis and its programs are neither meant to be executed nor written by hand. Its syntax, collected in Figure 4, is thus minimal and has few affordances.

Values are either primitive constants or variables. **Terms** are comprised of values and the usual constructs: function applications, function abstractions, bindings, recursive bindings, and branches. Applications are in ANF as 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 (§ 3.2). λ_Σ terms have no type annotations, except on $\boldsymbol{\lambda}$-binders and recursive **rec**-binders, whose (base) types we assume are inferred in the 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 refinement predicates, or by constructing (dependent) function types, whose output types can refer to input types.

Predicates are terms in a Boolean logic, with the usual Boolean connectives, plus (in)equality relations and arithmetic comparisons between predicate expressions, membership queries for regular expression matching, applications of unknown κ variables, and existential quantifiers, which arise during the Fusion phase of κ solving (§ 3.3). Both κ applications and existentials are not part of the user-visible surface syntax. **Expressions** within predicates are built from lifted values and functions, in particular linear integer arithmetic and operations over strings, e.g., length $|\square|$, character-at-index $\square[\square]$, substring $\square[\square..\square]$, etc. 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 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.

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 λ-abstractions and recursive bindings. The types of applied functions need to be known, however, and available in the typing context (see the discussion of axioms, § 2.1 and Table 1). Our system borrows heavily from Liquid Haskell [Vazou et al. 2014] and the expositions given by Cosman and Jhala [2017] and Jhala and Vazou [2020]. We combine typing rules and VC generation into one syntax-driven declarative system of inference rules, given in Figure 5 and discussed below.

$\boxed{t_1 \leqslant t_2 \dashv c}$ ***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 the

Values	v	$::=$	$x, y, z, \ldots$	variables
		$\mid$	$\ldots$ varies $\ldots$	constants
Terms	e	$::=$	v	value
		$\mid$	$e\ v$	application
		$\mid$	$\boldsymbol{\lambda}(x : b).\ e$	abstraction
		$\mid$	$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$	binding
		$\mid$	$\textbf{rec}\ x : t = e_1\ \ e_2$	recursion
		$\mid$	$\textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2$	branch
Base Types	b	$::=$	$\mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{Ch} \mid \mathbb{S}$	
Types	t	$::=$	$\{x : b \mid p\}$	refined base
		$\mid$	$(x : t_1) \longrightarrow t_2$	dependent function
Predicates	p	$::=$	$\mathsf{true} \mid \mathsf{false}$	Boolean constants
		$\mid$	$p_1 \wedge p_2 \mid p_1 \vee p_2$	connectives
		$\mid$	$p_1 \Rightarrow p_2 \mid p_1 \Leftrightarrow p_2$	implications
		$\mid$	$\neg p$	negation
		$\mid$	$w_1 = w_2 \mid w_1 \neq w_2$	(in)equality
		$\mid$	$w_1 < w_2 \mid w_1 \leq w_2$	arithmetic comparison
		$\mid$	$w \in \mathrm{RE}$	regular language membership
		$\mid$	$\kappa(\overline{v})$	κ application
		$\mid$	$\exists(x : b).\ p$	existential quantification
Expressions	w	$::=$	v	value
		$\mid$	$f(\overline{w})$	function
Constraints	c	$::=$	p	predicate
		$\mid$	$c_1 \wedge c_2$	conjunction
		$\mid$	$\forall(x : b).\ p \Rightarrow c$	universal implication

Fig. 4. Syntax of λ_Σ terms, types, and refinements.

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. This is done using a *generalized implication* operation $(x :: t) \Rightarrow c$, which simply ensures that only base types are bound to quantifiers in the refinement logic.

$\boxed{\Gamma \vdash t \rhd \hat{t}}$ ***Template Generation.*** To enable complete type synthesis for all intermediate terms, it is sometimes necessary to turn a type t into a template $\hat{t}$, where the 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, representing an n-ary relation between the type itself and all variables in the current environment Γ. Usually this environment is empty, but if t is a function, Kap/Fun recursively generates input and output templates, extending the environment along the way.

$$\boxed{t_1 \leqslant t_2 \dashv c} \quad \textbf{Subtyping}$$

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

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

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

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

$$t \rhd \hat{t} \overset{\text{def}}{=} \varnothing \vdash t \rhd \hat{t}$$

$$\frac{\kappa \text{ is a fresh variable of sort } b \times \overline{t}}{\overline{x : t} \vdash \{v : b \mid p\} \rhd \{v : b \mid \kappa(v, \overline{x})\}}\ \textsc{Kap/Base} \qquad \frac{\Gamma \vdash t_1 \rhd \hat{t}_1 \qquad \Gamma[x \mapsto t_1] \vdash t_2 \rhd \hat{t}_2}{\Gamma \vdash (x : t_1) \to t_2 \rhd (x : \hat{t}_1) \to \hat{t}_2}\ \textsc{Kap/Fun}$$

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

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

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

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

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

$$\frac{\begin{array}{cccc} \tilde{t}_1 \rhd \hat{t}_1 & \Gamma[x \mapsto \hat{t}_1] \vdash e_1 \nearrow t_1 \dashv c_1 & & t_1 \leqslant \hat{t}_1 \dashv \hat{c}_1 \\ & \Gamma[x \mapsto t_1] \vdash e_2 \nearrow t_2 \dashv c_2 & t_2 \rhd \hat{t}_2 & t_2 \leqslant \hat{t}_2 \dashv \hat{c}_2 \end{array}}{\Gamma \vdash \textbf{rec}\ x : \tilde{t}_1 = e_1\ \ e_2 \nearrow \hat{t}_2 \dashv ((x :: \hat{t}_1) \Rightarrow c_1 \land \hat{c}_1) \land ((x :: t_1) \Rightarrow c_2 \land \hat{c}_2)}\ \textsc{Syn/Rec}$$

$$\frac{\begin{array}{cccc} \Gamma \vdash x \nearrow \mathbb{B} & \Gamma \vdash e_1 \nearrow t_1 \dashv c_1 & t_1 \rhd \hat{t} & t_1 \leqslant \hat{t} \dashv \hat{c}_1 \\ & \Gamma \vdash e_2 \nearrow t_2 \dashv c_2 & & t_2 \leqslant \hat{t} \dashv \hat{c}_2 \end{array}}{\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \nearrow \hat{t} \dashv (x = \mathsf{true} \Rightarrow c_1 \land \hat{c}_1) \land (x = \mathsf{false} \Rightarrow c_2 \land \hat{c}_2)}\ \textsc{Syn/If}$$

Fig. 5. Typing rules for λ_Σ.

$\boxed{\Gamma \vdash e \nearrow t \dashv c}$ ***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/Con synthesizes built-in primitive types, denoted by $\mathsf{prim}(c)$ in the obvious way, e.g., $\mathsf{prim}(0) \stackrel{\text{def}}{=} \{v : \mathbb{Z} \mid v = 0\}$ and so on. Syn/Var retrieves the type of a variable from the current context, using *selfification* to produce the most precise possible type [Ou et al. 2004]. The self function lifts the variable into the refinement, allowing each occurrence of the variable in different branches of the program to be precisely typed.

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

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 a fresh κ variable, based on 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. Syn/If synthesizes the type of the whole conditional as a 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 (i.e., a constraint containing κ variables) and an initial κ assignment σ (usually empty). 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 detailed in § 4 to find an abstract solution for κ, i.e., a grammar. If the complete VC, with all κ variables replaced by their solutions, is satisfiable (modulo theories) then the inferred type is valid and σ contains concrete assignments for all of the type's refinement variables. Otherwise, no valid solution could be found, either because there does not exist one (e.g., the program has a type error) or due to insufficient invariants or limits of the abstract domain.

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 $\kappa(s) \mapsto \mathsf{false}$ 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

Algorithm 1 Solve an incomplete VC and find all κ assignments

procedure SOLVE(c, σ)
  $c \leftarrow \sigma(c)$
  $c \leftarrow$ FUSION(c) ▷ eliminate local acyclic κ variables [Cosman and Jhala 2017]
  $\sigma \leftarrow \sigma \cup$ LIQUID(c) ▷ solve residual non-grammar κ variables [Rondon et al. 2008]
  $c \leftarrow \sigma(c)$
  for all constraints in c of the form $\forall(s : \mathbb{S}).\ \kappa(s) \Rightarrow \varphi$ **do**
    $\hat{\sigma} \leftarrow [\![\varphi]\!]^\sharp$ ▷ infer grammar using abstract interpretation [§ 4]
    $\sigma \leftarrow \sigma\,[\kappa(s) \mapsto s \in \hat{\sigma}(s)]$
    $c \leftarrow \sigma(c)$
  end for
  if SATISFIABLE(c) **then return** VALID **else return** INVALID
end procedure

(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, see § 3.3), as well as the correctness of all involved axiomatic string function specifications (Table 1), then, under some model $\mathfrak{M}$ of the refinement logic (e.g., linear integer arithmetic and 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 than φ but semantically equivalent. They key components of our approach are:

(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 (§ 4.3),
(4) a procedure for eliminating quantified variables by abstract substitution (§ 4.4),
(5) abstract value representations of all base types (§ 5).

4.1 Abstract Interpretation

Background. Abstract interpretation [Cousot and Cousot 1977, 1979] is a well-established framework for formalizing static program analyses. It involves the sound approximation of all possible

Table 2. Summary of abstract domains in PANINI

base type				abstract domain	
unit	$\wp(\mathbb{1})$	$=$	$\hat{\mathbf{1}}$	the two-element lattice	§ 5.1
Booleans	$\wp(\mathbb{B})$	$=$	$\hat{\mathbf{B}}$	Boolean subset lattice	§ 5.2
integers	$\wp(\mathbb{Z})$	$\supseteq$	$\hat{\mathbf{Z}}$	open-ended interval lists	§ 5.3
characters	$\wp(\mathbb{Ch})$	$=$	$\hat{\mathbf{C}}$	Unicode character sets	§ 5.4
strings	$\wp(\mathbb{S})$	$\supseteq$	$\hat{\mathbf{S}}$	regular expressions over $\hat{\mathbf{C}}$	§ 5.5

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, for all programs $\mathcal{P}$, $\alpha([\![\mathcal{P}]\!]) \sqsubseteq [\![\mathcal{P}]\!]^\sharp$, 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 that the abstraction loses no information and the abstract semantics exactly captures the concrete semantics of the program [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, and in particular denote the parser's input string 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. We give a brief summary of the domains currently used by PANINI in Table 2 and provide complete definitions in § 5. Note that our abstract string domain $\hat{\mathbf{S}}$ represents sets of strings with regular expressions. This means that our approach must under-approximate any $L(P)$ above regular in the Chomsky hierarchy [Chomsky and Schützenberger 1963]. 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 refinement variable,

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

$$\llbracket\varphi\rrbracket^{\sharp} \doteq \left\{x \mapsto \llbracket\varphi\rrbracket{\upharpoonleft}_x \,\middle|\, x \in \mathrm{vars}(\varphi)\right\}$$

$$\begin{aligned}
\llbracket P_1 \wedge P_2\rrbracket{\upharpoonleft}_x &\doteq \llbracket P_1\rrbracket{\upharpoonleft}_x \sqcap \llbracket P_2\rrbracket{\upharpoonleft}_x && \text{if } P_1, P_2 \text{ quantifier-free}\\
\llbracket P_1 \vee P_2\rrbracket{\upharpoonleft}_x &\doteq \llbracket P_1\rrbracket{\upharpoonleft}_x \sqcup \llbracket P_2\rrbracket{\upharpoonleft}_x && \text{if } P_1, P_2 \text{ quantifier-free}\\
\llbracket \neg P\rrbracket{\upharpoonleft}_x &\doteq \neg\, \llbracket P\rrbracket{\upharpoonleft}_x && \text{if } P \text{ quantifier-free}\\
\llbracket \varphi\rrbracket{\upharpoonleft}_x &\doteq \llbracket \mathsf{qelim}(\varphi)\rrbracket{\upharpoonleft}_x\\
\llbracket \rho\rrbracket{\upharpoonleft}_x &\doteq \begin{cases}\omega, & \text{as defined by domain;}\\ \langle x\colon \rho\rangle\,, & \text{otherwise.}\end{cases}
\end{aligned}$$

Fig. 6. Abstract semantics of λ_Σ constraints.

$$\begin{array}{llcll}
\textbf{Constraints} & \varphi & ::= & \mathsf{true} \mid \mathsf{false} & \text{Boolean constants}\\
 & & \mid & \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 & \text{connectives}\\
 & & \mid & \varphi_1 \Rightarrow \varphi_2 \mid \varphi_1 \Leftrightarrow \varphi_2 & \text{implications}\\
 & & \mid & \neg\varphi & \text{negation}\\
 & & \mid & \exists x.\ \varphi & \text{existential quantification}\\
 & & \mid & \forall x.\ \varphi & \text{universal quantification}\\
 & & \mid & \rho & \text{relations}\\
 & & & & \\
\textbf{Relations} & \rho & ::= & \omega_1 \bowtie \omega_2 \quad \text{where } \bowtie\, \in \{=, \neq, <, \leq, \in, \notin, \between, \parallel\} & \\
 & & & & \\
\textbf{Expressions} & \omega & ::= & x, y, z, \ldots & \text{variables}\\
 & & \mid & c & \text{concrete value}\\
 & & \mid & \hat{c} & \text{abstract value}\\
 & & \mid & \langle x\colon \rho\rangle & \text{abstract relation}\\
 & & \mid & f(\overline{\omega}) & \text{function}
\end{array}$$

Fig. 7. Extended syntax of λ_Σ constraints.

The definition of the abstract semantics function $\llbracket\square\rrbracket^{\sharp}$ is given in Figure 6. It depends on a novel variable-focused relational semantics $\llbracket\square\rrbracket{\upharpoonleft}_{\square}$ and a quantifier elimination procedure $\mathsf{qelim}(\square)$. We describe these in the remainder of this section, after establishing some syntactic extensions to λ_Σ constraints.

4.2 Extended Constraint Syntax

We extend the formal syntax of predicates and constraints from Figure 4 to include abstract values and relations. The extended syntax is given in Figure 7.

Constraints φ, in addition to Boolean constants and the usual logical connectives, include both universal and existential quantification, and generalized relations between predicate 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 variables and concrete values, now include abstract values $\hat{c}$ and abstract relations $\langle x\colon \rho\rangle$. Abstract values $\hat{c}$ are taken from our abstract domains (§ 5) 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 $\Sigma^*\mathtt{a}$ represents the set of all strings that end with the character 'a'. For functions, we assume the usual notational conveniences, e.g., we write $a + b$ for $+(a, b)$ and $|s|$ for str.len(s) and so on. If an abstract value appears in a function, it lifts the whole expression into the abstract realm, e.g., $x + [1, \infty]$ represents the set of expressions $\{x + 1, x + 2, x + 3, \dots\}$. Abstract relations $\langle x\colon \rho\rangle$ similarly represent sets of values, specifically those values that are described by the given relation. For example, the abstract relation $\langle x\colon x > 5\rangle$ represents "the set of all values x for which $x > 5$ is true" and $\langle i\colon s[i] = c\rangle$ represents the set of all indexes i at which the string in variable s contains the character in variable c.

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

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

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

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 (§ 4.3).

$$\begin{array}{rclcl} 1 + x - 2 & \rightsquigarrow & x - 1 & & \\ |\text{“abc”}| & \rightsquigarrow & 3 & & \\ x - 1 \in [1, 5] & \rightsquigarrow & x \in [1, 5] + 1 & \rightsquigarrow & x \in [2, 6] \\ x < 5 & \rightsquigarrow & x \in [-\infty, 4] & & \\ |s| + 1 > 2 & \rightsquigarrow & |s| > 1 & \rightsquigarrow & s \in \Sigma^+ \\ [1, 2] \between \langle i\colon s[i] = \text{‘a’}\rangle & \rightsquigarrow & s\big[[1, 2]\big] \ni \text{‘a’} & \rightsquigarrow & s \in \Sigma\Sigma?\mathtt{a}\Sigma^* \end{array}$$

In the remainder, we assume that expressions and relations are always fully normalized.

4.3 Relational Semantics

The concrete semantics $\llbracket \rho \rrbracket$ of a single relation are all those assignments of the relation's free variables that make the relation true, e.g.,

$$\begin{aligned} \llbracket x > 3 \rrbracket &\doteq \{x \mapsto 4, x \mapsto 5, \dots\}, \\ \Big\llbracket |s| > 3 \Big\rrbracket &\doteq \{\dots, s \mapsto \text{“abaa”}, s \mapsto \text{“abab”}, \dots\} \\ \Big\llbracket |s| > x \Big\rrbracket &\doteq \left\{\dots, \begin{pmatrix} s \mapsto \text{“ab”} \\ x \mapsto 0 \end{pmatrix}, \begin{pmatrix} s \mapsto \text{“ab”} \\ x \mapsto 1 \end{pmatrix}, \dots\right\}. \end{aligned}$$

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 [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 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, i.e.,

$$\mathfrak{M}, [x \mapsto c] \models \rho \qquad \Longleftrightarrow \qquad c \in [\![\rho]\!]\Uparrow_x .$$

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

$$[\![\rho]\!]\Uparrow_x \doteq \langle x\colon \rho\rangle ,$$

since by definition they exactly capture the abstract semantics of the relation ρ for the given variable x. 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 (§ 5). For example, for the domains of regular expressions and integers, we can define precise abstractions for simple string length relations,

$$\Big[\!\!\Big[|s| = n\Big]\!\!\Big]\Uparrow_s \doteq \Sigma^n, \qquad \Big[\!\!\Big[|s| \geq n\Big]\!\!\Big]\Uparrow_s \doteq \Sigma^n\Sigma^*, \qquad \Big[\!\!\Big[|s| \leq n\Big]\!\!\Big]\Uparrow_s \doteq \Sigma^0 + \Sigma^1 + \cdots + \Sigma^n,$$

where n is a meta-variable standing for 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:

$$[\![x > 3]\!]\Uparrow_x \doteq [4, \infty] \qquad \Big[\!\!\Big[|s| > 3\Big]\!\!\Big]\Uparrow_s \doteq \Sigma^4\Sigma^* \qquad \Big[\!\!\Big[|s| > x\Big]\!\!\Big]\Uparrow_s \doteq \langle s\colon\ |s| > x\rangle$$

$$\Big[\!\!\Big[|s| > x\Big]\!\!\Big]\Uparrow_x \doteq |s| - [1, \infty]$$

4.4 Quantifier Elimination

Figure 8 presents our procedure for eliminating quantifiers by abstract substitution. The top-level function qelim traverses a given constraint to eliminate all of its quantifiers. During this traversal, we apply standard logical transformations to simplify the problem.

$$\forall x.\varphi \iff \neg\exists x.\neg\varphi \qquad \text{(DeMorgan's law)}$$

$$\exists x.\ \bigvee\bigwedge\rho \iff \bigvee\exists x.\ \bigwedge\rho \qquad \text{(distributivity of } \exists \text{ over } \bigvee)$$

The actual variable elimination happens in qelim1, where we only need to consider a single conjunctive set of relations R. To eliminate a particular variable x from the relations in this set, we first compute the x-relative relational semantics $[\![\rho]\!]\Uparrow_x$ for all relations ρ in the subset of relations in R that contain x as a free variable. This results in a set E of (abstract) expressions, all representing some aspect of x in R. The expressions in E do not contain the variable x but they might contain other free variables. We now generate all unordered pairwise combinations of expressions in E and create a new equality relation between each pair of expressions (using a relational operator appropriate for the pair's types), resulting in 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 in the first place.

5 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.

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

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

Fig. 8. Quantifier elimination

To simplify the definitions and avoid boilerplate repetition, 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 $\top_{\mathbb{1}}, \top_{\mathbb{B}}$, etc.

5.1 Unit

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

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

5.2 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

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

5.3 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

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

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

$$\begin{aligned}
[a,b] \sqsubseteq [c,d] &\iff c \le a \wedge b \le d \\
\top &= [-\infty, \infty] \\
[a,b] \sqcup [c,d] &= [\min(a,c), \max(b,d)] \\
[a,b] \sqcap [c,d] &= [\max(a,c), \min(b,d)]
\end{aligned}$$

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

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

While **IZ** can abstractly represent infinite contiguous sets, such as those defined by a single inequality relation like $\{x \in \mathbb{Z} \mid x > 5\}$, it can not 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:

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

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

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

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.

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

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

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

$$\begin{aligned}
\neg[-\infty, b_1 \mid a_2, b_2 \mid \dots \mid a_n, \infty] &= [b_1 + 1, a_2 - 1 \mid b_2 + 1, a_3 - 1 \mid \dots \mid b_{n-1} + 1, a_n - 1] \\
\neg[-\infty, b_1 \mid a_2, b_2 \mid \dots \mid a_n, b_n] &= [b_1 + 1, a_2 - 1 \mid \dots \mid b_{n-1} + 1, a_n - 1 \mid b_n + 1, \infty] \\
\neg[a_1, b_1 \mid a_2, b_2 \mid \dots \mid a_n, \infty] &= [-\infty, a_1 - 1 \mid b_1 + 1, a_2 - 1 \mid \dots \mid b_{n-1} + 1, a_n - 1] \\
\neg[a_1, b_1 \mid a_2, b_2 \mid \dots \mid a_n, b_n] &= [-\infty, a_1 - 1 \mid b_1 + 1, a_2 - 1 \mid \dots \mid b_{n-1} + 1, a_n - 1 \mid b_n + 1, \infty]
\end{aligned}$$

[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 $\mathbf{IZ}$. 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 $\alpha : \wp(\mathbb{Z}) \rightarrow \hat{\mathbf{Z}}$ 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 $\gamma : \hat{\mathbf{Z}} \rightarrow \wp(\mathbb{Z})$, 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.

$$\begin{array}{lll} [\![x = a]\!]\!\uparrow_x \doteq [a,a] & [\![x \ge a]\!]\!\uparrow_x \doteq [a,\infty] & [\![x \le a]\!]\!\uparrow_x \doteq [-\infty,a] \\ [\![x \ne a]\!]\!\uparrow_x \doteq [-\infty,a-1 \mid a+1,\infty] & [\![x > a]\!]\!\uparrow_x \doteq [a+1,\infty] & [\![x < a]\!]\!\uparrow_x \doteq [-\infty,a-1] \end{array}$$

5.4 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 149 878 characters [Unicode Consortium 2023]. 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., $\Sigma \backslash \{\mathtt{a}, \mathtt{b}\}$ means the set of all characters excluding ‘a’ and ‘b’. For singleton sets like $\{\mathtt{a}\}$ we will usually drop the braces and just write $\mathtt{a}$. 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

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

5.5 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]. 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, shown in Figure 9, that allows us to abstract all strings expressed via relations between common string operations.

$$\llbracket |s| \in \hat{n} \rrbracket\upharpoonright_s \doteq \Sigma^{\hat{n}} \qquad \hat{\square}^{\hat{n}} = \textstyle\bigsqcup_{[a,b]\in\hat{n}} \hat{\square}^{[a,b]}$$

$$\llbracket |s| \notin \hat{n} \rrbracket\upharpoonright_s \doteq \Sigma^{\neg\hat{n}} \qquad \hat{\square}^{[a,b]} = \begin{cases} \hat{\square}^{a}\hat{\square}^{*} & \text{if } b = \infty \\ \hat{\square}^{a} + \hat{\square}^{a+1} + \cdots + \hat{\square}^{b} & \text{otherwise} \end{cases}$$

$$\llbracket s[i] \in \hat{c} \rrbracket\upharpoonright_s \doteq \Sigma^{i}\hat{c}\Sigma^{*} \qquad \llbracket s[i] \notin \hat{c} \rrbracket\upharpoonright_s \doteq \Sigma^{[0,i]} + \Sigma^{i}(\neg\hat{c})\Sigma^{*}$$

$$\llbracket s[i] \in \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{[0,i]} \qquad \llbracket s[i] \notin \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{i+1}\Sigma*$$

$$\llbracket s[\hat{\imath}] \between \hat{c} \rrbracket\upharpoonright_s \doteq \Sigma^{\hat{\imath}}\hat{c}\Sigma^{*} \qquad \llbracket s[\hat{\imath}] \parallel \hat{c} \rrbracket\upharpoonright_s \doteq \bigsqcap_{[a,b]\in\hat{\imath}} \begin{cases} \Sigma^{[0,a]} + \Sigma^{a}(\neg\hat{c})^{+} & \text{if } b = \infty \\ \Sigma^{[0,a]} + \Sigma^{a}(\neg\hat{c})^{b-a+1}\Sigma^{*} & \text{otherwise} \end{cases}$$

$$\llbracket s[\hat{\imath}] \between \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{\min(\hat{\imath})} \qquad \llbracket s[\hat{\imath}] \parallel \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{\max(\hat{\imath})}\Sigma^{*} \quad \text{if } \max(\hat{\imath}) \neq \infty$$

$$\llbracket s\big[|s| - i\big] \in \varnothing \rrbracket\upharpoonright_s \doteq \varnothing \qquad \llbracket s\big[|s| - i\big] \notin \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{*}$$

$$\llbracket s\big[|s| - i\big] \in \hat{c} \rrbracket\upharpoonright_s \doteq \Sigma^{*}\hat{c}\Sigma^{i-1} \qquad \llbracket s\big[|s| - i\big] \notin \hat{c} \rrbracket\upharpoonright_s \doteq \Sigma^{*}(\neg\hat{c})\Sigma^{i-1} + \Sigma^{[0,i-1]}$$

$$\llbracket s\big[|s| - \hat{\imath}\big] \between \varnothing \rrbracket\upharpoonright_s \doteq \varnothing \qquad \llbracket s\big[|s| - \hat{\imath}\big] \parallel \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{*}$$

$$\llbracket s\big[|s| - \hat{\imath}\big] \between \hat{c} \rrbracket\upharpoonright_s \doteq \Sigma^{*}\hat{c}\Sigma^{\hat{\imath}-1}$$

$$\llbracket s\big[|s| - \hat{\imath}\big] \parallel \hat{c} \rrbracket\upharpoonright_s \doteq \bigsqcap_{[a,b]\in\hat{\imath}} \begin{cases} (\neg\hat{c})^{*}\Sigma^{a-1} + \Sigma^{[0,a-1]} & \text{if } b = \infty \\ \llbracket s\big[|s| - [a, b-1]\big] \parallel \hat{c} \rrbracket\upharpoonright_s \Sigma + \varepsilon & \text{otherwise} \end{cases}$$

$$\llbracket s[i..j] \in \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{[0,i]} \qquad \llbracket s[i..j] \notin \varnothing \rrbracket\upharpoonright_s \doteq \Sigma^{j+1}\Sigma^{*}$$

$$\llbracket s[i..j] \in \hat{t} \rrbracket\upharpoonright_s \doteq \Sigma^{i}(\Sigma^{j-i+1} \sqcap \hat{t})\Sigma^{*} \qquad \llbracket s[i..j] \notin \hat{t} \rrbracket\upharpoonright_s \doteq \Sigma^{[0,i]} + \Sigma^{i}(\Sigma^{j-i+1} \setminus \hat{t})\Sigma^{*}$$

$$\llbracket s[\hat{\imath}..\hat{\jmath}] \between \hat{t} \rrbracket\upharpoonright_s \doteq \Sigma^{\hat{\imath}}(\Sigma^{\hat{\jmath}-\hat{\imath}+1} \sqcap \hat{t})\Sigma^{*} \qquad \llbracket s[i..j] \parallel \hat{t} \rrbracket\upharpoonright_s \doteq \neg\llbracket s[i..j] \between \hat{t} \rrbracket\upharpoonright_s$$

Fig. 9. Abstract relational semantics for strings. Abstract values are denoted $\hat{\square}$. We assume single characters have been lifted to singleton character sets, e.g., $s[i] = c \rightsquigarrow s[i] \in \hat{c}$ where $\hat{c} = \{c\}$.

6 Implementation and Evaluation

We have implemented our approach in the PANINI prototype system, available at https://anonymous.4open.science/r/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.

6.1 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 $\hat{\mathrm{S}}$ and abstract characters $\hat{\mathrm{C}}$. The machine representation of $\hat{\mathrm{C}}$ is based on complemented PATRICIA tries [Kmett 2012; Morrison 1968; Okasaki and Gill 1998], which enable compact representation of negated sets while allowing all common set operations. For $\hat{\mathrm{S}}$, we implemented an efficient regular expression type whose literals are abstract characters from $\hat{\mathrm{C}}$ and whose operations, like regular intersection and complement, are performed purely algebraically, without going through finite automata. Our approach uses Brzozowski derivatives [Antimirov 1996; Brzozowski 1964] and is based on the equation-solving technique by Acay [2018], using Arden's lemma [Arden 1961] and Gaussian elimination to solve a system of regular equations. It is similar to the approach by Liang et al. [2015], but makes use of the local mintermization trick employed by Keil and Thiemann [2014] to effectively compute precise derivative over large alphabets. To keep regular expressions as concise and human-readable as possible, we aggressively simplify intermediate expressions using the transformation rules and heuristics described by Kahrs and Runciman [2022].

6.2 Experiments

To provide insights into the efficacy and applicability of the Panini prototype for inferring grammars from ad hoc parser implementations, we created a benchmark dataset comprising 202 regular ad hoc parsers written in Python. 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 high-level Python string functions such as `index`, using loops to iterate over the characters in a string, parsing the input from back-to-front, etc. We also added domain-specific variations to illustrate real-world applicability (e.g., Figures 10 and 11).

To facilitate a structured analysis, we classified the parsers within our benchmark 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. Figure 10 shows an example of such a program, used to parse the angle-bracketed *addr-spec* part of an email address with a display name. *Programs with Conditionals* are ad hoc parsers that incorporate conditional statements to make decisions based on input characteristics. We saw such a program in the example of Figure 3. *Programs with Loops* are ad hoc parsers containing iterative constructs alongside conditional statements for string manipulation tasks. Figure 11 presents a parser looping over a bit-string to ensure that only the least-significant bit is set.

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

$$
\begin{array}{l}
\mathrm{getAddrSpec} : \{\mathit{email} : \mathbb{S} \mid \star\} \to \mathbb{S} \\
\mathrm{getAddrSpec} = \boldsymbol{\lambda}(\mathit{email} : \mathbb{S}). \\
\quad \textbf{let } v_0 = \mathrm{indexFrom}\ \mathit{email}\ \text{“<”}\ 0 \textbf{ in} \\
\quad \textbf{let } v_1 = \mathrm{ge}\ v_0\ 0 \textbf{ in} \\
\quad \textbf{let } \_ = \mathrm{assert}\ v_1 \textbf{ in} \\
\quad \textbf{let } b_1 = \mathrm{add}\ v_0\ 1 \textbf{ in} \\
\quad \textbf{let } b_2 = \mathrm{indexFrom}\ \mathit{email}\ \text{“>”}\ b_1 \textbf{ in} \\
\quad \textbf{let } v_3 = \mathrm{ge}\ b_2\ 0 \textbf{ in} \\
\quad \textbf{let } \_ = \mathrm{assert}\ v_3 \textbf{ in} \\
\quad \mathrm{slice}\ \mathit{email}\ b_1\ b_2
\end{array}
$$

Fig. 10. A straight-line ad hoc parser.

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

Table 3. Results of evaluating Panini on a varied dataset of ad hoc parsers.

Type	Parser Example		#	Inferred Language =	⊂	∅	Error	P	R	Time (s)
Straight	getAddrSpec	`[^<>]*<[^>]*>.*`	89	81	0	0	8	1.00	0.91	0.11 ±0.27
+ Cond.	Figure 3	`a|[^a]b.*`	50	45	0	1	4	1.00	0.90	0.05 ±0.02
+ Loops	lsb_check	`0*1`	63	31	5	8	19	1.00	0.49	5.25 ±11.86
			202	157	5	9	31	1.00	0.78	1.70 ±7.01

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 $L(G_i) = \emptyset$; and cases where Panini reports an error due to insufficient semantics or other limitations of the abstract domains.

We additionally computed the *precision* and *recall* of each inferred grammar. If the grammar was an exact match, both precision and recall are 100%. It the grammar was empty or could not be produced, we recorded 0% recall. In cases where Panini was able to infer an under-approximation, we used the ground truth grammar to generate 1000 random sample strings and tested them against the inferred grammar to calculate recall.

We also measured the computational overhead incurred during the grammar inference process, particularly wall-clock execution time. All benchmarks were run on an iMac with a 3 GHz Intel Core i5 processor and 24 GB RAM, using Z3 4.8.14 for SMT solving.

Results. Table 3 presents our experimental results, showcasing an illustrative example representative of each ad hoc parser category, together with summary statistics on the accuracy and performance of Panini's grammar inference. As expected, the average precision of an inferred grammar is 100 % across all types of parsers: Panini never over-approximates. Out of the 202 parser programs in our dataset, we achieve exact inference for 81 out of 89 straight-line programs (91 % recall); 45 out of 50 purely conditional programs (90 % recall); and 31 out of 63 programs containing loops, with an additional 5 loop programs being safely under-approximated (49 % recall). Panini fails to find a meaningful refinement beyond the empty language for 1 conditional and 8 loop programs, and gets stuck without results on 8 straight-line, 4 conditional, and 19 loop programs. We discuss the underlying reasons for under-approximations and failed inferences in § 6.3.

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. However, our implementation is not yet optimized in this area.

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

```
lsb_check : {s : 𝕊 | ★} → 𝟙
lsb_check = λ(s : 𝕊).
  rec L₂ : {i : ℤ | ★} → 𝟙 = λi.
    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
      let _ = assert v₆ in
      unit
  in L₂ 0
```

Fig. 11. An ad hoc parser with loops.

6.3 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. We see the following main areas of improvement as part of future work:

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. In many cases, predicate abstraction is quite sufficient, as in Figure 11, where the invariant $0 \leq i \leq |s|$ of the recursive function L_2 can be easily inferred. But even simple-seeming code variations can stump the system, as in Figure 12, where a final assertion statement outside of the loop body effectively constrains the loop to two iterations, inducing the invariant $0 \leq i \leq 2$. Unfortunately, our prototype cannot infer this invariant automatically, resulting in $\varnothing$ (if the invariant is provided as an annotation, the correct grammar is produced). To remedy this situation, 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 (§ 5). For example, our integer domain $\hat{\mathbb{Z}}$ cannot efficiently represent congruence classes, e.g., infinite sets of even or odd numbers. This can lead to under-approximations, as in Figure 13, 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 f102(s):
  i = 0
  while i < len(s):
    assert s[i] == "a"
    i += 1
  assert i == 2
```

Fig. 12. A parser for aa

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

Fig. 13. A parser for (ab)*

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

Fig. 14. A parser for aa.*

Relational Semantics. If Panini lacks the semantics to rewrite, normalize, or abstract a particular relation (see § 4.3), 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 Figure 9. But take the parser in Figure 14, 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.

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.

Context-Free Grammars. If Panini encounters a context-free parser, it will likely yield $\varnothing$ 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.

Table 4. Comparison of state-of-the-art grammar inference approaches.

Approach	Requirements			Output
	Samples	Parser	Interaction	
Panini	-	source	-	Regex
STALAGMITE	-	source	traced symbolic execution	CFG
Mimid	positive	source	traced execution	CFG
TreeVada	positive	oracle	membership query	CFG
TTT	(positive)	oracle	membership + equivalence query	DFA
Exbar	pos. + neg.	-	-	DFA

6.4 Comparison with Other Approaches

Table 4 compares the operating requirements of Panini and other grammar inference systems: whether or not they need pre-existing input samples, and of what kind; whether the parser needs to be available in source form to be inspected or modified, or as a black-box oracle, or not at all; and what form of interaction with the parser is necessary, if any. 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. 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 a set of known positive input samples.

Table 5. TTT on the Panini dataset

	P	R	Time (s)
Straight	0.27	0.76	12.89 ±13.15
+ Cond.	0.09	0.76	4.17 ±07.92
+ Loops	0.32	0.88	19.28 ±53.66
	0.24	0.80	12.72 ±31.80

Table 5 shows the results of running TTT on the same dataset of ad hoc Python parsers we used to evaluate Panini (cf. Table 3). To simulate equivalence queries, we randomly generated 20 positive samples for each parser from its ground truth grammar (an unrealistic prospect in a real-world setting, where ground truth does not exist a priori). The aggregated precision and recall measures (calculated as described in § 6.2), indicate that TTT tends to significantly over-approximate the true input language for these kinds of ad hoc parsers.

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].

Mimid [Gopinath et al. 2020] 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 2017], 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.

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 et al. 2008a] 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.

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 et al. 2023]. We argue that all other existing approaches have requirements that make them impractical to use in this setting.

6.5 Case Study: cgidecode.py

We demonstrate Panini's suitability for real-world ad hoc grammar inference using the cgidecode.py subject of the *Mimid* benchmark suite. This is a Python program to decode CGI-encoded strings. 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.

We compare Panini against *Mimid*, TreeVada, and TTT and reuse the *Mimid* benchmark setup: to evaluate *precision*, the inferred grammar is used to generate 1000 inputs to fuzz the original program, noting how many of those inputs were accepted; to evaluate *recall*, a human-written *golden grammar* is used to generate 1000 inputs to test against the inferred grammar. The results of our comparison (averaged over ten runs) are presented in Table 6.

Table 6. cgidecode.py benchmark

	P	R	Time (s)
Panini	1.00	1.00	1.81
Mimid	1.00	<0.27	48.10
TreeVada	0.96	<0.18	661.28
TTT	0.64	<0.94	31.10

Panini is able to infer the true input language of cgidecode.py exactly, in 1.81 seconds, without requiring any prior knowledge of positive inputs.

Mimid can infer a subset of the input language for cgidecode.py, in 48.10 seconds, using 17 hand-selected known-positive input samples as additional information. Although Gopinath et al. [2020] report 100 % precision and recall for *Mimid* on this benchmark, this is based on an insufficient golden grammar, which does not include all possible two-digit hexadecimals and is limited to a subset of the ASCII alphabet, perhaps because the grammar representation used by *Mimid* cannot easily represent the entire Unicode alphabet [Zeller et al. 2024]. We reran their benchmark on an extended golden grammar, closer to the actual grammar but still limited to only printable ASCII characters, which revealed that the true recall of *Mimid*'s inferred grammar was at most 27 %.

TreeVada, using the same set of positive input samples provided by the *Mimid* benchmark for the training set, takes about 11 minutes to infer a slightly incorrect grammar, with a precision of 96 % and recall of at most 18 % (measured against the extended *Mimid* golden grammar).

TTT can generate an over-approximating DFA in about half a minute, when restricted to an alphabet of printable ASCII characters. To simulate equivalence queries, we again used the positive input samples included with the *Mimid* benchmark. The resulting grammar exhibits 94 % recall, compared to the extended *Mimid* golden grammar, but with only 64 % precision.

This comparison highlights that previous approaches are too coarse for accurate inference of regular ad hoc grammars. They are all highly dependent on good input samples to cover the state space of the parsing program, and are also generally limited to small subsets of the full Unicode alphabet. CFG-focused approaches additionally prioritize larger-scale inference of context-free properties, perhaps to the detriment of regular language accuracy. The necessity to execute the parsers during inference also naturally increases the inference times of these dynamic approaches.

7 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 2021]. It is usually assumed that collecting string constraints requires some kind of (dynamic) symbolic execution [Kausler and Sherman 2014], and practical SCS applications are generally concerned with the inverse of our problem: modeling the possible strings a function can return or express [Bultan et al. 2018], instead of the strings a function can accept. In our approach, we use purely static means (viz. refinement type inference) to essentially collect input string constraints (see § 3), which we then simplify/solve in ways not dissimilar but nonetheless different from traditional SCS techniques (see § 4). There have been many recent advances in SCS for SMT [Abdulla et al. 2015; Kan et al. 2022; Kiezun et al. 2009; Trinh et al. 2014, 2020; Zheng et al. 2013]. We particularly make use of string theories embedded in the Z3 constraint solver [Berzish et al. 2017] in our implementation (§ 6).

Related work in (dynamic) symbolic execution make use of constraint solvers over string domains at their core to reason about how strings are manipulated in programs, with applications ranging from generating test inputs [Bjørner et al. 2009; Cadar et al. 2008b; Li et al. 2011] and detecting vulnerabilities (e.g., cross-site scripting, SQL injection) [Holík et al. 2017; Loring et al. 2017; Saxena et al. 2010]. These approaches differ from our work on two specific aspects. First, they rely on traces from dynamic executions to infer more precise constraints, while we are able to reason about string constraints statically. Second, in the case of detecting vulnerabilities, they reason about the resulting output induced through string operations, and do not infer a grammar over the input language, which is our main goal.

Abstract Domains. Abstract string domains approximate strings to track information precisely enough to analyze particular behaviors of interest 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] introduces 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 et al. [2020] review the dashed string abstraction, an approach that considers strings as blocks of characters and the constraints on these blocks, which has shown good performance on benchmarks involving constraints on string length, equality, concatenation, and regular expression membership. M-String [Cortesi and Olliaro 2018] considers a parametric abstract domain for strings in the C programming language by leveraging abstract domains for the content of a string and for expressions to infer when a string index position corresponds to an expression of interest. While most of the existing work has focused on approximating a single variable, very recent work by Arceri et al. [2022] focuses 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.

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

Precondition Inference. Despite a wide variety of approaches for computing preconditions [Barnett and Leino 2005; Cousot et al. 2013; Dillig et al. 2013; Padhi et al. 2016; Seghir and Kroening 2013], we are not aware of any that focus specifically on string operations, or that would allow us to reconstruct an input grammar in a way suitable for our envisioned applications.

Data-Availability Statement

We have implemented our approach in the Panini prototype system, whose source repository will become publicly available upon acceptance and is currently viewable for reviewers in anonymized form at https://anonymous.4open.science/r/panini. This repository also includes our full evaluation dataset. We will additionally provide a self-contained artifact (most likely in the form of a Docker container) to easily reproduce all claims made in this paper.

References

Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2015. Norn: An SMT solver for string constraints. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Springer, 462–469.

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

Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. 2024. Benchmarking Combinations of Learning and Testing Algorithms for Automata Learning. *Form. Asp. Comput.* 36, 1, Article 3 (mar 2024), 37 pages. https://doi.org/10.1145/3605360

James F. Allen. 1983. Maintaining Knowledge about Temporal Intervals. *Commun. ACM* 26, 11 (nov 1983), 832–843. https://doi.org/10.1145/182.358434

Roberto Amadini. 2021. A Survey on String Constraint Solving. arXiv:2002.02376 [cs.AI]

Roberto Amadini, Graeme Gange, and Peter J. Stuckey. 2020. Dashed strings for string constraint solving. *Artificial Intelligence* 289 (2020), 103368. https://doi.org/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 *Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205*. Springer-Verlag, Berlin, Heidelberg, 41–57. https://doi.org/10.1007/978-3-662-54577-5_3

Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and Computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

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

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* (Philadelphia, PA, USA). Springer-Verlag, Berlin, Heidelberg, 20–42. https://doi.org/10.1007/978-3-030-94583-1_2

Dean N. Arden. 1961. Delayed-logic and finite-state machines. In *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961)*. 133–151. https://doi.org/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 *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 117, 12 pages. https://doi.org/10.1145/3597503.3639214

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

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

Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability* (2nd ed.). IOS Press, Chapter 33, 1267–1329.

Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 95–110. https://doi.org/10.1145/3062341.3062349

Bachir Bendrissou, Rahul Gopinath, and Andreas Zeller. 2022. "Synthesizing input grammars": a replication study. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 260–268. https://doi.org/10.1145/3519939.3523716

Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. 2017. Z3str3: A string solver with theory-aware heuristics. In *2017 Formal Methods in Computer Aided Design* (Vienna, Austria) *(FMCAD 2017)*. IEEE, 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

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

Saroja Bhate and Subhash Kak. 1991. Pāṇini's Grammar and Computer Science. *Annals of the Bhandarkar Oriental Research Institute* 72/73, 1/4 (1991), 79–94.

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*. Springer, 24–51.

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 Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer, 307–321.

William J. Bowman. 2022. The A Means A. 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 *Proceedings of the 22nd International Conference on Compiler Construction* (Rome, Italy) *(CC'13)*. Springer-Verlag, Berlin, Heidelberg, 102–122. https://doi.org/10.1007/978-3-642-37051-9_6

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

Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. 2018. *String Analysis for Software Verification and Security* (1st ed.). Springer Cham. https://doi.org/10.1007/978-3-319-68670-7

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008a. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008b. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38.

Manuel MT Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2004. A functional perspective on SSA optimisation algorithms. *Electronic Notes in Theoretical Computer Science* 82, 2 (2004), 347–361. https://doi.org/10.1016/S1571-0661(05)82596-4

N. Chomsky and M.P. Schützenberger. 1963. The Algebraic Theory of Context-Free Languages. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 118–161. https://doi.org/10.1016/S0049-237X(08)72023-8

Agostino Cortesi and Martina Olliaro. 2018. M-String Segmentation: A Refined Abstract Domain for String Analysis in C Programs. In *2018 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. 1–8. https://doi.org/10.1109/TASE.2018.00009

Benjamin Cosman and Ranjit Jhala. 2017. Local Refinement Typing. *PACM on Programming Languages* 1, ICFP, Article 26 (Aug. 2017), 27 pages. https://doi.org/10.1145/3110270

Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2015. A Suite of Abstract Domains for Static Analysis of String Values. *Softw. Pract. Exper.* 45, 2 (feb 2015), 245–287. https://doi.org/10.1002/spe.2218

Patrick Cousot. 1997. Types as Abstract Interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/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 *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/567752.567778

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. 2013. Automatic Inference of Necessary Preconditions. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation* (Rome, Italy) *(VMCAI 2013)*. Springer-Verlag, Berlin, Heidelberg, 128–148. https://doi.org/10.1007/978-3-642-35873-9_10

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '78 (POPL '78)*. ACM Press. https://doi.org/10.1145/512760.512770

Luis Damas and Robin Milner. 1982. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, New York, NY, USA, 443–456. https://doi.org/10.1145/2509136.2509511

Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. *Comput. Surveys* 54, 5, Article 98 (May 2021), 38 pages. https://doi.org/10.1145/3450952

Aryaz Eghbali and Michael Pradel. 2020. No strings attached: An empirical study of string-related software bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 956–967.

M Fitter and TRG Green. 1979. When do diagrams make good computer languages? *International Journal of man-machine studies* 11, 2 (1979), 235–261.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Roberto Giacobazzi and Elisa Quintarelli. 2001. Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking. In *Proceedings of the 8th International Symposium on Static Analysis (SAS '01)*. Springer-Verlag, Berlin, Heidelberg, 356–373.

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

E Mark Gold. 1967. Language identification in the limit. *Information and control* 10, 5 (1967), 447–474.

Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. ACM, New York, NY, USA, 172–183. https://doi.org/10.1145/3368089.3409679

R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (December 1969), 29–60.

Lukáš Holík, Petr Janků, Anthony W. Lin, Philipp Rümmer, and Tomáš Vojnar. 2017. String Constraints with Concatenation and Transducers Solved Efficiently. *Proc. ACM Program. Lang.* 2, POPL, Article 4 (dec 2017), 32 pages. https://doi.org/10.1145/3158092

John Hopcroft and Jeffrey Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley.

Matthias Höschele and Andreas Zeller. 2017. Mining Input Grammars with AUTOGRAM. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 31–34. https://doi.org/10.1109/ICSE-C.2017.14

Malte Isberner. 2015. *Foundations of Active Automate Learning: An Algorithmic Perspective.* Ph. D. Dissertation. TU Dortmund.

Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning. In *Runtime Verification*, Borzoo Bonakdarpour and Scott A. Smolka (Eds.). Springer International Publishing, Cham, 307–322.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for javascript.. In *SAS*, Vol. 9. Springer, 238–255.

Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. (2020). arXiv:2010.07763 [cs.PL]

Stephen C Johnson and Ravi Sethi. 1990. Yacc: A Parser Generator. *UNIX Vol. II: Research System* (1990), 347–374.

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

Shuanglong Kan, Anthony Widjaja Lin, Philipp Rümmer, and Micha Schrader. 2022. Certistr: a certified string solver. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 210–224.

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 *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 215–226.

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 *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.

Scott Kausler and Elena Sherman. 2014. Evaluation of String Constraint Solvers in the Context of Symbolic Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA, 259–270. https://doi.org/10.1145/2642937.2643003

Matthias Keil and Peter Thiemann. 2014. Symbolic Solving of Extended Regular Expression Inequalities. (2014). arXiv:1410.3227 [cs.FL]

Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 105–116.

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

Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 456–467. https://doi.org/10.1109/ASE51524.2021.9678879

Kevin J Lang. 1999. *Faster algorithms for finding minimal consistent DFAs*. Technical Report. NEC Research Institute, 4 Independence Way, Princeton, NJ.

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World*. Technical Report UU-CS-2001-35. Department of Information and Computing Sciences, Utrecht University. http://www.cs.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 14-20, 2011. Proceedings 23*. Springer, 609–615.

Tianyi Liang, Nestan Tsiskaridze, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2015. *A Decision Procedure for Regular Membership and Length Constraints over Unbounded Strings*. Springer International Publishing, 135–150. https://doi.org/10.1007/978-3-319-24246-0_9

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

Blake Loring, Duncan Mitchell, and Johannes Kinder. 2017. ExpoSE: practical symbolic execution of standalone JavaScript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. 196–199.

David MacQueen, Robert Harper, and John Reppy. 2020. The History of Standard ML. *PACM on Programming Languages* 4, HOPL, Article 86 (June 2020), 100 pages. https://doi.org/10.1145/3386336

Falcon 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 (SecDev)*. IEEE, 45–52.

Manuel Montenegro, Susana Nieva, Ricardo Peña, and Clara Segura. 2020. Extending Liquid Types to Arrays. *ACM Transactions on Computational Logic* 21, 2, Article 13 (Jan. 2020), 41 pages. https://doi.org/10.1145/3362740

Donald R. Morrison. 1968. PATRICIA — Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (oct 1968), 514–534. https://doi.org/10.1145/321479.321481

Charles Gregory Nelson. 1980. *Techniques for Program Verification*. Ph. D. Dissertation. Stanford University. A revised version was published in June 1981 by Xerox PARC as report number CSL-81-10.

Chris Okasaki and Andy Gill. 1998. Fast mergeable integer maps. In *ACM SIGPLAN Workshop on ML*. 77–86.

José Oncina and Pedro García. 1992. *Inferring Regular Languages in Polynomial Updated Time*. World Scientific, 49–61. https://doi.org/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*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). 437–450. https://doi.org/10.1007/1-4020-8141-3_34

Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 42–56. https://doi.org/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 *Proceedings of the 12th Symposium on Dynamic Languages*. 25–36.

Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-$LL(k)$ parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/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*. IEEE, 513–528.

Michael Schröder, Marc Goritschnig, and Jürgen Cito. 2023. An Exploratory Study of Ad Hoc Parsers in Python. arXiv:2304.09733 [cs.SE] https://arxiv.org/abs/2304.09733 Accepted as a registered report for MSR 2023 with Continuity Acceptance (CA).

Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-Guided Precondition Inference. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 451–471. https://doi.org/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*. Springer Berlin Heidelberg, 71–89. https://doi.org/10.1007/3-540-45013-0_7

Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1232–1243.

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 (2020), 1–27.

The Unicode Consortium. 2023. *The Unicode Standard, Version 15.1.0*. The Unicode Consortium, South San Francisco, CA. https://www.unicode.org/versions/Unicode15.1.0/

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Alessandro Warth and Ian Piumarta. 2007. OMeta: An Object-Oriented Language for Pattern Matching. In *Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, Quebec, Canada) *(DLS '07)*. 11–19.

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2024. Fuzzing with Grammars. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/html/Grammars.html Retrieved 2024-06-30 18:31:28+02:00.

Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 114–124.